The 3 things you should know about hashCode()

In Java, every object has a method *hashCode* that is simple to understand but still it's sometimes forgotten or misused. Here are three things to keep in mind to avoid the common pitfalls.

An object's hash code allows algorithms and data structures to put objects into compartments, just like letter types in a printer's type case. The printer puts all "A" types into the compartment for "A", and he looks for an "A" only in this one compartment. This simple system lets him find types much faster than searching in an unsorted drawer. That's also the idea of hash-based collections, such as *HashMap* and *HashSet*.

Source: Wikimedia Commons

In order to make your class work properly with hash-based collections and other algorithms that rely on hash codes, all *hashCode* implementations must stick to a simple contract.

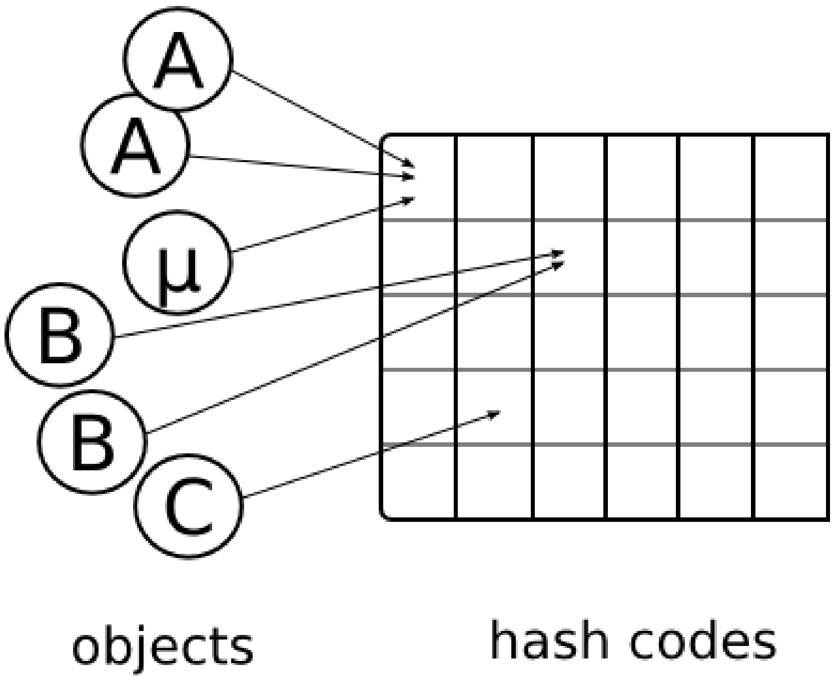
The hashCode contract

The contract is explained in the *hashCode* method's JavaDoc. It can be roughly summarized with this statement:

Objects that are equal must have the same hash code within a running process

Please note that this does not imply the following common misconceptions:

- Unequal objects must have different hash codes – **WRONG!**
- Objects with the same hash code must be equal – **WRONG!**



The contract allows for unequal objects to share the same hash code, such as the "A" and "µ" objects in the sketch above. In math terms, the mapping from objects to hash codes doesn't have to be injective or even bijective. This is obvious because the number of possible distinct objects is usually bigger than the number of possible hash codes (2^{32}).

Edit: *In an earlier version, I mistakenly stated that the hashCode mapping must be injective, but doesn't have to be bijective, which is obviously wrong. Thanks Lucian for pointing out this mistake!*

This contract directly leads to the first rule:

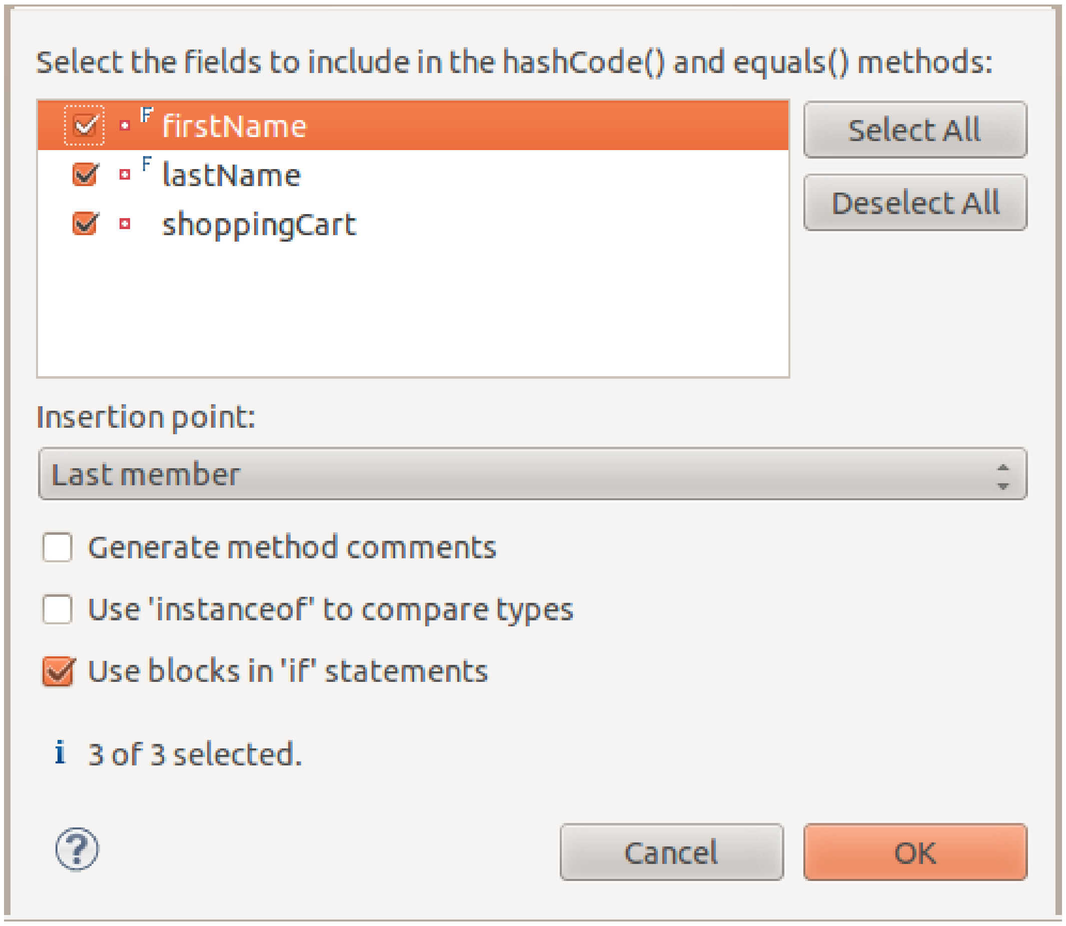
1. Whenever you implement *equals*, you MUST also implement *hashCode*

If you fail to do so, you will end up with broken objects. Why? An object's *hashCode* method must take the same fields into account as its *equals* method. By overriding the *equals* method, you're declaring some objects as equal to other objects, but the original *hashCode* method treats all objects as different. So you will have equal objects with different hash codes. For example, calling *contains()* on a *HashMap* will return *false*, even though the object has been added.

How to write a good hashCode function is beyond the scope of this article, it is perfectly explained in Joshua Bloch's popular book Effective Java, which should not be missing in a Java developer's bookshelf.

[Need expert advice for your project? Our Developer Support is here to resolve your questions. | Find more tips on how to write clean code on our Software Craftsmanship page.]

