Comparing Objects in Java

In this article, we'll look at how the equals() method can be used to compare objects in Java.

In this article we are going to take a look at comparing objects in Java! Comparing objects can appear mysterious at first resulting in buggy code, but once we learn a few of the rules you will be comparing objects like a pro!

Using Relational Operators to Compare Primitives

Let's start with something familiar. In Java, we can compare primitive values using relational, also called comparison, operators. Remember, in Java, primitive types include: byte, short, int, long, float, double, char and boolean.

Code Challenge

Go ahead and run the following code which uses `==`, and `!=` to compare primitives. After you see the results go ahead and try applying `==` and `!=` while comparing any other primitive types, such as the double type.

```
class ComparingPrimitives {
 public static void main(String[] args) {
  System.out.println("Comparing ints:");
  int number1 = 4;
  int number2 = 5;
  int number3 = 4;
  System.out.println(number1 == number2);
  System.out.println(number1 != number2);
  System.out.println(number1 == number3);

  System.out.println("Comparing chars:");
  char letter1 = 'a';
  char letter2 = 'b';
  char letter3 = 'a';

  System.out.println(letter1 == letter2);
  System.out.println(letter1 != letter2);
  System.out.println(letter1 == letter3);
}
}
```

The above example is fairly straightforward. If the values are the same, `==`, will return `true`. If not, `false`. The not equal operator, `!=`, will do the opposite. We can expect this behavior when using `==`, and `!=` to make comparisons among all primitives with straightforward results.

However, when we start to compare objects, the comparison behavior seems less straightforward until we learn about the rules. This is especially true when we look at the String object type.

Using Relational Operators to Compare Strings

Before we get too far into our discussion. Let me give a disclaimer: *Always use the `.equals()` method to compare Strings*. However, because this is a common pitfall Java developers run into, let's talk about what happens when we use the `==` and `!=` operators.

In Java, the String type is an object. There are some unique cases that allow us to compare String objects using `==` and `!=` that might result in writing buggy code. Let's first look at an example in which it seems like `==` and `!=` seem to work just like when we compared primitive values.

Code Challenge

Please run the following code comparing String literals:

```java
class ComparingStringLiterals {
 public static void main(String[] args) {
   String duckNoise1 = "quack";
   String duckNoise2 = "bark";
   String duckNoise3 = "quack";

   System.out.println(duckNoise1 == duckNoise2);
   System.out.println(duckNoise1 != duckNoise2);
   System.out.println(duckNoise1 == duckNoise3);
 }
}
```

At first glance, the String type, an object, is working just like our primitive example using the equality operators, `==` and `!=`. A **`"quack"`** sounds like a **`"quack"`** and a **`"quack"`** does not sound like a **`"bark"`**.

You might be asking, "Why shouldn't we use the `==`'s then?" Why did you give the disclaimer to always use **`.equals()`** method to compare Strings? Let's look at where `==` lets us down.

Let's look at what happens when a `"quack"` isn't a `"quack"`:

Code Challenge

Take a look at the following code. What do you think will be the result of the comparisons? Go ahead and run the code and take a look at the results:

```
class CompareStringObjects {
 public static void main(String[] args) {
   String duckNoise1 = new String("quack");
   String duckNoise2 = new String("quack");

   System.out.println(duckNoise1 == duckNoise2);
 }
}
```

We might have expected **`duckNoise1 == duckNoise2`** to result in **`true`** based on our previous comparison of String values it actually results in **`false`**.

Here is the reason. When working with objects, the **`==`** operator looks for location in memory before comparing values. When an object is created it is given a location in memory. **`duckNoise1`** is not an object itself. It is a reference in memory to where the object is found. **`duckNoise2`** is a reference to the location of another object in memory. The **`==`** operator sees that they are not in the same place in memory, so the value doesn't matter. They are not equal because they are first deemed to not be referring to the same object in memory so no comparison of values will take place.

Wait a second! Why did this work the first time we looked at Strings?

```
String duckNoise1 = "quack";
String duckNoise2 = "quack";
System.out.println(duckNoise1 == duckNoise2); // prints true
```

By using **`""`** to create a String instead of **`new String()`** we are creating string literals. When you declare a String like this, you are actually calling the **`inter()`** method behind the scenes on String. This method will refer to an internal pool of String objects. If there already exists a **`"quack"`** assigned to **`duckNoise1`**, then **`duckNoise2`** will reference that String, and no new String object will be created.

Remember, our disclaimer? Whenever you compare Strings, use the **`.equals()`** method. In a moment we will show why the **`.equals()`** method always checks the values of a String object without hitting the snag of whether they belong to objects that have different places in memory.

Before that, let's look at two more situations in which you may find yourself using comparison operators.

Comparison Operators with Object Aliases

Before, we look at a solution that works for both Strings created with **`""`** and the constructor method, **`new String()`**, let's look at Object aliases. In Java, an alias means that more than one reference is tied to the same object.

