Best implementation for hashCode method

Ask Question

How do we decide on the best implementation of `hashCode()` method for a collection (assuming that equals method has been overridden correctly) ?

java hash equals hashcode

edited May 7 '15 at 17:16
nbro
5,124 8 42 85

asked Sep 22 '08 at 6:53
Omnipotent
10.3k 10 25 33

1 with Java 7+, I guess `Objects.hashCode(collection)` should be a perfect solution! – Diablo Mar 17 '16 at 11:45

2 @Diablo I don't think that answers the question at all - that method simply returns `collection.hashCode()` (hg.openjdk.java.net/jdk7/jdk7/jdk/file/9b8c96f96a0f/src/share/…) – cbreezier Apr 19 '16 at 5:35

20 Answers

The best implementation? That is a hard question because it depends on the usage pattern.

A for nearly all cases reasonable good implementation was proposed in *Josh Bloch*'s ***Effective Java*** in Item 8 (second edition). The best thing is to look it up there because the author explains there why the approach is good.

A short version

1. Create a `int result` and assign a **non-zero** value.
2. For *every field* `f` tested in the `equals()` method, calculate a hash code `c` by:
   - If the field f is a `boolean` : calculate `(f ? 0 : 1)` ;
   - If the field f is a `byte` , `char` , `short` or `int` : calculate `(int)f` ;

   - If the field f is a `long` : calculate `(int)(f ^ (f >>> 32))` ;
   - If the field f is a `float` : calculate `Float.floatToIntBits(f)` ;
   - If the field f is a `double` : calculate `Double.doubleToLongBits(f)` and handle the return value like every long value;
   - If the field f is an *object*: Use the result of the `hashCode()` method or 0 if `f == null` ;
   - If the field f is an *array*: see every field as separate element and calculate the hash value in a *recursive fashion* and combine the values as described next.

3. Combine the hash value `c` with `result` :

   ```
   result = 37 * result + c
   ```

4. Return `result`

This should result in a proper distribution of hash values for most use situations.

edited Apr 2 at 16:19

answered Sep 22 '08 at 7:22

dmeister
21.6k 14 61 85

42 Yeah I'm particularly curious about where the number 37 comes from. – Kip Sep 22 '08 at 17:25

11 I'm not aware of any proof. The number of 37 is arbitrary, but it should be prime. Why? I'm not really sure but it has to do with modulo arthritics and properties of prime numbers which lead to go distributions. – dmeister Sep 22 '08 at 23:55

17 I used item 8 of Josh Bloch's "Effective Java" book. – dmeister Oct 4 '10 at 14:39

31 @dma_k The reason for using prime numbers and the method described in this answer is to ensure that the **computed hashcode will be unique**. When using non-prime numbers, you cannot guarantee this. It does not matter which prime nummer you choose, there is nothing magical about the number 37 (too bad 42 isn't a prime number, eh?) – Simon Forsberg Feb 15 '13 at 13:58

34 @SimonAndréForsberg Well, computed hash code cannot be always unique :) Is a hashcode. However I got the idea: the prime number has only one multiplier, while non-prime has at least two. That creates an extra combination for multiplication operator to result the same hash, i.e. cause collision. – dma_k Feb 15 '13 at 14:08

If you're happy with the Effective Java implementation recommended by dmeister, you can use a library call instead of rolling your own:

```
@Override
public int hashCode(){
    return Objects.hashCode(this.firstName, this.lastName);
}
```

This requires either guava (`com.google.common.base.Objects.hashCode(...)`) or JDK7 (`java.util.Objects.hash(...)`) but works the same way.

edited Aug 5 '13 at 20:03

answered Aug 5 '13 at 19:49
bacar
5,848 8 40 67

8 Unless one has a good reason not to use these, one should definitely use these in any case. (Formulating it stronger, as it IMHO should be formulated.) The typical arguments for using standard implementations/libraries apply (best practices, well tested, less error prone, etc). – Kissaki Jan 28 '14 at 13:23

7 @justin.hughey you seem to be confused. The only case you should override `hashCode` is if you have a custom `equals` , and that is precisely what these library methods are designed for. The documentation is quite clear on their behaviour in relation to `equals` . A library implementation does not claim to absolve you from knowing what the characteristics of a correct `hashCode` implementation are - these libraries make it *easier* for you to implement such a conforming implementation for the majority of cases where `equals` is overriden. – bacar Mar 11 '14 at 0:06

