A=B?
Sounds easy – no?

James Miller

Names, Names, Names

- Every object has a well-defined type (or class)
- And Every object has a name
- **OOP is all about manipulating names!**

Assignment

$$\begin{tikzcd}
X \arrow[r] & \text{object 1} & & X \arrow[ddr] & \text{object 1} \\
& & {} \arrow[r, Rightarrow, "X = Y"] & {} & \\
Y \arrow[r] & \text{object 2} & & Y \arrow[r] & \text{object 2}
\end{tikzcd}$$

Java has three type systems

- Primitive Types
  - These are not object-oriented types
  - Their name represents a termination point
- Object Types
  - Often called reference types
  - These are object-oriented types
  - Their name references another name
  - Which references another name
  - Which …….

The third type?

- Well somehow the recursive assignment of names needs to stop.
- Type NULL
  - Null is not a keyword, but a special literal of the null type.
  - It can be cast to any reference type, but not to any primitive type such as int or boolean.
  - The null literal doesn't necessarily have value zero.
  - And it is impossible to cast to the null type or declare a variable of this type.

The Java super class **java.lang.Object**

- Everyone produce objects; hence everyone inherits this class
- This classes has many differing definitions of equality, including

- ==
- public boolean equals(Object obj)
- public int hashCode()

Comparing Object references with ==

- This operator that can be used with object references for comparing equality (**==**).
- It compares two values to see if they **refer to the same object**. Although this comparison is very fast, it is often not what you want.
- Usually you want to know if the objects have the same *value*, and not whether two objects are a *reference* to the same object.
- For example,

  if (name == “James Miller") // Legal, but ALMOST SURELY WRONG

  -- This is true only if name is a reference to the *same object* that “James Miller" refers to.

  -- This will be false if the String in name was read from input or computed (by putting strings together or taking the substring), even though name really does have exactly those characters in it.
- Many classes (e.g., String) define the equals() method to compare the *values* of objects.

public boolean equals(Object obj)

- This method checks if some other object passed to it as an argument is *equal* to the object on which this method is invoked.
- The default implementation of this method in Object class simply checks if two object references x and y refer to the same object. i.e. It checks if x == y.
- This particular comparison is also known as "shallow comparison".
- However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison"; by actually comparing the relevant data members.
- Since class Object has no data members that define its state, it simply performs shallow comparison.

The contract of the equals method precisely states what it requires.

Reflexive

$$a \Leftrightarrow a$$

Symmetric

$$(a=b) \Leftrightarrow (b=a)$$

Hence, it is improper and incorrect to have your own class with equals method that has comparison with an object of java.lang.String class, or with any other built-in Java class for that matter. It is very important to understand this requirement properly, because it is quite likely that a naive implementation of equals method may violate this requirement which would result in undesired consequences.

Transitive

$$(a = b) \text{ and } (a = c) \Leftrightarrow (b = c)$$

Consider this example - a, b and c are three classes. a and b both implement the equals method in such a way that it provides comparison for objects of class a and class b. Now, if author of class b decides to modify its equals method such that it would also provide equality comparison with class c; he would be violating the transitivity principle. Because, no proper equals comparison mechanism would exist for class a and class c objects.

Consistent - It means that if two objects are equal, they must remain equal as long as they are not modified. Likewise, if they are not equal, they must remain non-equal as long as they are not modified. The modification may take place in any one of them or in both of them.

Null comparison - It means that any instantiable class object is not equal to null, hence the equals method must return *false* if a null is passed to it as an argument. You have to ensure that your implementation of the equals method returns false if a null is passed to it as an argument.

Equals & Hash Code relationship - The last note from the API documentation is very important, it states the relationship requirement between these two methods. It simply means that if two objects are equal, then they must have the same hash code, however the opposite is NOT true. This is discussed in details later in this article.

public int hashCode()

- **Consistency during same execution -** Firstly, it states that the hash code returned by the hashCode method must be consistently the same for multiple invocations during the same execution of the application as long as the object is not modified to affect the equals method.
- **Hash Code & Equals relationship -** The second requirement of the contract is the hashCode counterpart of the requirement specified by the equals method. It simply emphasizes the same relationship - equal objects must produce the same hash code. However, the third point elaborates that unequal objects *need not* produce distinct hash codes.

- After reviewing the general contracts of these two methods, it is clear that the relationship between these two methods can be summed up in the following statement -

  Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.

Correct Implementation Example

- The following code exemplifies how all the requirements of equals and hashCode methods should be fulfilled so that the class behaves correctly and consistently with other Java classes. This class implements the equals method in such a way that it only provides equality comparison for the objects of the same class, similar to built-in Java classes like String and other wrapper classes.

```
public class Test {
   private int num;
   private String data;

 @Override public boolean equals(Object obj) {
       if(this == obj) return true;
       if((obj == null) || (obj.getClass() != this.getClass())) return false;
              // object must be Test at this point
        Test test = (Test)obj;
         return (num == test.num) &&  (data == test.data || (data != null && 
data.equals(test.data))); }

 @Override public int hashCode()  {
       int hash = 7;
       hash = 31 * hash ^ num;
       hash = 31 * hash ^ (null == data ? 0 : data.hashCode());
       return hash;  }
   // other methods  …..
}
```

HashCode() for a String Object

$$\mathtt{h} = \mathtt{s[0]} * 31^{n-1} + \mathtt{s[1]} * 31^{n-2} + \cdots + \mathtt{s[n-1]}$$

Building our own Hashcode

- we want to combine **all data values that vary** into the hash code;
- we want the **bits of the data that vary most randomly** to affect the **lower bits** of the hash code;
- we want hash codes where as many bits as possible have roughly 50-50 chance of being a 0 or 1, especially in the lower bits;

Two ints, each fairly randomly distributed between 0 and a fairly large number.

x ^y

Two ints that have a biased distribution.

(x * p) ^ y

Where p is a prime number and/or odd number close to a power of 2, such as 17 or 33.

Two objects whose classes already have good hash functions (for example Strings or generally other standard library classes).

```
x.hashCode() ^ y.hashCode()
```

But …..

- Hashing tends to be very time-sensitive in real applications.
- So people come up with bespoke solutions based upon a detailed analysis of the concept.