A Drip of JavaScript

Equals Equals Null in JavaScript

Originally published in the A Drip of JavaScript newsletter.

One of the strongest injunctions that new JavaScript developers receive is to always use strict equality (`===`) in comparisons. Douglas Crockford recommends this approach in JavaScript: The Good Parts, and it is considered by some parts of the JavaScript community to be a best practice.

But even in code that follows this practice, you're likely to run across one exception: `== null` . Or possibly `!= null` . Here is an example from jQuery's source.

```
if ( val == null ) {
    val = "";
}
```

What does this code do? And why is it a common exception to the triple equals rule? Let's take a look at an example where it might be useful.

```
var ethos = {
    achilles: "glory",
    aeneas: "duty",
    hades: null // Beyond human understanding
};

function printEthos (name) {
    console.log(ethos[name]);
}
```

```
// Outputs: "glory"
printEthos("achilles");

// Outputs: "null"
printEthos("hades");

// Outputs: "undefined"
printEthos("thor");
```

The `printEthos` function will be called in response to a user inputting a name. And while this works fine for `"achilles"` and `"aeneas"`, it doesn't really work for `"hades"` or `"thor"`. The problem here is that they give us two different types of nothing, and neither of them means anything to the end user. It would be much better to capture any "nothings" and display a user-friendly message. So let's do that.

```
function printEthos (name) {
    if (ethos[name]) {
        console.log(ethos[name]);
    } else {
        console.log(name + " has no recorded ethos.");
    }
}

// Outputs: "hades has no recorded ethos."
printEthos("hades");

// Outputs: "thor has no recorded ethos."
printEthos("thor");
```

We've solved our problem. Or have we? By testing the truthiness of `ethos[name]`, we've opened ourselves up to a different sort of issue. Consider what happens in the case of Pythagoras.

```
ethos.pythagoras = 0; // The sublimity of Number
```

```
// Outputs: "pythagoras has no recorded ethos"
printEthos("pythagoras");
```

Because `0` is a falsy value, `"pythagoras"` isn't recognized as having an ethos. So we need something that won't accidentally catch falsy values when we really just want to catch nothingness as opposed to somethingness. And that's what `== null` does.

```
function printEthos (name) {
    if (ethos[name] != null) {
        console.log(ethos[name]);
    } else {
        console.log(name + " has no recorded ethos.");
    }
}

// Outputs: "hades has no recorded ethos."
printEthos("hades");

// Outputs: "thor has no recorded ethos."
printEthos("thor");

// Outputs: 0
printEthos("pythagoras");
```

Despite the fact that `null` is a falsy value (i.e. it evaluates to `false` if coerced to a boolean), it isn't considered loosely equal to any of the other falsy values in JavaScript. In fact, the only values that `null` is loosely equal to are `undefined` and itself.

Because of this interesting property, it has become somewhat conventional to use `== null` as a more concise way of checking whether a given value is "nothing" (`null` / `undefined`) or "something" (anything else). In addition to jQuery, you can find examples of this convention in Underscore.and Less. Indeed, JSHint, one of the more popular JavaScript linting tools, provides an option to allow the use of loose equality only when comparing to null.

Of course, you can choose to be more explicit about the fact that you are checking for both `null` and `undefined`, but next time you run across `== null`, don't automatically think, "Bad practice!" Instead, take a look at the context and see if it is being used strategically.

Thanks for reading!

Josh Clanton