To be on the safe side, let the Eclipse IDE generate the *equals* and *hashCode* functions as a pair: *Source > Generate hashCode() and equals()….*


Select the fields to include in the hashCode() and equals() methods:

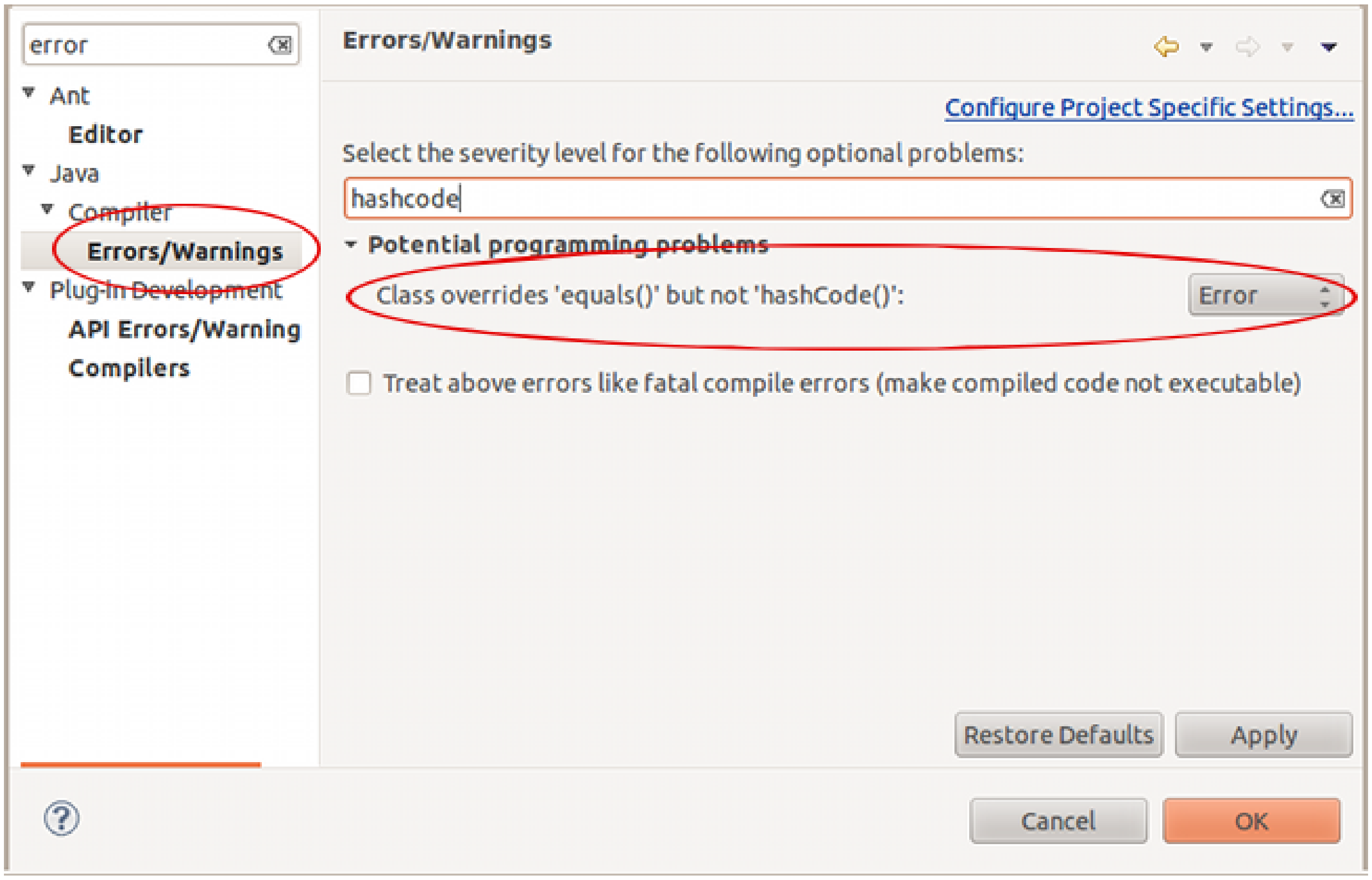
- [✓] firstName
- [✓] lastName
- [✓] shoppingCart

Insertion point: Last member

- [ ] Generate method comments
- [ ] Use 'instanceof' to compare types
- [✓] Use blocks in 'if' statements

3 of 3 selected.

