JavaScript Null & Undefined

25 OCTOBER 2015 on javascript, types

Today will be a shorter post to touch on the types `null` and `undefined`. These are sometimes referred to as `non-values`. `undefined` and `null` are both the type and value. They are to be treated as empty or 'non' values. Specifically, `null` is a JavaScript literal representing null or an "empty" value, whereas `undefined` is a global property that represents the primitive value `undefined`.

These types are sometimes interchangeable, so some developers prefer specificity in distinguishing them:

- `undefined` hasn't had a value yet
- `null` had a value and doesn't anymore

OR

- `null` is an empty value
- `undefined` is a missing value

Go with whatever works for your way of thinking.

null

Let's focus on **null** real quick. There's really not much to it. It's basically a placeholder for something with no value, but there's a major caveat to `null`, and that is `typeof null` returns `'object'`. This is a bug that has been around for so long that fixing it would cause too much code to break. So, if we want to check if something is `null` we have to do the following:

```
var nothing = null
(!nothing && typeof nothing === 'object') // true
```

JavaScript evaluates anything in parentheses `()`. In the above snippet we evaluate that `nothing` has no value, and it is of the type `object`. If both these things are true, this expression will evaluate to `true`, meaning our variable is `null`.

---

undefined

Variables that have no value are **undefined**. Variables that are declared, but not assigned a value, are given the value and type `undefined`. We can see calling `typeof` on such variables will return `'undefined'`.

```
var a
typeof a // 'undefined'
```

Similarly to `undefined` we might see `undeclared`, most likely in the context of a `ReferenceError`. If something goes wrong in

your program, it will display an error. Errors are a large subject, and so we will just peek at `ReferenceError`.

```
var a
console.log(a) // undefined
console.log(b) // ReferenceError: b is not defined
```

The language is not the most clear here, but we can see in the above example that `a` logged `undefined` because we had declared it, but left no assignment. Then we attempted to log the variable `b`, and we got `ReferenceError: b is not defined`. This is telling us that the program attempted to do something with variable `b`, but it was not defined (undeclared), and it threw an error. The term `throw` is often used around errors, and basically just means log some details about what went wrong, so we can hopefully use it for debugging and to fix the issue. If you find yourself having good reason to avoid this error from throwing, here is a solution:

```
// Safety check to avoid a ReferenceError for
undeclared.


// will error
if (DEBUGGER) console.log('Debugging started.')
// safe existence check
if (typeof DEBUGGER !== 'undefined')
console.log('Debugging started.')
// feature check for API
if (typeof yourFeature === 'undefined') yourFeature =
function(){/*...*/}
// or check the global object. window is the global in
browsers.
```

```
if (!window.yourFeature) yourFeature = function()
{/*...*/}
```

Here's a little bonus for the day. Don't sweat this one. It's not very common, but neat to know about. JavaScript has a **void** operator. It evaluates the given expression and then returns `undefined`. `void` will void out any value, so the result is always `undefined`. Note, this does not affect the original value.

```
var num = 999
console.log(void num) // undefined
console.log(num) // 999
```

Just remember that `null` and `undefined` represent 'non' or empty values, and you'll be OK. We'll be at you with Objects and Numbers this week. Thanks for joining, and see you then!

Seth Shober

Looking for work :)

http://sethshober.com/