Guide to hashCode() in Java

Last modified: August 6, 2017

by baeldung (/author/baeldung/)

Java (/category/java/)

1. Overview

Hashing is a fundamental concept of computer science.

In Java, efficient hashing algorithms stand behind some of the most popular collections we have available – such as the *HashMap (https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)* (for an in-depth look at *HashMap*, feel free to check this article (/java-hashmap)) and the *HashSet (https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html).*

In this article, we'll focus on how *hashCode()* works, how it plays into collections and how to implement it correctly.

2. Usage of *hashCode()* in Data Structures

The simplest operations on collections can be inefficient in certain situations.

For example, this triggers a linear search which is highly ineffective for lists of huge sizes:

```java
List<String> words = Arrays.asList("Welcome", "to", "Baeldung");
if (words.contains("Baeldung")) {
    System.out.println("Baeldung is in the list");
}
```

Java provides a number of data structures for dealing with this issue specifically – for example, several *Map* interface implementations are hash tables. (https://en.wikipedia.org/wiki/Hash_table)

When using a hash table, **these collections calculate the hash value for a given key using the *hashCode()* method** and use this value internally to store the data – so that access operations are much more efficient.

3. Understanding How *hashCode()* Works

Simply put, *hashCode()* returns an integer value, generated by a hashing algorithm.

Objects that are equal (according to their *equals()*) must return the same hash code. **It's not required for different objects to return different hash codes.**

The general contract of *hashCode()* states:

- Whenever it is invoked on the same object more than once during an execution of a Java application, *hashCode()* must consistently return the same value, provided no information used in equals comparisons on the object is modified. This value needs 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 value
- It is not required that if two objects are unequal according to the ***equals(java.lang.Object)*** **(https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29)** method, then calling the *hashCode* method on each of the two objects must produce distinct integer results. However, developers should be aware that producing distinct integer results for unequal objects improves the performance of hash tables

> "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.)"

4. A Naive *hashCode()* Implementation

It's actually quite straightforward to have a naive *hashCode()* implementation that fully adheres to the above contract.

To demonstrate this, we're going to define a sample *User* class that overrides the method's default implementation:

```java
public class User {

    private long id;
    private String name;
    private String email;

    // standard getters/setters/constructors

    @Override
    public int hashCode() {
        return 1;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null) return false;
        if (this.getClass() != o.getClass()) return false;
        User user = (User) o;
        return id != user.id
          && (!name.equals(user.name)
          && !email.equals(user.email));
    }

    // getters and setters here
}
```

The *User* class provides custom implementations for both *equals()* and *hashCode()* that fully adhere to the respective contracts. Even more, there's nothing illegitimate with having *hashCode()* returning any fixed value.

However, this implementation degrades the functionality of hash tables to basically zero, as every object would be stored in the same, single bucket.

In this context, a hash table lookup is performed linearly and does not give us any real advantage – more on this in section 7.

5. Improving the *hashCode()* Implementation

Let's improve a little bit the current *hashCode()* implementation by including all fields of the *User* class so that it can produce different results for unequal objects:

```java
@Override
public int hashCode() {
    return (int) id * name.hashCode() * email.hashCode();
}
```

This basic hashing algorithm is definitively much better than the previous one, as it computes the object's hash code by just multiplying the hash codes of the *name* and *email* fields and the *id*.

In general terms, we can say that this is a reasonable *hashCode()* implementation, as long as we keep the *equals()* implementation consistent with it.

6. Standard *hashCode()* Implementations

The better the hashing algorithm that we use to compute hash codes, the better will the performance of hash tables be.

Let's have a look at a "standard" implementation that uses two primes numbers to add even more uniqueness to computed hash codes:

```java
@Override
public int hashCode() {
    int hash = 7;
    hash = 31 * hash + (int) id;
    hash = 31 * hash + (name == null ? 0 : name.hashCode());
    hash = 31 * hash + (email == null ? 0 : email.hashCode());
    return hash;
}
```

While it's essential to understand the roles that *hashCode()* and *equals()* methods play, we don't have to implement them from scratch every time, as most IDEs can generate custom *hashCode()* and *equals()* implementations and since Java 7, we got an *Objects.hash()* utility method for comfortable hashing:

```java
Objects.hash(name, email)
```

IntelliJ IDEA (https://www.jetbrains.com/idea/) generates the following implementation:

```java
@Override
public int hashCode() {
    int result = (int) (id ^ (id >>> 32));
    result = 31 * result + name.hashCode();
    result = 31 * result + email.hashCode();
    return result;
}
```

And Eclipse (https://www.eclipse.org/downloads/?) produces this one:

```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((email == null) ? 0 : email.hashCode());
    result = prime * result + (int) (id ^ (id >>> 32));
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}
```

In addition to the above IDE-based *hashCode()* implementations, it's also possible to automatically generate an efficient implementation, for example using Lombok (https://projectlombok.org/features/EqualsAndHashCode). In this case, the lombok-maven (https://search.maven.org/#search%7Cga%7C1%7Clombok) dependency must be added to *pom.xml*:

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok-maven</artifactId>
    <version>1.16.18.0</version>
    <type>pom</type>
</dependency>
```

It's now enough to annotate the *User* class with *@EqualsAndHashCode*:

```java
@EqualsAndHashCode
public class User {
    // fields and methods here
}
```

Similarly, if we want Apache Commons Lang's *HashCodeBuilder* class (https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/builder/HashCodeBuilder.html) to generate a *hashCode()* implementation for us, the commons-lang (https://search.maven.org/#search%7Cga%7C1%7Capache-commons-lang) Maven dependency must be included in the pom file:

```xml
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
```

And *hashCode()* can be implemented like this:

```java
public class User {
    public int hashCode() {
        return new HashCodeBuilder(17, 37).
        append(id).
        append(name).
        append(email).
        toHashCode();
    }
}
```

In general, there's no universal recipe to stick to when it comes to implementing *hashCode()*. We highly recommend reading Joshua Bloch's Effective Java (https://www.amazon.com/Effective-Java-3rd-Joshua-Bloch/dp/0134685997) , which provides a list of thorough guidelines (https://es.slideshare.net/MukkamalaKamal/joshua-bloch-effect-java-chapter-3) for implementing efficient hashing algorithms.

What can be noticed here is that all those implementations utilize number 31 in some form – this is because 31 has a nice property – its multiplication can be replaced by a bitwise shift which is faster than the standard multiplication:

```java
31 * i == (i << 5) - i
```

7. Handling Hash Collisions

The intrinsic behavior of hash tables raises up a relevant aspect of these data structures: even with an efficient hashing algorithm, two or more objects might have the same hash code, even if they're unequal. So, their hash codes would point to the same bucket, even though they would have different hash table keys.

This situation is commonly known as a hash collision, and various methodologies exist for handling it (https://www.kullabs.com/classes/subjects/units/lessons/notes/note-detail/3888), with each one having their pros and cons. Java's *HashMap* uses the separate chaining method (https://en.wikipedia.org/wiki/Hash_table#Separate_chaining_with_linked_lists) for handling collisions:

"When two or more objects point to the same bucket, they're simply stored in a linked list. In such a case, the hash table is an array of linked lists, and each object with the same hash is appended to the linked list at the bucket index in the array.

In the worst case, several buckets would have a linked list bound to it, and the retrieval of an object in the list would be performed linearly."

Hash collision methodologies show in a nutshell why it's so important to implement *hashCode()* efficiently.

Java 8 brought an interesting enhancement to *HashMap* implementation (http://openjdk.java.net/jeps/180) – if a bucket size goes beyond the certain threshold, the linked list gets replaced with a tree map. This allows achieving *O(*logn*)* look up instead of pessimistic *O(n)*.

8. Creating a Trivial Application

To test the functionality of a standard *hashCode()* implementation, let's create a simple Java application that adds some *User* objects to a *HashMap* and uses SLF4J (/slf4j-with-log4j2-logback) for logging a message to the console each time the method is called.

Here's the sample application's entry point:

```java
public class Application {

    public static void main(String[] args) {
        Map<User, User> users = new HashMap<>();
        User user1 = new User(1L, "John", "john@domain.com");
        User user2 = new User(2L, "Jennifer", "jennifer@domain.com");
        User user3 = new User(3L, "Mary", "mary@domain.com");

        users.put(user1, user1);
        users.put(user2, user2);
        users.put(user3, user3);
        if (users.containsKey(user1)) {
            System.out.print("User found in the collection");
        }
    }
}
```

And this is the *hashCode()* implementation:

```java
public class User {

    // ...

    public int hashCode() {
        int hash = 7;
        hash = 31 * hash + (int) id;
        hash = 31 * hash + (name == null ? 0 : name.hashCode());
        hash = 31 * hash + (email == null ? 0 : email.hashCode());
        logger.info("hashCode() called - Computed hash: " + hash);
        return hash;
    }
}
```

The only detail worth stressing here is that each time an object is stored in the hash map and checked with the *containsKey()* method, *hashCode()* is invoked and the computed hash code is printed out to the console:

```
[main] INFO com.baeldung.entities.User - hashCode() called - Computed hash: 1255477819
[main] INFO com.baeldung.entities.User - hashCode() called - Computed hash: -282948472
[main] INFO com.baeldung.entities.User - hashCode() called - Computed hash: -1540702691
[main] INFO com.baeldung.entities.User - hashCode() called - Computed hash: 1255477819
User found in the collection
```

9. Conclusion

It's clear that producing efficient *hashCode()* implementations often requires a mixture of a few mathematical concepts, (i.e. prime and arbitrary numbers), logical and basic mathematical operations.

Regardless, it's entirely possible to implement *hashCode()* effectively without resorting to these techniques at all, as long as we make sure the hashing algorithm produce different hash codes for unequal objects and is consistent with the implementation of *equals()*.

As always, all the code examples shown in this article are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java/src/main/java/com/baeldung/hashcode).

Subscribe ▾

▴ newest ▴ oldest ▴ most voted

ZW++

Guest

Please update this post to mention Objects.hash from standard library among options to implement hashCode method. For User class from example it is just Objects.hash(id, name, email) .

+ 0 −

10 months ago

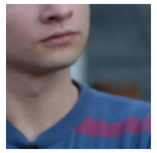
Grzegorz Piwowarek

Guest

Sure, added – thanks!

+ 0 −

10 months ago



radociech

Guest

It's also worth to mention about improvement in Java 8 HashMap, in case of high-collision prone hash code function entires may be stored in tree instead of linked list. See the TREEIFY_THRESHOLD. The explanation why prime numbers are used to implement hashCode can also be useful.

+ 0 −

10 months ago

Alejandro Gervasio

Guest

Hi radociech,

Thanks for the insightful comment., Yes, both the explanations about why most hashing algorithms use the number 31 and how hash collision handling has been improved in Java 8 with balanced trees were included in the article..

+ 0 −

10 months ago

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JACKSON/)
HTTPCLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)