6 For any Android developers looking at the java.util.Objects class, it was only introduced in API 19, so make sure you're running on KitKat or above otherwise you'll get NoClassDefFoundError. – Andrew Kelly Feb 5 '15 at 5:30

1 Best answer IMO, although by way of example I would rather have chosen the JDK7 `java.util.Objects.hash(...)` method than the guava `com.google.common.base.Objects.hashCode(...)` method. I think most people would choose the standard library over an extra dependency. – Malte Skoruppa Nov 4 '15 at 19:00

2 If there are two arguments or more and if any of them is an array, the result might be not what you expect because `hashCode()` for an array is just its `java.lang.System.identityHashCode(...)` . – starikoff Dec 21 '15 at 11:48

It is better to use the functionality provided by Eclipse which does a pretty good job and you can put your efforts and energy in developing the business logic.

answered Nov 29 '08 at 6:30
Warrior
22k 36 122 195

4 +1 A good practical solution. dmeister's solution is more comprehensive, but I tend to forget to handle nulls when I try to write hashcodes myself. – Quantum7 Apr 7 '11 at 0:31

1 +1 Agree with Quantum7, but I would say it's also really good to understand what the Eclipse-generated implementation is doing, and where it gets its implementation details from. – jwir3 Jan 27 '14 at 21:05

10 Sorry but answers involving "functionality provided by [some IDE]" are not really relevant in the context of the programming language in general. There are dozens of IDEs and this does not answer the question... namely because this is more about algorithmic determination and directly associated to equals() implementation - something an IDE will know nothing about. – Darrell Teague Jun 3 '16 at 18:59

Although this is linked to `Android` documentation (Wayback Machine) and My own code on Github, it will work for Java in general. My answer is an extension of dmeister's Answer with just code that is much easier to read and understand.

```
@Override
public int hashCode() {

    // Start with a non-zero constant. Prime is preferred
    int result = 17;

    // Include a hash for each field.

    // Primatives

    result = 31 * result + (booleanField ? 1 : 0);                   // 1 bit   »
32-bit

    result = 31 * result + byteField;                                // 8 bits  »
32-bit
    result = 31 * result + charField;                                // 16 bits »
32-bit
    result = 31 * result + shortField;                               // 16 bits »
32-bit
    result = 31 * result + intField;                                 // 32 bits »
32-bit

    result = 31 * result + (int)(longField ^ (longField >>> 32));    // 64 bits »
32-bit
```

```
    result = 31 * result + Float.floatToIntBits(floatField);         // 32 bits »
32-bit

    long doubleFieldBits = Double.doubleToLongBits(doubleField);     // 64 bits
(double) » 64-bit (long) » 32-bit (int)
    result = 31 * result + (int)(doubleFieldBits ^ (doubleFieldBits >>> 32));

    // Objects

    result = 31 * result + Arrays.hashCode(arrayField);              // var bits
» 32-bit

    result = 31 * result + referenceField.hashCode();                // var bits
» 32-bit (non-nullable)
    result = 31 * result +                                           // var bits
» 32-bit (nullable)
        (nullableReferenceField == null
            ? 0
            : nullableReferenceField.hashCode());

    return result;

}
```

EDIT

Typically, when you override `hashcode(...)` , you also want to override `equals(...)` . So for those that will or has already implemented `equals` , here is a good reference from my Github...

```
@Override
public boolean equals(Object o) {

    // Optimization (not required).
    if (this == o) {
        return true;
    }

    // Return false if the other object has the wrong type, interface, or is
null.
    if (!(o instanceof MyType)) {
        return false;
    }

    MyType lhs = (MyType) o; // lhs means "left hand side"

            // Primitive fields
    return     booleanField == lhs.booleanField
```

```
            && byteField    == lhs.byteField
            && charField    == lhs.charField
            && shortField   == lhs.shortField
            && intField     == lhs.intField
            && longField    == lhs.longField
            && floatField   == lhs.floatField
            && doubleField  == lhs.doubleField

            // Arrays

            && Arrays.equals(arrayField, lhs.arrayField)

            // Objects

            && referenceField.equals(lhs.referenceField)
            && (nullableReferenceField == null
                        ? lhs.nullableReferenceField == null
                        :
nullableReferenceField.equals(lhs.nullableReferenceField));
}
```

edited Aug 5 '17 at 15:01

answered Jul 4 '15 at 11:45
Christopher Rucinski
3,121 2 17 43

1 Android Documentation now does not include the above code anymore, so here is a cached version from the Wayback Machine - Android Documentation (Feb 07, 2015) – Christopher Rucinski Jul 18 '17 at 9:14

perfect code snippet – ZhiXingZhe - WangYuQi Apr 4 at 14:31

First make sure that equals is implemented correctly. From an IBM DeveloperWorks article:

> - Symmetry: For two references, a and b, a.equals(b) if and only if b.equals(a)
> - Reflexivity: For all non-null references, a.equals(a)
> - Transitivity: If a.equals(b) and b.equals(c), then a.equals(c)

Then make sure that their relation with hashCode respects the contact (from the same article):

> - Consistency with hashCode(): Two equal objects must have the same hashCode() value

Finally a good hash function should strive to approach the ideal hash function.

answered Sep 22 '08 at 7:08
Grey Panther
10.3k 6 34 59

---

about8.blogspot.com, you said

> if equals() returns true for two objects, then hashCode() should return the same value. If equals() returns false, then hashCode() should return different values

I cannot agree with you. If two objects have the same hashcode it doesn't have to mean that they are equal.

If A equals B then A.hashcode must be equal to B.hascode

but

if A.hashcode equals B.hascode it does not mean that A must equals B

edited Aug 31 '17 at 13:49
Attila
1,952 1 17 25

answered Sep 22 '08 at 8:47
panzupa

perfect..thanks for pointing out that.. – Omnipotent Sep 22 '08 at 8:55

3 If `(A != B) and (A.hashcode() == B.hashcode())` , that's what we call hash function collision. It's because hash function's codomain is always finite, while it's domain is usually not. The bigger the codomain is, the less often the collision should occur. Good hash function's should return different hashes for different objects with greatest possibility achievable given particular codomain size. It can rarely be fully guaranteed though. – Krzysztof Jabłoński Apr 29 '13 at 8:45

This should just be a comment to the above post to Grey. Good information but it does not really answer the question – Christopher Rucinski Sep 15 '15 at 11:55

Good comments but be careful about using the term 'different objects' ... because equals() and thus the hashCode() implementation are not necessarily about different objects in an OO context but are usually more about their domain model representations (e.g., two people can be considered the same if they share a country code and country ID - though these may be two different 'objects' in a JVM - they are considered 'equal' and having a given hashCode)... – Darrell Teague Jun 3 '16 at 19:04

There's a good implementation of the *Effective Java*'s `hashcode()` and `equals()` logic in Apache Commons Lang. Checkout HashCodeBuilder and EqualsBuilder.

edited May 7 '15 at 17:30
nbro
5,124 8 42 85

answered Sep 22 '08 at 8:35
Rudi Adianto
232 1 6

1 The downside of this API is you pay the cost of object construction every time you call equals and hashcode (unless your object is immutable and you precompute the hash), which can be a lot in certain cases. – James McMahon Feb 4 '12 at 1:35

this was my favorite approach, until recently. I have ran into StackOverFlowError while using a criteria for SharedKey OneToOne association. More over, `Objects` class provides `hash(Object ..args)` & `equals()` methods from Java7 on. These are recommended for any applications using jdk 1.7+ – Diablo Mar 17 '16 at 11:26

@Diablo I guess, your problem was a cycle in the object graph and then you're out of luck with most implementation as you need to ignore some reference or to break the cycle (mandating an `IdentityHashMap`). FWIW I use an id-based hashCode and equals for all entities. – maaartinus Dec 10 '17 at 19:12

If you use eclipse, you can generate `equals()` and `hashCode()` using:

> Source -> Generate hashCode() and equals().

Using this function you can decide *which fields* you want to use for equality and hash code calculation, and Eclipse generates the corresponding methods.

edited May 7 '15 at 17:26
nbro
5,124 8 42 85

answered Sep 22 '08 at 12:50
Johannes K. Lehnert
689 1 8 9

Just a quick note for completing other more detailed answer (in term of code):

If I consider the question how-do-i-create-a-hash-table-in-java and especially the jGuru FAQ entry, I believe some other criteria upon which a hash code could be judged are:

- synchronization (does the algo support concurrent access or not) ?
- fail safe iteration (does the algo detect a collection which changes during iteration)
- null value (does the hash code support null value in the collection)

edited May 23 '17 at 12:02
Community ♦
1 1

answered Sep 22 '08 at 7:08
VonC
784k 266 2430 2905

1 weird that such important aspects were not upvoted. – Eugene Sep 11 '12 at 8:20

If I understand your question correctly, you have a custom collection class (i.e. a new class that extends from the Collection interface) and you want to implement the hashCode() method.

