hashCode and equals

This article is part of Marcus Biel's free Java course focusing on clean code principles. It concludes a series where we go over all the methods of the `java.lang.Object` class. Since `hashCode` and `equals` are two of `java.lang.Object`'s methods which follow a contract that binds them together, it makes sense to talk about both methods together. In fact, they are bound in such a way that we cannot implement one without the other and expect to have a well-written class. Knowing all the details about these two methods is essential to becoming a better Java programmer. So what do these two methods do?

equals method

```java
@Test
public void primitivesShouldBeEqual() {
  int i = 4;
  int j = 4;
  assertTrue(i == j);
}
```

Listing 1

The `equals` method is used to compare two objects for equality, similar to the equals operator used for primitive values. In Listing 1, we can see that `CarTest` will pass for `primitivesShouldBeEqual`, since the equality operator == inherently works for primitives. The two primitive values `i` and `j` are assigned respectively as follows, `i = 4` and `j = 4`. Asserting with the equals operator succeeds and we get a passing test, as expected.

```java
@Test
public void StringShouldBeEqual() {
  String hello1 = "Hello";
  String hello2 = "Hello";
  String hello3 = "H";
  hello3 = hello3 + "ello";
  System.out.println(hello3);
  assertTrue(hello1.equals(hello3));
}
```

Listing 2

In Listing 2, `stringShouldBeEqual` has two distinct `String` reference variables, `hello1` and `hello2`, with two distinct "Hello" `String` constants assigned to them. Before we talk about `equals`, let's look at what would happen if we tried comparing these two `Strings` with the **equality operator** which is used for primitive values. `hello1 == hello2` would return `true` without needing to use the `equals` method at all. This is because `String` values are optimized in the background for constants wherein there is actually only one of them and both variables point to the same object in memory. The part of memory where the variables `hello1` and `hello2` are stored is called a stack. On the other hand, the actual single "`Hello`" object both variables point to is stored in another part of memory called the heap.

Going back to our equality operator scenario, things become a little more complicated as soon as we add a new `String` variable where we initialize `hello3` to "H" and re-assign `hello3` to itself concatenated with "`ello`". `hello1 == hello3` would evaluate to false. Printing them both out, we would expect the `hello1 == hello3` to evaluate to true. However, `String` has the same gotcha as do other objects, i.e. we should not use the equality operator on objects, including Strings. Instead, we use the equals method as is shown on Listing 2, line 8.

String's equals method

```java
public boolean equals(Object anObject) {
  if (this == anObject) {
    return true;
  }
  if (anObject instanceof String) {
    String anotherString = (String) anObject;
    int n = value.length;
    if (n == anotherString.value.length) {
      char v1[] = value;
      char v2[] = anotherString.value;
      int i = 0;
      while (n-- != 0) {
        if (v1[i] != v2[i]) return false;
        i++;
      }
      return true;
    }
  }
  return false;
}
```

Listing 3

The String class has an equals method as seen in Listing 3. First, it checks if the equality operator works for the two Strings being compared. This is actually checking to see if both variables share the same reference, i.e. pointing to the same String in memory – making it a performance optimization.

Afterwards, it checks if the second object is actually a String. This is achieved with the **instanceof** operator. instanceof safely casts the object to a String so that we can compare both of them. An instanceof feature which many developers do not know about is that it includes a check for null. A null object will never be a String and will never evaluate to true. Lastly, the method simply loops through and compares each String's characters for equality, returning true if and *only* if they are all completely equal.

String is actually a subclass of Object and its equals method is overridden, making it work on Strings like hello1 and hello3 from Listing 1 as expected.

User classes

```java
@Test
public void porscheShouldBeEqual() {
  Car myPorsche1 = new Car("Marcus", "Porsche", "silver");
  Car myPorsche2 = new Car("Marcus", "Porsche", "silver");
  myPorsche2 = myPorsche1;
  assertTrue(myPorsche1.equals(myPorsche2));
}
```

Listing 4

Consider two instances of a custom class Car in Listing 4. It is an instance of our Car class defined in Listing 4 which happens to be a silver Porsche owned by Marcus. Both instances seem to have exactly the same attributes, so we would expect them to be the same car. Using the equals operator on them however returns false. It doesn't work and the test fails because we have two distinct instances.

