Gmail

Dhaval Chheda <kiddo.dhaval@googlemail.com>

[Just JavaScript] 02. The JavaScript Universe
1 message

Dan Abramov <dan@overreacted.io> 28 May 2020 at 13:21
To: Dhaval <kiddo.dhaval@gmail.com>

In the beginning was the Value.

What *is* a value? It's hard to say.

This is like asking what a number is in math, or what a point is in geometry. **A value is a *thing* in the JavaScript universe.**

Numbers are values — but so are a few other things, like objects and functions. However, many things, such as an `if` statement or a variable declaration, are *not* values.

Code and Values

To distinguish values from everything else in my JavaScript program, I like to imagine this drawing of the Little Prince by Antoine de Saint-Exupéry:

I'm standing on a small asteroid — it is the code of my program

I'm standing on a small asteroid — it is the code of my program.

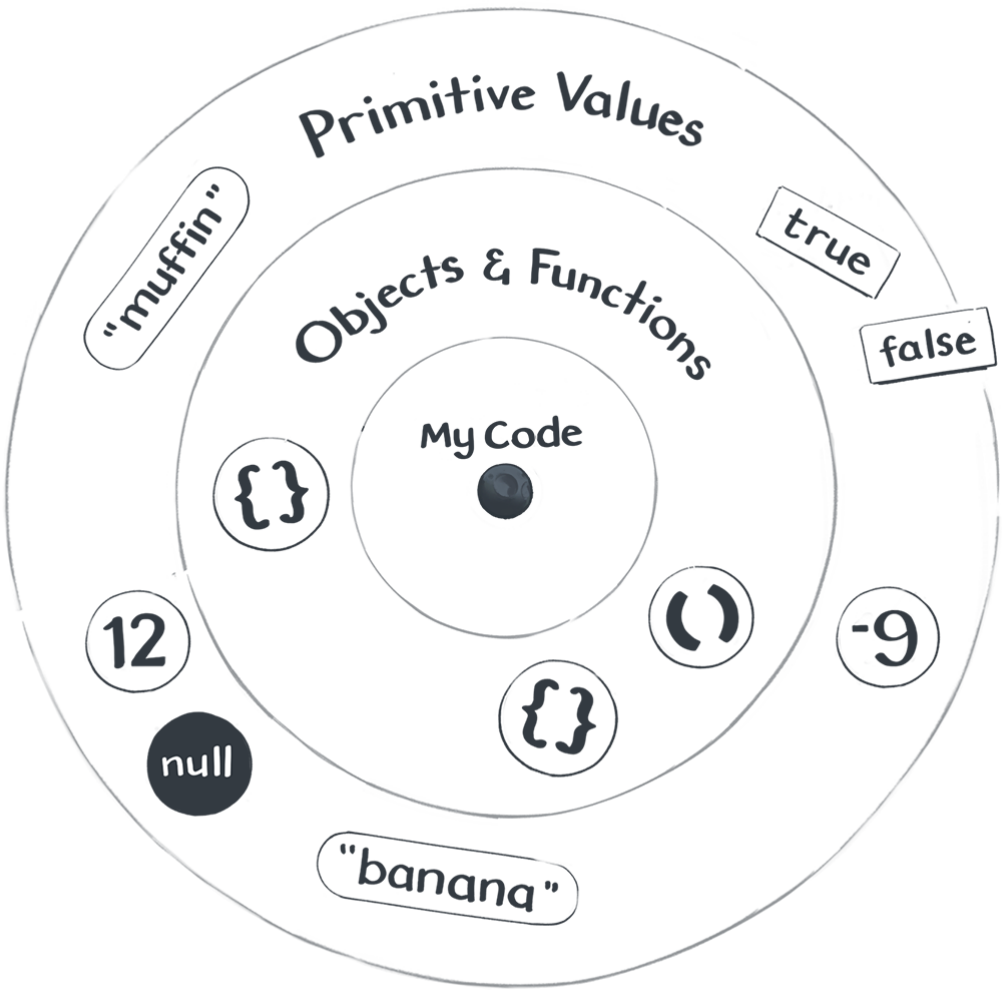
On its surface, I see the `if` statements and variable declarations, commas, curly braces, and all the other things one might find in the JavaScript code.