If your collection class extends AbstractList, then you don't have to worry about it, there is already an implementation of equals() and hashCode() that works by iterating through all the objects and adding their hashCodes() together.

```
    public int hashCode() {
       int hashCode = 1;
       Iterator i = iterator();
       while (i.hasNext()) {
         Object obj = i.next();
         hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
       }
   return hashCode;
    }
```

Now if what you want is the best way to calculate the hash code for a specific class, I normally use the ^ (bitwise exclusive or) operator to process all fields that I use in the equals method:

```
public int hashCode(){
   return intMember ^ (stringField != null ? stringField.hashCode() : 0);
}
```

answered Sep 22 '08 at 7:16
Mario Ortegón
12k 14 64 79

@about8 : there is a pretty serious bug there.

```
Zam obj1 = new Zam("foo", "bar", "baz");
Zam obj2 = new Zam("fo", "obar", "baz");
```

same hashcode

you probably want something like

```
public int hashCode() {
    return (getFoo().hashCode() + getBar().hashCode()).toString().hashCode();
```

(can you get hashCode directly from int in Java these days? I think it does some autocasting.. if that's the case, skip the toString, it's ugly.)

answered Sep 22 '08 at 7:06
SquareCog
16.8k 5 42 59

I looked at the question twice but didn't find a bug there... – Huppie Sep 22 '08 at 7:25

3 the bug is in the long answer by about8.blogspot.com -- getting the hashcode from a concatenation of strings leaves you with a hash function that is the same for any combination of strings that add up to the same string. – SquareCog Sep 22 '08 at 13:53

1 So this is meta-discussion and not related to the question at all? ;-) – Huppie Sep 22 '08 at 17:40

1 It's a correction to a proposed answer that has a fairly significant flaw. – SquareCog Sep 22 '08 at 22:13

This is a very limited implementation – Christopher Rucinski Sep 15 '15 at 11:58

As you specifically asked for collections, I'd like to add an aspect that the other answers haven't mentioned yet: A HashMap doesn't expect their keys to change their hashcode once they are added to the collection. Would defeat the whole purpose...

answered Sep 22 '08 at 7:15
Olaf Kock
35.8k 7 42 75

Use the reflection methods on Apache Commons EqualsBuilder and HashCodeBuilder.

edited May 7 '15 at 17:32
nbro
5,124 8 42 85

answered Sep 22 '08 at 10:16
Vihung
5,496 12 43 64

1 If you are going to use this be aware that reflection is expensive. I honestly wouldn't use this for anything besides throw away code. – James McMahon Feb 4 '12 at 1:31

any hashing method that evenly distributes the hash value over the possible range is a good implementation. See effective java (http://books.google.com.au/books?id=ZZOiqZQIbRMC&dq=effective+java&pg=PP1&ots=UZMZ2siN25&sig=kR0n73DHJOn-D77qGj0wOxAxiZw&hl=en&sa=X&oi=book_result&resnum=1&ct=result) , there is a good tip in there for hashcode implementation (item 9 i think...).

answered Sep 22 '08 at 7:20
Chii
11.2k 2 28 43

I prefer using utility methods fromm *Google Collections lib from class Objects* that helps me to keep my code clean. Very often `equals` and `hashcode` methods are made from IDE's template, so their are not clean to read.

edited May 7 '15 at 17:28
nbro
5,124 8 42 85

answered Sep 22 '08 at 8:51
panzupa

I use a tiny wrapper around `Arrays.deepHashCode(...)` because it handles arrays supplied as parameters correctly

```
public static int hash(final Object... objects) {
    return Arrays.deepHashCode(objects);
}
```

answered Dec 21 '15 at 12:08

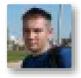
starikoff
860 10 20

Here is another JDK 1.7+ approach demonstration with superclass logics accounted. I see it as pretty convinient with Object class hashCode() accounted, pure JDK dependency and no extra manual work. Please note `Objects.hash()` is null tolerant.