To protect yourself, you can also configure Eclipse to detect violations of this rule and display errors for classes that implement *equals* but not *hashCode*. Unfortunately, this options is set to “Ignore” by default: *Preferences > Java > Compiler > Errors/Warnings*, then use the quick filter to search for “hashcode”:


error

Ant
Editor
Java
Compiler
Errors/Warnings
Plug-in Development
API Errors/Warning
Compilers

Errors/Warnings

Configure Project Specific Settings...

Select the severity level for the following optional problems:

hashcode

Potential programming problems

Class overrides 'equals()' but not 'hashCode()': Error

Treat above errors like fatal compile errors (make compiled code not executable)

***Update:** As laurent points out, the equalsverifier is a great tool to verify the contract of hashCode and equals. You should consider using it in your unit tests.*

HashCode collisions

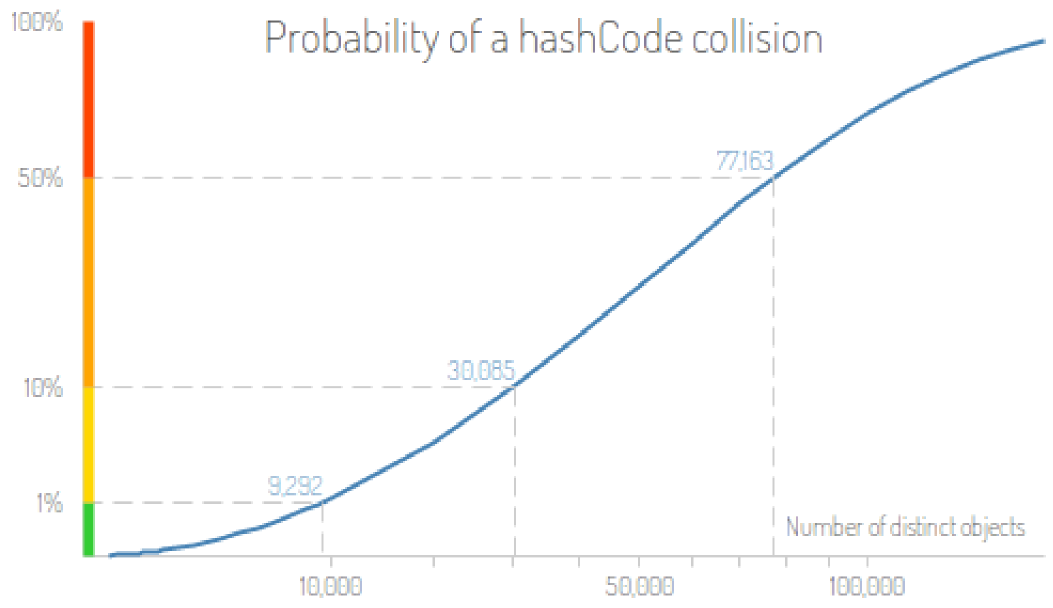
Whenever two different objects have the same hash code, we call this a collision. A collision is nothing critical, it just means that there is more than one object in a single bucket, so a HashMap lookup has to look again to find the right object. A lot of collisions will degrade the performance of a system, but they won't lead to incorrect results.

But if you mistake the hash code for a unique handle to an object, e.g use it as a key in a Map, then you will sometimes get the wrong object. Because even though collisions are rare, they are inevitable. For example, the Strings `"Aa"` and `"BB"` produce the same hashCode: `2112`. Therefore:

2. Never misuse hashCode as a key

You may object that, unlike the printer's type case, in Java there are 4,294,967,296 compartments (2^{32} possible *int* values). With 4 billion slots, collisions seem to be extremely unlikely, right?

