CS 240: Implementing a Hashcode Method Transcript

[00:00:00] **JEROD WILKERSON:** In the lecture on method overwriting, I mentioned three methods that are commonly overwritten: toString, equals, and hashCode. I described toString and equals, but hashCode is little more involved and requires a little more explanation, and it requires you to remember something from 235.

[00:00:18] If you think back to CS 235 when you took that, you learned about hash tables. Remember that hash tables are a way to create an efficient data structure that you can put objects in and then you can find an object later without needing to scan through all the values in the structure.

[00:00:39] The way that works is you somehow convert the object to an integer and then you use that integer to determine a positioning for the object inside of the underlying data structure.

[00:00:50] That's the underlying concept. In order to do that, we have to have some method that will represent an object as an integer, and that's what hashCode does.

[00:01:01] It's very common to override hashCode inside of some object that you create so instances of that class can be efficiently placed in hashing data structures.

[00:01:14] You typically don't have to write those data structures yourself, there are several that are already part of Java. When you download Java, you get them for free.

[00:01:22] We typically don't write a hash table, for example, but in order to efficiently place objects inside of a hash table, we have to have a hashCode that works appropriately.

[00:01:32] We need to understand, first of all, what does it mean for a hashCode method to work appropriately, and then we need to know how to specifically write one. We need to understand the contract of a hashCode.

[00:01:45] When you write a hashCode method, you must follow a couple of rules. There are two rules here that you have to follow, and then there's a third rule that's really saying the third thing is not a rule.

[00:01:56] That'll make sense when I get to it. The first rule says that whenever hashCode is invoked on the same object more than once during an execution of the Java application, the hashCode method must consistently return the same integer.

[00:02:12] That must be true as long as no information used in equals comparisons of the object is modified.

[00:02:20] In other words, if I have an equals method and I compare some variables, then as long as I haven't changed those variables, then calling hashCode multiple times on some object must always return the same hashCode value.

[00:02:36] That tells me that it can't be a random number. It can't be a random number, it has to be always the same during any run of that program.

[00:02:43] If I run my program, it's running in the JVM, and I call hashCode, and I haven't changed anything about the variables used in the equals method, then if I call hashCode again, I must get the same number I got before.

[00:02:55] That's the first rule. The second rule says that if two objects are equal according to the equals (Object) method—in other words, if I have a reference to one of the objects, I call equals and pass the other object in as a parameter—then those two methods must produce the same hashCode.

[00:03:12] If I call it hashCode on either one, it must be exactly the same number. Those two things are rules of the hashCode method.

[00:03:20] If I violate that, then I'm going to have problems with hashing my objects and probably not be able to find them as I should, and so I never violate those two rules with hashCode.

[00:03:33] The third thing is telling me something that I might think is required that technically is not required. First of all, let me give you some background.

[00:03:41] The reason you write a hashCode method is so you can efficiently look some object up in some hashing data structure, hash table or something like that or a hashMap.

[00:03:51] That's supposed to be efficient. In other words, fast. If we didn't have number three here, this third item, we would probably think that if two methods are not equal according to the equals method, they must produce different hash codes.

[00:04:06] Now technically, we would want that. If two methods are unequal according to the equals method, we generally want the hashCodes to be different because we want them to be placed in different places in any underlying hashing data structure.

[00:04:21] If they're always the same, then it's going to be very inefficient. We want to be close to guaranteed that if two objects are not equal, their hashCode will not be equal.

[00:04:34] The problem is, if I were required to guarantee that, then often, it would be very inefficient to calculate the hashCode. It would be really slow and the whole point to having a hashCode is to be able to access an object quickly.

[00:04:52] If I have a slow method of calculating its hashCode, then I'll lose any gains that I get by using a hashing data structure.

[00:04:59] That's why this third item says that it is not required that if two objects are unequal according to the equals method that they must produce different hashCodes.

[00:05:08] That would just be inefficient to always have to guarantee that. However, we want that to be true in most cases.

[00:05:15] What we do is we make a best effort implementation that will generally have objects that are not equal and returning different hashCodes, but it's not required to be true all the time so it can be efficient.