Consider the simple case where myPorsche1 == myPorsche1, comparing myPorsche1 to itself. This comparison returns true as the equality operator inherently returns true whenever we compare an object to itself. Moving on to myPorsche1 == myPorsche2, it would seem that we would need to use the equals method again. In fact, running the assertion fails our test. So we try using equals by asserting myPorsche1.equals(myPorsche2) only to find ourselves greeted with a failing test result. This was working so nicely for Strings, so why is it not working with Car?

If we closely follow the execution of our code, we would discover that we eventually end up in the Object class because our Car class does not define and override its own equals method. What actually ends up happening internally is a comparison by the object with itself. In our case, myPorsche1.equals(myPorsche2) would only compare the reference variables themselves and not the actual Car objects. This comparison would invariably return false simply because we have two different reference variables. At any rate, it would be possible to make this true again if we were to assign myPorsche2 to myPorsche1. This would make it so the object initialized to myPorsche2 no longer has a variable pointing to it. This would also make both variables point to the object myPorsche1 was initialized to. Running the comparison yet again, we would then have a successful test run because we would have compared the same object with itself again. Unfortunately, this is not what we want. Instead, we want to compare both objects in a way that makes sense for our program.

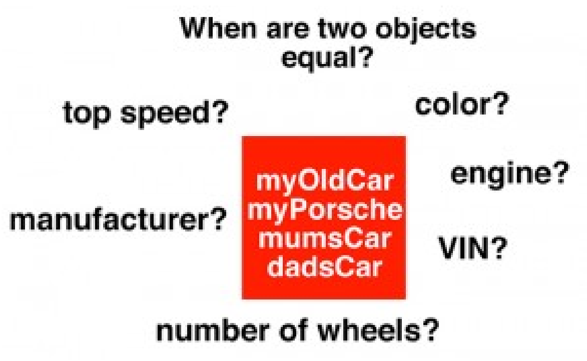
Let's have a look at the Car class.

```java
package com.marcusbiel.javacourse;

public class Car {
  private String owner;
  private String manufacturer;
  private String color;

  public Car(String owner, String manufacturer, String color) {
    this.owner = owner;
    this.manufacturer = manufacturer;
    this.color = color;
  }

  @Override
  public boolean equals(Object obj) {
    return false;
  }
}
```

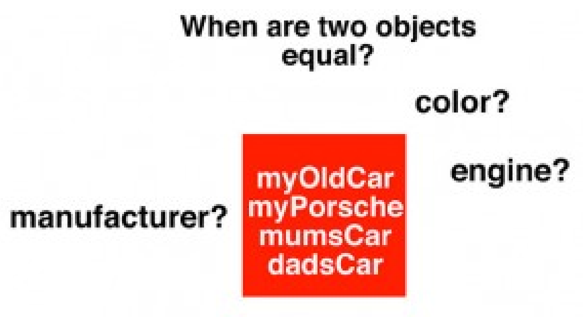
Listing 5

Listing 5 shows a small simple class with a constructor, some attributes, and an equals method. Notice that we do not have Car car as the parameter for our equals method. We cannot do this because of Object's original equals signature, so instead we want our parameter to be something like Object obj. Later on we have to cast it to Car similar to what we saw earlier with the String class. Moreover, it's always a good idea to add the @Override annotation to ensure that we have properly overridden the method. Our equals implementation is a stub that only returns false as a placeholder value though, and we will eventually have to flesh out the details. It turns out however that implementing equals is not so easy, so we will need to tackle some more theory before we continue.

Design



Most of the time, developers just skip the design of this method and simply click on "auto-generate" which in many cases leads to severe bugs or at least suboptimal performance. To implement equals correctly, however, we have to define what makes two `Car` instances equal in a way that makes sense for the context of the program it is used in. Does it make sense in its context for two cars to be equal when they have the same manufacturer? When they have the same manufacturer and color? The same engine? Or the same number of wheels? The same top speed? The same vehicle identification number (VIN)? Based on the context, we have to decide which set of fields can be used to identify an object and which fields to compare are redundant. Additionally, for improved performance, we need to define the order in which we want to compare the attributes. Are there any fields with a high chance of being unequal? Do some fields compare faster than others? Implementing a meaningful equals method requires analysis of these aspects in great detail.