Turns out that it's not so unlikely. Here's the surprising math of collisions: Please imagine 23 random people in a room. How would you estimate the odds of finding two fellows with the same birthday among them? Pretty low, because there are 365 days in a year? In fact, the odds are about 50%! And with 50 people it's a save bet. This phenomenon is called the Birthday paradox. Transferred to hash codes, this means that with 77,163 different objects, you have a 50/50 chance for a collision – given that you have an ideal *hashCode* function, that evenly distributes objects over all available buckets.



Example:

The Enron email dataset contains 520,924 emails. Computing the *String* hash codes of the email contents, I found 50 pairs (and even 2 triples) of different emails with the same hash code. For half a million strings, this is a pretty good result. But the message here is: if you have many data items, collisions will occur. If you were using the hashCode as a key here, you would not immediately notice your mistake. But a few people would get the wrong mail.

HashCodes can change

Finally, there's one important detail in the *hashCode* contract that can be quite surprising: *hashCode* does not guarantee the same result in different executions. Let's have a look at the JavaDoc:

> *"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."*

This is uncommon, in fact, some classes in the class library even specify the exact formula they use to calculate hash codes (e.g. *String*). For these classes, the hash code will always be the same. But while most of the *hashCode* implementations provide stable values, you must not rely on it. As this article points out, there are Java libraries that actually return different *hashCode* values in different processes and this tends to confuse people. Google's Protocol Buffers is an example.

Therefore, you should not use the hash code in distributed applications. A remote object may have a different hash code than a local one, even if the two are equal.

3. Do not use hashCode in distributed applications

Moreover, you should be aware that the implementation of a *hashCode* function may change from one version to another. Therefore your code should not depend on any particular hash code values. For example, your should not use the hash code to persist state. Next time you run the application, the hash codes of the "same" objects may be different.

The best advice is probably: **don't use hashCode at all**, except when you create hash-based algorithms.

An alternative: SHA1

You may know that cryptographic hash codes such as SHA1 are sometimes used to identify objects (Git does this, for example). Is this also unsafe? No. SHA1 uses 160-bit keys, which makes collisions virtually impossible. Even with a gigantic number of objects, the odds of a collision in this space are far below the odds of a meteor crashing the computer that runs your program. This article has a great overview of collision probabilities.

There's probably more to say about hash codes, but these seem to be the most important things. If there's anything I've missed, I'm happy to hear about it!

You may also like...

- A Fast and Minimal JSON Parser for Java
- when(true).throwIllegalArgument("something went wrong");

Share this Post

45 32

Tags

Java Software craftsmanship tips

Subscribe to Blog

7 Responses to "The 3 things you should know about hashCode()"

1. *Eric Jablow* says:
   September 5, 2012 at 00:11
   An important addition is: "Do not mutate objects that are members of HashSets, or are keys in HashMaps."

   Assume Person has the appropriate operations, equals, and hashCode methods.

```
Map telephoneNumbers = new HashMap();
Person jane = new Person(“Jane”, “Doe”);
System.out.println(jane); // Prints Jane Doe.
telephoneNumbers.put(jane, “555-1234″);
Person richard = new Person(“Richard”, “Roe”);
jane.marryTo(richard);
System.out.println(jane); // Now prints Jane Roe.
String janePhone = telephoneNumbers.get(jane); // Almost certainly returns null.
```

Why? The hashCode of jane changed when she married richard, but jane has not been moved to the appropriate hash bucket; the get does not find her.

If you have to do this and cannot change the Person class, remove jane from the hashMap before she marries, and add her back afterwards. It’s better to make Person immutable by making its fields final and removing the setters. You’ll still have to remove jane from HashMap and put janeRoe in. Or, act as though you have a database. Put a Long id into Person, and make your map be a HashMap.

2. *Eric Jablow* says:
   September 5, 2012 at 00:14
   Sorry–I forgot to escape the less than or greater than signs. Nothing would be confusing until the end–”make your map be a HashMap<Long, String>”.