My code contains instructions like “make a function call” or “do this thing many times”, or even “throw an error”. I walk through these instructions step by step — running errands from my small asteroid.

But every once in a while, I look up.

On a clear night, I see the different values in the JavaScript sky: booleans, numbers, strings, symbols, functions and objects, `null` and `undefined` — oh my! I might refer to them in my code, but they don’t exist *inside* my code.

In my JavaScript universe, values float in space.



"Hold on," you might say, "I always thought of values as being *inside* of my code!" Here, I'm asking you to take a leap of faith. It will take a few more modules for this mental model to pay off. Give it five minutes.

Back to values. Broadly, there are two kinds of them.

Primitive Values

Primitive Values are numbers and strings, among other things. Open your browser's console and print these primitive values using `console.log()`:

```
console.log(2);
console.log("hello");
console.log(undefined);
```

All primitive values have something in common. **There's nothing I can do in my code that would affect them.** This sounds a bit vague, so we'll explore what this means concretely in the next module. For now, I'll say that primitive values are like stars — cold and distant, but always there when I need them.

That's the first kind of values.

Objects and Functions

Objects and Functions are also values, but they are not primitive. This makes them very special. Go ahead and log a few of them to the browser console:

```
console.log({});
console.log([]);
console.log(x => x * 2);
```

Notice how the browser console displays them differently from the primitive values. Some browsers might display an arrow before them, or do something special when you click them. If you have a few different browsers installed

special when you click them. If you have a few different browsers installed (e.g. Chrome and Firefox), compare how they visualize objects and functions.

Objects and functions are special because **I can manipulate them from my code**. For example, I can connect them to other values. This is rather vague — so we'll refine this idea in a later module. For now, I can say that if primitive values are like distant stars, then objects and functions are more like rocks floating nearby my code. They're close enough that I can manipulate them.

And that's the second kind of values.

You might have questions. Good. If you ask a question, the JavaScript universe might answer it! Provided, of course, that you know how to ask.

Expressions

There are many questions JavaScript can't answer. If you want to know whether it's better to confess your true feelings to your best friend or to keep waiting until you both turn into skeletons, JavaScript won't be of much help.

But there are some questions that JavaScript would be *delighted* to answer. These questions have a special name — they are called *expressions*.

If we "ask" the expression `2 + 2`, JavaScript will "answer" with the value `4`.

```
console.log(2 + 2); // 4
```

Expressions are questions that JavaScript can answer. JavaScript answers expressions in the only way it knows how — with values.

If the word “expression” confuses you, think of it as a piece of code that *expresses* a value. You might hear people say that `2 + 2` “results in” or “evaluates to” `4`. These are all different ways to say the same thing.

We ask JavaScript `2 + 2`, and it answers with `4`. **Expressions always result in a single value.** Now we know enough about expressions to be dangerous!

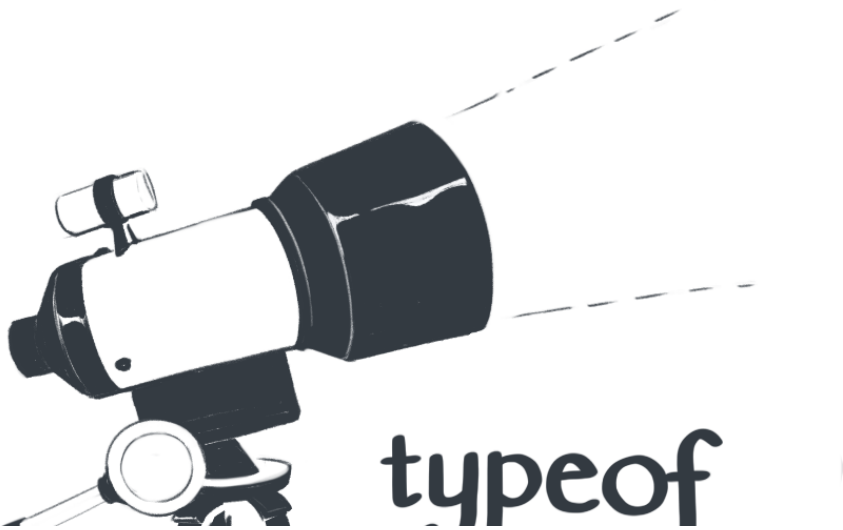
I previously said that there are many types of JavaScript values: numbers, strings, objects, and so on. How do we know any particular value’s type?