I have not include any `equals()` implementation but in reality you will of course need it.

```
import java.util.Objects;

public class Demo {

    public static class A {

        private final String param1;

        public A(final String param1) {
            this.param1 = param1;
        }

        @Override
        public int hashCode() {
            return Objects.hash(
                super.hashCode(),
                this.param1);
        }

    }

    public static class B extends A {

        private final String param2;
        private final String param3;
        public B(
            final String param1,
            final String param2,
            final String param3) {

            super(param1);
            this.param2 = param2;
            this.param3 = param3;
        }

        @Override
        public final int hashCode() {
            return Objects.hash(
                super.hashCode(),
                this.param2,
                this.param3);
        }
    }

    public static void main(String [] args) {

        A a = new A("A");
        B b = new B("A", "B", "C");

        System.out.println("A: " + a.hashCode());
        System.out.println("B: " + b.hashCode());
    }

}
```

answered Dec 30 '16 at 13:18


Roman Nikitchenko
8,419 5 46 86

The standard implementation is weak and using it leads to unnecessary collisions. Imagine a

```
class ListPair {
    List<Integer> first;
    List<Integer> second;

    ListPair(List<Integer> first, List<Integer> second) {
        this.first = first;
        this.second = second;
    }
    public int hashCode() {
        return Objects.hashCode(first, second);
    }

    ...
}
```

Now,

```
new ListPair(List.of(a), List.of(b, c))
```

and

```
new ListPair(List.of(b), List.of(a, c))
```

have the same `hashCode` , namely `31*(a+b) + c` as the multiplier used for `List.hashCode` gets reused here. Obviously, collisions are unavoidable, but producing needless collisions is just... needless.

There's nothing substantially smart about using `31` . The multiplier must be odd in order to avoid losing information (any even multiplier loses at least the most significant bit, multiples of four lose two, etc.). Any odd multiplier is usable. Small multipliers may lead to faster computation (the JIT can use shifts and additions), but given that multiplication has latency of only three cycles on modern Intel/AMD, this hardly matters. Small multipliers also leads to more collision for small inputs, which may be a problem sometimes.

Using a prime is pointless as primes have no meaning in the ring Z/(2**32).

So, I'd recommend using a randomly chosen big odd number (feel free to take a prime). As i86/amd64 CPUs can use a shorter instruction for operands fitting in a single signed byte, there is a tiny speed advantage for multipliers like 109. For minimizing collisions, take something like 0x58a54cf5.

Using different multipliers in different places is helpful, but probably not enough to justify the additional work.

answered Dec 10 '17 at 18:02
maaartinus
25.4k 20 88 215

When combining hash values, I usually use the combining method that's used in the boost c++ library, namely:

```
seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
```

This does a fairly good job of ensuring an even distribution. For some discussion of how this formula works, see the StackOverflow post: Magic number in boost::hash_combine

There's a good discussion of different hash functions at: http://burtleburtle.net/bob/hash/doobs.html

edited May 23 '17 at 12:10
Community ♦
1 1

answered Oct 3 '12 at 15:18
Edward Loper
9,430 3 28 44

1 This is a question about Java, not C++. – dano Oct 10 '17 at 19:52

For a simple class it is often easiest to implement hashCode() based on the class fields which are checked by the equals() implementation.

```
public class Zam {
    private String foo;
    private String bar;
    private String somethingElse;

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }

        if (obj == null) {
            return false;
        }

        if (getClass() != obj.getClass()) {
            return false;
        }

        Zam otherObj = (Zam)obj;

        if ((getFoo() == null && otherObj.getFoo() == null) || (getFoo() != null
&& getFoo().equals(otherObj.getFoo()))) {
            if ((getBar() == null && otherObj. getBar() == null) || (getBar() !=
null && getBar().equals(otherObj. getBar()))) {
                return true;
            }
        }

        return false;
    }

    public int hashCode() {
        return (getFoo() + getBar()).hashCode();
    }

    public String getFoo() {
        return foo;
    }

    public String getBar() {
        return bar;
    }
}
```

The most important thing is to keep hashCode() and equals() consistent: if equals() returns true for two objects, then hashCode() should return the same value. If equals() returns false, then hashCode() should return different values.

edited Sep 22 '08 at 7:28

answered Sep 22 '08 at 7:00
Chris Carruthers
2,430 6 23 29

1 Like SquareCog have already noticed. If hashcode is generated once from concatenation of two strings it is extremely easy to generate masses of collisions: `("abc"+""=="ab"+"c"=="a"+"bc"==""+"abc")` . It is severe flaw. It would be better to evaluate hashcode for both fields and then calculate linear combination of them (preferably using primes as coefficients). – Krzysztof Jabłoński Apr 30 '13 at 6:34

@KrzysztofJabłoński Right. Moreover, swapping `foo` and `bar` produces a needless collision, too. – maaartinus Dec 10 '17 at 19:25