In this example, we assume that the car's top speed relates to the type of the engine in some form, making it a redundant field which is not so helpful for implementing equals – so we remove it. Let's say our cars always have four wheels in our example. The number of wheels would not help us to differentiate between cars. What about the VIN? This would again depend on the type of program we want to create. For the police, this is probably the only proper way of identification. But what about a program the manufacturer uses? While the car is still being built, the VIN would probably be less important. We just cannot make broad assumptions about how we tell between cars. This is something that should be verified with the business people involved with the program. Based on what we gather from them, we select attributes from our two `Car` instances to compare and in what order. In our upcoming example, we are the business analyst and the developer as well, so we will just go ahead and define two cars to be different when they have different VINs.

The `equals` contract

First of all, there are five conditions that our `equals` implementation have to comply with. These are all actually pretty straightforward, so let's go through some examples and see what they imply:

Reflexivity

"An object must be equal to itself."

This implies that `myOldCar.equals(myOldCar)` should always return `true`. This is pretty trivial but it is still important to make sure we test for it.

Symmetry

"Two objects must agree whether or not they are equal."

If `myOldCar.equals(someOtherCar)` returns `true`, then so should `someOtherCar.equals(myOldCar)`. This also sounds pretty obvious when it fact it isn't so when it comes to inheritance. If you have a `Car` class and a `BMW` subclass of `Car`, it follows that all BMWs are cars but not every car is a BMW. This makes the rule quite tricky to follow. Make sure to cover each contract condition with a dedicated unit test to make sure the class stays fully compliant with the contract.

Transitivity

"If one object is equal to a second, and the second to a third, the first must be equal to the third."

This rule is actually more straightforward than it actually looks. For instance, let's say we have three cars: `carA`, `carB` and `carC`. If `carA == carB`, and `carB == carC`, then `carA == carC`.

Consistency

"If two objects are equal, they must remain equal for all time, unless one of them is changed."

This implies that two objects should not be altered in any way by the `equals` method. So when you repeatedly compare the same two objects with the `equals` method, it should always return the same result.

Null returns false

"All objects must be unequal to null."

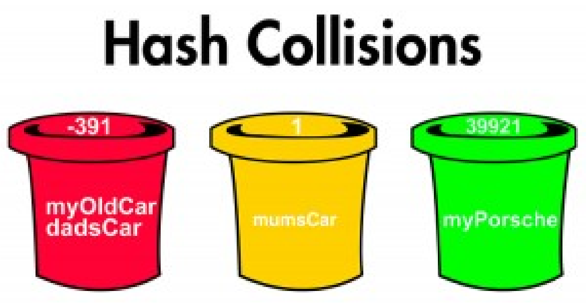
This last rule is what Josh Bloch, author of *Effective Java*, calls "Non-nullity". When given a null as an `equals` method parameter, we should always return false and never throw a `NullPointerException`.

Collections and Hashes

-9897
myPorsche
7
mumsCar
14831
dadsCar
-129
myOldCar

In Java, similar objects are usually put into a "collection" for processing. Java collections are like more powerful arrays, or "arrays on steroids"! Among other things, it allows us to look up objects not only based on position, but also by their specific values. This is where the `equals` method comes into play. To speed up the lookup, the creators of Java added specific hash-based containers which use the hash value as a grouping mechanism to reduce the number of equal comparisons needed. Ideally, objects **considered unequal** by **`equals`** will return **different hash codes** which are used to group objects in "buckets". In this ideal case, we will be able to find each object simply by a lookup based on its hash value.

However, there are “hash collisions” that come up from two unequal objects sharing the same hash code; in which case they end up in the same bucket. If we are looking for, say, the instance dadsCar, we have to look up the correct bucket based on the hash code minus 391 that dadsCar will return. With the hash collision however, we will have to do an equals comparison on a list of two cars on top of getting the hash codes.



The hashCode contract

Implementing hashCode comes with two rules. The first one being:

“For any two objects, return the same hash code when equals returns true.”

To achieve this, we use the same identifying attributes for both methods, in the same order. The second condition is:

“When hashCode is invoked more than once and on the same object, it must consistently return the same int value as long as the object is not changed.”

This rule is similar to the equals consistency rule introduced previously. Both equals and hashCode methods must return consistent results.

To fulfill this contract you must overwrite hashCode whenever you overwrite equals and vice versa. Also, when you add or remove attributes from your class you will have to adjust your equals and hashCode methods. Last but not least, aim to return different hash codes when equals returns false. Aiming to return different hash codes is not a hard and fast rule, but it will improve the performance of your program by **minimizing the number of hash collisions**. Taken to its extreme, the *hashCode* contract allows us to statically return a placeholder value such as 42 for all objects. As Josh Bloch states in his book *Effective Java* however, this could result in quadratic rather than linear execution time and therefore, could be the difference between working and not working.