Code Challenge

Run the following code and take a look at what happens when we compare object aliases. Before you run the code below, take a moment and think about whether these comparisons will result in `true` or `false`.

```
class ComparingAliases {
 public static void main(String[] args) {
   String animalNoise1 = new String("meow");
   String animalNoise2 = new String("moo");
   String animalNoise3 = animalNoise2;

   // comparing different objects
   System.out.println(animalNoise1 == animalNoise2);
   // comparing object aliases
   System.out.println(animalNoise2 == animalNoise3);
 }
}
```

In the above example, our references are **`animalNoise1`**, **`animalNoise2`** and **`animalNoise3`**. By setting **`animalNoise3`** to **`animalNoise2`** we are now pointing to the same object in memory. Because of that, the `==` operator works just fine.

Comparing with Null

Another common place to use `==` or `!=` with objects is to compare then to **`null`** to see if they really exist. For instance, if we were to run the following code we would get a **`NullPointerException`** error because our String **`word`** is set to **`null`**, **`indexOf()`** throws a NullPointer error for **`word`**.

```
class Main
{
   public static void main(String[] args)
   {
     String word = null;
     if (word.indexOf("a") >= 0)
     {
         System.out.println(word + " contains an a.");
     }
   }
}
```

To avoid this error we can check if **`word != null && word.indexOf("a") >= 0`**. Because **`word != null`** is **`false`**, the rest of the boolean expression is not evaluated, which would result in a NullPointerException error.

```
class Main
{
   public static void main(String[] args)
   {
     String word = null;
     if (word != null && word.indexOf("a") >= 0)
     {
         System.out.println(word + " contains an a.");
     }
   }
}
```

Now, if we change `word` to be set to `"Alphabet"` rather than `null` you will see that we print out a message, `Alphabet contains an a`.

```
class Main
{
   public static void main(String[] args)
   {
     String word = "Alphabet";
     if (word != null && word.indexOf("a") >= 0)
     {
         System.out.println(word + " contains an a.");
     }
   }
}
```

Equals Method

Now, let's talk about the technique we can always use to compare Strings whether the Strings were created with `""`, literals, or the constructor method, `new String()`.

The obvious difference between `==` and `.equals()` is that the `==` is an operator and `.equals()` is a method. We use operators for reference comparisons (location in memory), and `.equals()` for *content comparison*.

More precisely, `==` checks if both objects point to the same memory location while `.equals()` evaluates the comparison of values in the objects.

This means we should use `.equals()` method to check whether two String objects contain the same data or not.

<u>Code Challenge</u>

Take a look at the following code. Before you run it, make a guess at what you think will be the result:

```java
class EqualsMethodComparison {
 public static void main(String[] args) {
   String duckNoise1 = new String("quack");
   String duckNoise2 = new String("quack");

   System.out.println(duckNoise1.equals(duckNoise2));
 }
}
```

Even though we are pointing to two different objects in memory, **`duckNoise1.equals(duckNoise2)`**, results in **`true`**.

`duckNoise2` is not an alias of **`duckNoise1`**. They are in different places in memory, however, **`.equals()`** allows us to compare the content rather than the location in determining equality.

Now, that we have made it this far, let's look at the **`.equals()`** method in custom classes we create. String objects have a unique **`.equals()`** method that looks for content comparison rather than location.

Let's look at what happens when we are creating our own custom classes. Will the **`.equals()`** method compare content from the instantiated objects like it did with String objects?

Default `.equals()` Method in Classes

In the code below we have a class **`Person`**. We are instantiating two **`Person`** objects. To create a person we need to pass in a **`firstname`** and **`lastname`** value. Let's submit identical values.

Code Challenge

What do you expect will happen when we run this code comparing our instantiated objects making comparisons with **`==`** and the **`.equals()`** method?

```java
class Person {
  public String firstname;
  public String lastname;
  public Person (String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  public static void main(String[] args) {
    Person person1 = new Person("Jennifer", "Smith");
    Person person2 = new Person("Jennifer", "Smith");
    // compare with `==`
    System.out.println(person1 == person2);
    // compare with `.equals()`
    System.out.println(person1.equals(person2));
  }
}
```

What happened? We probably expected `person1 == person2` to result in `false` but probably expected `person1.equals(person2)` to result in `true`!

Remember, our disclaimer was to always use `.equals()` for String objects. This is also true of comparing all objects. String objects come with a `.equals()` method already built for us that compares content rather than location. However, the classes that you create from scratch won't have that `.equals()` method by default.

By default, any class that we create will inherit its `.equals()` method from the Object class. In the Object class the `.equals()` method works just like the `==` comparison operator. It looks for location in memory before looking at content. So in our example above we fell into the pitfall of comparing the objects' locations in memory.

Override the `.equals()` method

The solution is to override the `.equals()` method in any class you create. In fact, whenever you create a new class where you know you'll be comparing objects for equality, you'll want to override the `.equals()` method. In doing so, you get to chose for yourself what it means for two objects to be equal! (As a side note, terminology like "override", and "inherit", may be new to you. We cover inheritance in more detail towards the end of the AP Computer Science path).