3. *James* says:
   September 5, 2012 at 05:58
   I am a beginner and don’t have much experience but talking a about core Java, I never really understood equals and hashCode contract until I come across this article How HashMap works in Java and find out that both of these method internally used by HashMap.

4. *Ralf Sternberg* says:
   September 5, 2012 at 16:27
   @Eric That’s an interesting addition, thanks. In fact, when an object in a hash map is modified it becomes sort of a zombie. It’s in the wrong slot and can neither be found nor removed from the map, but still the map keeps a reference on it. Only a rehash operation triggered by adding some more objects can fix it.

That's a very good example for the perils of mutable state!

5. *Lucian* says:
September 6, 2012 at 08:31
"The contract allows for unequal objects to share the same hash code, such as the "A" and "µ" objects in the sketch above. In math terms, the mapping from objects to hash codes must be injective…"

   It's actually the opposite. An injective mapping needs to produce distinct values for distinct arguments, and as you stated the hash method doesn't do that.

   The hash method is not injective in most cases, it's just a mapping.

6. *laurent* says:
September 6, 2012 at 09:49
since you are describing this topic, you should check the equals verifier project on google code.

7. *Ralf Sternberg* says:
September 6, 2012 at 10:26
Congratulations, Lucian! You're the first one to notice this slip 😉 Of course, you're right. It's neither injective, nor surjective or anything. It's just a mapping. I'll update the post and fix this statement.

   Thanks for reading and pointing this out!

7 responses so far

Written by Ralf Sternberg. Published in Categories: EclipseSource News

Author:

Ralf Sternberg

Published:

Sep 4th, 2012

Bookmark:

The 3 things you should know about hashCode()

Follow:

Eclipse Training

- RCP Aufbaukurs - 3 Tage
  Karlsruhe: 25-27 Feb
- Eclipse 4 RCP Entwickler - 4 Tage
  Munich: 30 Mar-2 Apr
- Equinox und OSGi - 3 Tage
  Karlsruhe: 14-16 Apr
- Eclipse 4 für RCP Entwickler - 2 Tage
  Munich: 20-21 Apr
- Eclipse EMF Grundlagen - 2 Tage
  Munich: 4-5 May
- RAP für Java Entwickler - 4 Tage
  Karlsruhe: 4-7 May
- RAP für RCP Entwickler - 2 Tage
  Karlsruhe: 10-11 Jun
- RCP Aufbaukurs - 3 Tage

Karlsruhe: 18-20 Aug

- Equinox und OSGi - 3 Tage
  Karlsruhe: 8-10 Sep
- RAP für Java Entwickler - 4 Tage
  Karlsruhe: 12-15 Oct

Popular Posts

- Eclipse proxies and p2
- Dark Theme, Top Eclipse Luna Feature #5
- The 3 things you should know about hashCode()
- Top 10 Eclipse Luna Features
- 10 Tips for using the Eclipse Memory Analyzer
- My Top 10 Tips on how to be more productive with the Eclipse IDE
- Eclipse Preferences You Need to Know
- Serial Communication in Java with Raspberry Pi and RXTX
- Effective Java Debugging with Eclipse
- A Fast and Minimal JSON Parser for Java

Recent Posts

- Tabris.js 0.9.0 is here
- Mobile App Development in JavaScript — Three Minute Tutorial
- OpenSource Week in the Alps
- Working with modules and libraries in Tabris.js
- Fast Development Roundtrip: Mobile Apps with Tabris.js
- Native Mobile Apps in JavaScript with Tabris.js
- Call for Submission: Modeling Symposium @ EclipseCon North America 2015
- Eclipse December Democamp Munich 2014 – Retrospective
- EMF Forms 1.4.0 Feature #1: Support for alternative UI Technologies
- RAP 3.0 Release Schedule Update

search

About EclipseSource
Imprint
Privacy
Terms of Use
Contact Us