The **hashCode method** is in fact a rather complicated beast. It is of such a level of complexity that should there be new, vastly superior algorithms for hashing to a 32-bit integer as is the case with Java, such a discovery would probably earn the highest honors and awards in computer science. Some of Josh Bloch’s algorithms are some of the standards we use in Java for hashing as we will see in later examples.

hashCode and equals implemented

equals

Both equals and hashCode use `manufacturer`, `engine` and `color` in their implementation. These fields are arbitrarily chosen as per the simulated design phase in our theory discussion because they make sense for our business scenarios. Consequently, we disregard using other fields like `numberOfWheels` and VIN for the two methods. Note that the order in which they are compared is also arbitrary.

```java
 1 public class Car {
 2   private Vin vehicleIdentificationNumber;
 3   private Manufacturer manufacturer;
 4   private Engine engine;
 5   private Color color;
 6   private int numberOfWheels;
 7
 8   @Override
 9   public boolean equals(Object obj) {
10     if (this == obj)
11       return true;
12     if (obj == null)
13       return false;
14     if (getClass() != obj.getClass())
15       return false;
16
17     Car other = (Car) obj;
18
19     if (!manufacturer.equals(other.manufacturer))
20       return false;
21     if (!engine.equals(other.engine))
22       return false;
23     if (!color.equals(other.color))
24       return false;
25     return true;
26   }
27
28   @Override
29   public int hashCode() {...}
```

Listing 6

For our scenario, we choose to compare manufacturers first, followed by engines, and lastly by color. If we compare two objects which are not equal, we want to leave the method as early as possible because the earlier we leave, the faster the entire code will run. In our scenario, we know that there is a plethora of manufacturers, so we can quickly return false for cars which are most likely to have different manufacturers. For our example, engines are more likely to be equal as many manufacturers use the same engines for many cars. Even though there are actually millions of colors, we make the assumption that there are only so many paint jobs and finishes that cars can come in. This way, we avoid the even higher probability of cars coming in the same color. What we end up having is a comparison order where we optimize by starting from the most likely to be unequal to the least likely, i.e. comparing manufacturers, then engines and lastly colors. Ideally, performance tests on actual sets of data will point us towards the best implementation.

Lines 10-15 in Listing 6 come at the top for performance reasons as well. Firsly, we check if the object is being compared to itself by comparing `this` to `obj`, as is shown on line 10. This can actually be removed if we know that such a scenario would never happen. That being said, it is one of the cheaper checks we can make. We then make a very important check for `null` on line 12, returning `false` when passed it and fulfilling the last part of the equals contract. Doing so helps us avoid **NullPointerException**s. Afterwards, we assert that we are comparing two Car instances. Failing this, our cast would throw a **ClassCastException**. Now that we can cast obj to Car, we can compare them by their **private** fields – which we can do because both are instances of the same class. This allows us to easily say that engine.equals(other.engine), making the code much more readable and shorter. For each pair of fields we compare, we can directly leave the method when they are unequal. Only when we go through all the fields without returning false do we get to return true. Note that we do not compare numberOfWheels as well as vehicleIdentificationNumber. Firstly, this is because we have cars which always have four wheels in our example. Secondly, we could have two physically identical cars, say both blue BMWs with the same engine, and we would still consider them to be the same car in our program.

hashCode

We have learned so far that implementing hashCode is quite complex. In order to understand the importance of the 31 constant, we need to go over a few details. 31 is a prime number, can be multiplied very quickly using shift and subtraction as follows: 31 * i == (i << 5) - i, and according to research, it provides a better key distribution which minimizes the number of hash collisions. In Listing 7, 31 is successively multiplied with the accumulated result, adding in the hash codes of the three fields we used earlier for equals. All of this optimizes our code performance by minimizing the collisions.

```java
@Override
public int hashCode() {
  int result = 1;
  result = 31 * result + manufacturer.hashCode();
  result = 31 * result + engine.hashCode();
  return 31 * result + color.hashCode();

```

Listing 7

As you can see, the method is actually not doing much other than delegating the calculation of the hash code to the other classes representing the parts of our car. Out of these three classes, let's have a detailed look into Engine to see how hashCode is implemented in further detail.