So let's override the `.equals()` method we inherit with our own custom `.equals()` method. This method will compare the values of the instantiated objects `firstname`'s and `lastnames`'s values.

Code Challenge

Take a look at the `Person` class and the custom `equals()` method that determines if two objects are equal if their `firstname` and `lastname` attributes have the same value:

```
class Person {
  public String firstname;
  public String lastname;
  public Person (String nickname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
  // custom `equals()` method
  public boolean equals(Person p) {
    return (p.firstname == firstname && p.lastname == lastname);
```

```java
  }
  public static void main(String[] args) {
    Person person1 = new Person("Jennifer", "Smith");
    Person person2 = new Person("Jennifer", "Smith");
    // compare with `==`
    System.out.println(person1 == person2);
    // compare with `.equals()`
    System.out.println(person1.equals(person2));
  }
}
```

Let's try a little challenge that involves writing your own custom `.equals()` method!

Imagine that you are in charge of payroll and collect timecards from employees daily, including their `firstname`, `lastname`, `date`, and employee `id`.

You are looking for a timecard for a specific employee on a specific date. You want to know if an employee, "Jennifer Smith", worked on November 3rd, 2020. We have instantiated an object `timecardSearch` that you will be using to compare to the rest of the objects. We have instantiated an object from the `Timecard` class called `timecardSearch` that we will use to compare to the other objects instantiated from the `Timecard` class.

To do this, you are going to have to write your own `.equals()` method. What values should we check?

We need to write an `.equals()` method to compare our values on properties that would find if "Jennifer Smith" worked on November 3rd, 2020.

We could do this by checking the value of the `firstname` property. The only problem is that "Jennifer" might sometimes sign in as "Jenny", "Jennie", "Jen", etc. So let's not use that in our custom `.equals()` method.

We could search by the `lastname` property but "Smith" is a fairly common name and might pull the timesheets of multiple employees. Even if we had a strict requirement of employees using their legal name you could still have matches as matching first and last names are not a guarantee of uniqueness.

What is unique to all employees is their employee `id`. In this case "Jennifer Smith" has a unique `id` of `jSmith`. Perfect, let's use that in our custom `.equals()` method!

We also want to check the date of the time card. We want to `date` property to be set to a value of `11032020` for November 3rd, 2020.

Code Challenge

Please run the following code. Because class `Timecard` is using the `.equals()` method inherited from the Object class it will work just like the `==` comparison operator. We can expect all of our comparisons will return `false`. Then follow the narrative after the coding challenge for direction on how to write your own custom `equals()` method if you need help!

```java
class Timecard {
  public String firstname;
  public String lastname;
  public int date;
  public String id;
  public Timecard (String firstname, String lastname, int date, String id) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.date = date;
    this.id = id;
  }

  // public boolean equals(Timecard t) {
    // uncomment method and write code here
  // }

  public static void main(String[] args) {
  // declare and instantiate
  Timecard timecardSearch = new Timecard("Jennifer", "Smith", 11032020, "jSmith");
    Timecard timecard1 = new Timecard("Jennie", "Jones", 1102020, "jJones");
    Timecard timecard2 = new Timecard("Jennifer", "Tompkins", 11022020, "jTompkins");
    Timecard timecard3 = new Timecard("Jen", "Smith", 11032020, "jSmith");
    Timecard timecard4 = new Timecard("Jenny", "Lawrence", 11032020, "jLawrence");
    Timecard timecard5 = new Timecard("Jenn", "Williams", 11042020, "jWilliams");

  // comparisons
  System.out.println(timecardSearch.equals(timecard1));
  System.out.println(timecardSearch.equals(timecard2));
  System.out.println(timecardSearch.equals(timecard3));
  System.out.println(timecardSearch.equals(timecard4));
  System.out.println(timecardSearch.equals(timecard5));
  }
}
```

We should be able to look at our objects and see that `timecardSearch` and `timecard3` are equal. They both have the same values for `id` and `date`.

Let’s go ahead and write a custom **`.equals()`** method above that will only return **`true`** if both a primitive type, the **`date`**, and object type, an **`id`**, are a match. This means that we will write our custom **`.equals()`** method using `==` for comparing the date and **`.equals()`** to compare **`id`**‘s.

Go ahead and run the code above to make sure that your method works!

Conclusion

Great job! The object that matched our **`timecardSearch`** for both **`id`** and **`date`** properties was the **`timecard3`** object.

We learned a few things about making comparisons in Java.

- The equality operator works great in comparing primitives and letting us know if objects are aliases of each other.
- We comparing objects, operators make reference comparisons (location in memory), and **`.equals()`** will make content comparison.
- When comparing String values it is best to use the **`.equals()`** method to avoid buggy unexpected behavior.
- When comparing property values of objects we create from custom classes you will need to override the default behavior of the **`.equals()`** method we inherit from the Object class that works like the `==` operator.