This sounds like a question. Do we dare to ask it?

Checking a Type

At first, all values in the JavaScript cosmos might look the same — bright dots in the sky. But if you look closely, you’ll realize there are fewer than ten different types of values. Values of the same type behave in similar ways.

If we want to check a value’s type, we can ask it with the `typeof` operator. JavaScript will answer our question with one of the predetermined string values, such as `"number"`, `"string"`, or `"object"`.



Below are a few examples you can try in the browser console:

```
console.log(typeof(2)); // "number"
console.log(typeof("hello")); // "string"
console.log(typeof(undefined)); // "undefined"
```

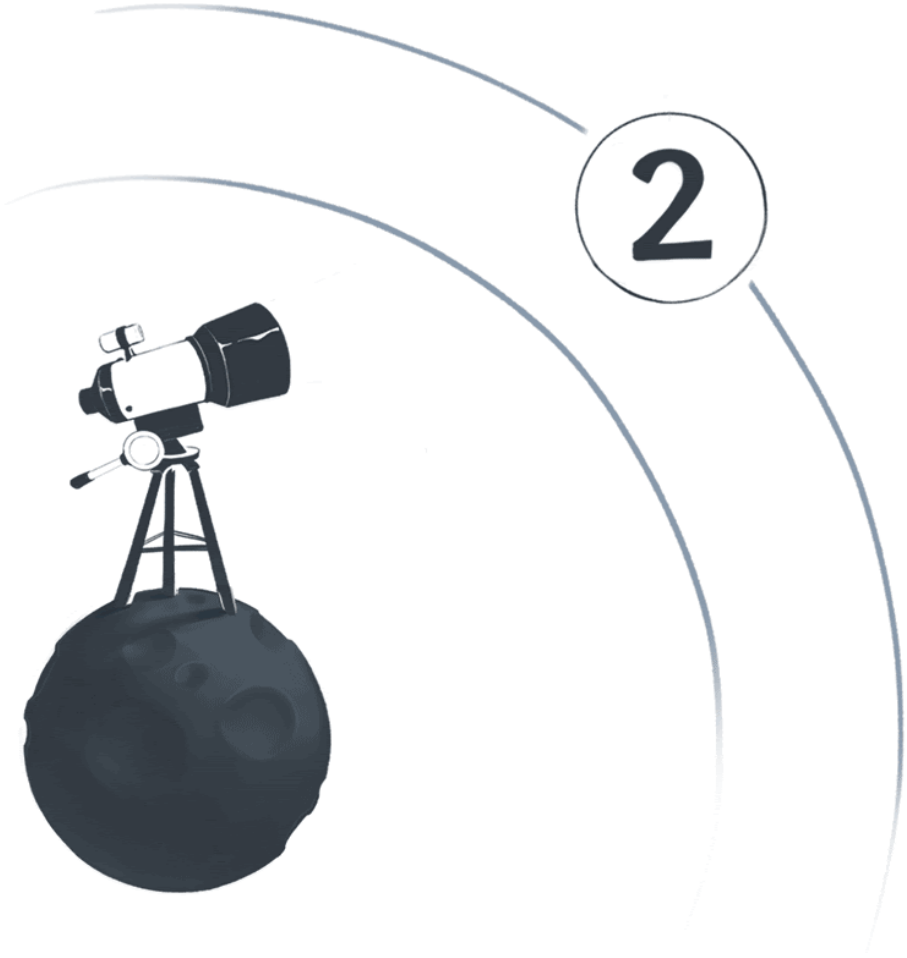
Here, `typeof(2)` is an expression — and it results in the `"number"` value.

