Guide to hashCode() in Java

Last updated: May 6, 2023

Written by: baeldung

Java +

Core Java

1. Overview

Hashing is a fundamental concept of computer science.

In Java, efficient hashing algorithms stand behind some of the most popular collections, such as the *HashMap* (check out this in-depth article) and the *HashSet*.

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

Further reading:

Java equals() and hashCode() Contracts

Learn about the contracts that equals() and hasCode() need to fulfill and the relationship between the two methods

Read more →

Generate equals() and hashCode() with Eclipse

A quick and practical guide to generating equals() and hashcode() with the Eclipse IDE

Read more →

Introduction to Project Lombok

A comprehensive and very practical introduction to many useful usecases of Project Lombok on standard Java code.

Read more →

2. Using *hashCode()* in Data Structures

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

To illustrate, this triggers a linear search, which is highly ineffective for huge lists:

```
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.

When using a hash table, **these collections calculate the hash value for a given key using the *hashCode()* method.** Then they 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. **Different objects do not need 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 doesn't need to stay 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, calling the *hashCode()* method on each of the two objects must produce the same value.
- If two objects are unequal according to the ***equals(java.lang.Object)*** method, calling the *hashCode* method on each of the two objects doesn't need to 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

A naive *hashCode()* implementation that fully adheres to the above contract is actually quite straightforward.

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

```
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. We talk more about this in Section 7.

5. Improving the *hashCode()* Implementation

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

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

This basic hashing algorithm is definitively much better than the previous one. This is because 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 the performance of hash tables.

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

```
@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 we need to understand the roles that *hashCode()* and *equals()* methods play, we don't have to implement them from scratch every time. This is because most IDEs can generate custom *hashCode()* and *equals()* implementations. And since Java 7, we have an *Objects.hash()* utility method for comfortable hashing:

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

IntelliJ IDEA generates the following implementation:

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

And Eclipse produces this one:

```
@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.

In this case, we need to add the lombok-maven dependency to *pom.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*:

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

Similarly, if we want Apache Commons Lang's *HashCodeBuilder* class to

generate a *hashCode()* implementation for us, we include the commons-lang Maven dependency in the pom file:

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

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

```
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 when it comes to implementing *hashCode()*. We highly recommend reading Joshua Bloch's Effective Java. It provides a list of thorough guidelines for implementing efficient hashing algorithms.

Notice here 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:

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

7. Handling Hash Collisions

The intrinsic behavior of hash tables brings 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 methods exist for handling it, with each one having their pros and cons. Java's *HashMap* uses the separate chaining method 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. If a bucket size goes beyond the certain threshold, a tree map replaces the linked list. This allows achieving *O(*logn*)* lookup instead of pessimistic *O(n)*.

8. Creating a Trivial Application

Now we'll 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 for logging a message to the console each time the method is called.

Here's the sample application's entry point:

```
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:

```
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;
    }
}
```

Here, it's important to note 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, we can implement *hashCode()* effectively without resorting to these techniques at all. We just need to make sure the hashing algorithm produces different hash codes for unequal objects and that it's consistent with the implementation of *equals()*.

As always, all the code examples shown in this article are available over on GitHub.

Comments are closed on this article!

JACKSON JSON TUTORIAL

APACHE HTTPCLIENT TUTORIAL

REST WITH SPRING TUTORIAL

SPRING PERSISTENCE TUTORIAL

SECURITY WITH SPRING

SPRING REACTIVE TUTORIALS

ABOUT

ABOUT BAELDUNG

THE FULL ARCHIVE

EDITORS

JOBS

OUR PARTNERS

PARTNER WITH BAELDUNG

TERMS OF SERVICE

PRIVACY POLICY

COMPANY INFO

CONTACT