The Engine Class

```java
package com.marcusbiel.javacourse;

public class Engine {
  private long type;
  private String optionalField;

  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;

    Engine other = (Engine) obj;

    if (type != other.type) return false;
16     if (optionalField == null)
17       if (other.optionalField != null) return false;
18       else if (!optionalField.equals(other.optionalField))
19         return false;
20
21     return true;
22   }
23
24   @Override
25   public int hashCode() {
26     int result = 1;
27     result = 31 * result + (int) (type ^ (type >>> 32));
28     return 31 * result
29         + ((optionalField == null) ? 0 : optionalField.hashCode());
30   }
31 }
```
Listing 8

Engine's equals method is similar to that of Car's, a notable difference being the check for **null** on the optionalField. We have to do some slightly more complicated null checks on both objects' optionalField variables to prevent any NullPointerException. With optional primitive fields however, we do not need to check for nulls. We do not make these same assumptions on required fields such as manufacturer, engine and color, so we also do not make null checks for them. Doing so would clutter the code with "rocket code", making it extra safe but messy.

As for hashCode, we convert the long field variable type into an int as shown on Listing 8 line 27. Since longs have 64 bits and ints only have 32, we halve type somehow to minimize collisions. To understand how this is accomplished, we need to look at (int) (type ^ (type >>> 32)). First, we do a bitwise shift with the operator >>> on type over 32 bits. This moves the first 32 bits of type over to its last 32 bits. Second, we do an "exclusive or" or "xor" with the ^ operator which effectively preserves and merges some of the information from the two halves of type, ensuring that we minimize collisions when we finally downcast our long type to an int. Note that this is the current standard for hashing long fields. We do roughly the same routine as with Car's hashCode and accumulate the resulting hash.

someClass

Listing 9 shows a special example which illustrates hashCode implementation with primitive

```java
@Override
public int hashCode() {
  int result = 0;
  result = 31 * result + myByte;
  result = 31 * result + myShort;
  result = 31 * result + myInt;

  result = 31 * result
      + (int) (myLong ^ (myLong >>> 32));
  result = 31 * result
      + Float.floatToIntBits(myFloat);

  long temp = Double.doubleToLongBits(myDouble);
  result = 31 * result
      + (int) (temp ^ (temp >>> 32));

  result = 31 * result
      + (myBoolean ? 1231 : 1237);
  result = 31 * result + myChar;
  return 31 * result + myString.hashCode();
}
```
Listing 9

types in more detail. We see the same standard long field hashing being used. Floats are handled differently in that they are converted to integer bits with a native helper function before being accumulated into the hash. Double, having the same length as a long, is converted to a long with a native helper function with the long being converted back and accumulated into the integer result. Lines 17-18 look very different as they use some special syntax and two specifically chosen prime constants, namely 1231 and 1237. Line 18 uses the ternary operator "?:" which works in a similar way to an if-else statement, except that it is an expression that evaluates to a boolean. In this case, you have myBoolean as the if condition where if true would return 1231, else which returns 1237. As for 1231 and 1237, these numbers are chosen for being large primes that minimize the number of hash collisions. Lastly, char is basically a smaller integer number type, so it is trivial to accumulate it into the hash.

equals

For primitive values, we simply use the equality operator. We start with the smallest possible values because checking them has the highest chance of making the code run faster across multiple runs, i.e. we go from byte, to short, to int, to long and so on. Floating types are a little more complex to compare. float and double have to be converted to int and long respectively before we compare them. booleans are trivial to compare, and we could have placed them at the top for better performance.

Thorough business analysis can reveal even better ways to arrange these fields for performance. Again, performance tests on live data are ultimately the best ways to eke out the fastest possible runtimes when needed. Of course, all of these optimization techniques would only be useful if performance were an issue. Otherwise, the comparison order probably doesn't matter anyway.

```java
@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;

  SomeClass other = (SomeClass) obj;

  if (myByte != other.myByte)
    return false;
  if (myShort != other.myShort)
    return false;
  if (myInt != other.myInt)
    return false;
  if (myLong != other.myLong)
    return false;
  if (Float.floatToIntBits(myFloat) !=
      Float.floatToIntBits(other.myFloat))
    return false;
  if (Double.doubleToLongBits(myDouble) !=
      Double.doubleToLongBits(other.myDouble))
    return false;
  if (myBoolean != other.myBoolean)
    return false;
  if (myChar != other.myChar)
    return false;
  return myString.equals(other.myString);
}
```

Listing 10