[00:05:32] That still allows us to get the efficiency gains of using a hash table without losing it by generating the hashCode. That was a lot of words so I'm just going to summarize really quickly.

[00:05:44] The summary of these rules is that if the hashCode method is invoked on the same object more than once during any run of a program, it needs to produce the same hashCode.

[00:05:58] If two objects are equal according to the equals method, then they must produce the same hashCode result.

[00:06:05] If they are not equal, it's not required that they produce different hashCode results, but we would like them to, in most cases, in order for it to be useful, they need to produce the same hashCode in most cases.

Start slide description.

The following text appears on the screen.

The hashCode() Method

The general contract of hashCode:

- 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.

- 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 integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

End slide description.

[00:06:17] Let's understand some details about how to implement the hashCode method. The way we do this, in general, you just convert the instance variables to an integer and add them together.

[00:06:30] There's a little bit more to it than that, but that's the general answer. First of all, we need to know how to convert things to integers. For integer numbers or integral numbers, just use that number as the hash for that value.

[00:06:45] For strings, we can call the string's hashCode method because string already has a hashCode method. The string class already has one so just call hashCode on it. For objects that may be null, we can call Objects.hashCode and that will give us a reasonable hashCode for some object.

[00:07:03] For arrays, we either hash the individual elements of the array. We might do that if it's an integer array, or we can call Arrays.hashCode and that method will do it for us.

[00:07:16] Now the rest of this gets a little mathy and so most of us don't care too much about the details I'm about to share. It's useful to know why you'll see hashCode methods the way you will though.

[00:07:28] People that care a lot about math, people that have gotten PhDs in math, have determined that if we just simply convert the instance variables to numbers and add them together, we're going to get more hash collisions than we need to, and when hashing, we don't want to hash collisions because that slows down the use of a hash table.

[00:07:51] What has been determined is if we use a clever use of a prime number and use that to multiply as we're adding these numbers together, we can have fewer hash collisions.

[00:08:04] It's suggested that we use the number 31 because it's prime and it's also really close to the number 32.

[00:08:11] Computers are very efficient in doing mathematical operations on powers of 2, and so that's why 31, although it's not a power of 2, it's close to a power of 2, so that's a good number to use.

[00:08:22] If you want a lot of details, here's a reference that you can look up for a lot of details about how to calculate a hashCode method.

Start slide description.

The following text appears on the screen.

Implementing a HashCode() Method

- Hash each value in the object that you want included in the hash
    - For integral numbers, use the number as the hash for that value
    - For Strings (or other objects), call the object's hashCode() method
    - For Objects that may be null, can use the Objects.hashCode() method
    - For arrays, either hash each element in the array, or call Arrays.hashCode()

- While computing a hash, accumulate the hashed values, multiplying the accumulated value by an odd prime number (not 2) before adding the next hashed value
    - We usually multiply by 31
- Resources for learning more:
    - API documentation for hashCode() method in Object class
    - https://www.baeldung.com/java-hashcode

End slide description.

[00:08:30] If you just want to know how to do one, here's how you do one. You write your hashCode method, and you take the first variable that's been converted to an "int" and set it to some variable where you're calculating the hash.

[00:08:46] Then after that, you take that hash value, multiply it by 31, and then add that to the next variable and you do that for every variable that should be included in the hash.

[00:08:57] If you do that, you will have a reasonably unique hashCode that satisfies all the requirements from the previous slide, and it doesn't really matter that you understand the details about why we're using a prime number, but you should do that to have a good hashCode.

[00:09:13] You also...the last thing that I'll say about this is you can get IntelliJ to generate a simple hashCode method for you, and you saw me do that in a previous lecture.

[00:09:24] You can specify which variables should be included in the hashCode and that will work for a lot of cases, but it won't work for some of the assignments that we're going to have you use hashCodes for.

[00:09:34] Just know that you can generate a hashCode and that's a good idea for a lot of classes, but sometimes you have some special requirements that will require you to write one on your own.

Start slide description.

The following text appears on the screen.

hashCode() Method Example

```
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;
}
```

Source: https://www.baeldung.com/java-hashcode

End slide description.