Strictly saying, using parens isn't required with `typeof`. For example, `typeof 2` would work just as fine as `typeof(2)`. However, sometimes parens are required to avoid an ambiguity. One of the cases below would break if we omitted the parens after `typeof`. Try to guess which one it is:

```
console.log(typeof({})); // "object"
```

```
console.log(typeof({})); // "object"
console.log(typeof([])); // "object"
console.log(typeof(x => x * 2)); // "function"
```

You can verify your guess in the browser console.



Now take another look at the last three examples — this time with close attention to their results. Did you find any of these results surprising? Why?

Types of Values

As an aspiring astronomer, you might want to know about *every* type of value that can be observed in the JavaScript sky. After almost twenty five years of studying JavaScript, the scientists have only discovered nine such types:

Primitive Values

- **Undefined** (`undefined`), used for unintentionally missing values.
- **Null** (`null`), used for intentionally missing values.
- **Booleans** (`true` and `false`), used for logical operations.
- **Numbers** (`-100`, `3.14`, and others), used for math calculations.
- **Strings** (`"hello"`, `"abracadabra"`, and others), used for text.
- **Symbols** (uncommon), used to hide implementation details.
- **BigInts** (uncommon and new), used for math on big numbers.

Objects and Functions

- **Objects** (`{}` and others), used to group related data and code.
- **Functions** (`x => x * 2` and others), used to refer to code.

No Other Types

You might ask: “But what about other types I have used, like arrays?”

In JavaScript, there are no other fundamental value types other than the ones we have just enumerated. The rest are all objects! For example, even arrays, dates, and regular expressions fundamentally *are* objects in JavaScript:

```
console.log(typeof([])); // "object"
console.log(typeof(new Date())); // "object"
console.log(typeof(/(hello|goodbye)/)); // "object"
```

“I see,” you might reply, “this is because *everything* is an object!” Alas, this is a popular urban legend, but it’s not true. Although code like `"hi".toUpperCase()` makes `"hi"` seem like an object, this is nothing but

an illusion. JavaScript creates a wrapper object when you do this, and then immediately discards it.

It's fine if this mechanism doesn't quite click yet. **For now, you only need to remember that primitive values, such as numbers and strings, are *not* objects.**

Recap

Let's recap what we know so far:

1. **There are values, and then there's everything else.** We can think of values as different things "floating" in our JavaScript universe. They don't exist *inside* our code, but we can refer to them from our code.
2. **There are two categories of values: there are *Primitive Values*, and then there are *Objects and Functions*.** In total, there are nine separate types. Each type serves a specific purpose, but some are rarely used.
3. **Some values are lonely.** For example, `null` is the only value of the Null type, and `undefined` is the only value of the Undefined type. As we will learn later, these two lonely values are quite the troublemakers!
4. **We can ask questions with expressions.** JavaScript will answer to us with values. For example, the `2 + 2` expression is answered with `4`.
5. **We can inspect the type of something by wrapping it in a `typeof` expression.** For example, `typeof(4)` is the string value `"number"`.

Exercises

Now it's time to put what we learned to action.

Even if you already have a decent amount of experience with JavaScript don't skip the exercise questions! I personally learned some of these things only a few years ago.

only a few years ago.

Click here to answer these questions and provide feedback about this module. *When you complete the exercises I will send the next module right away.*

Next up we will explore the Primitive Values in more detail. We look at what these different primitive types like numbers and Null have in common, and learn a thing or two about what equality means in JavaScript.

We will also continue to refine our mental model. This module presents a crude sketch — an approximation. We will focus on different parts of the picture and fill them in with more details, like a progressive JPEG image.

These might seem like small steps, but we're laying the foundation for everything else to come. We're building the JavaScript universe, together.