Gmail

Dhaval Chheda <kiddo.dhaval@googlemail.com>

[Just JavaScript] 04. Counting the Values (Part 1)
1 message

Dan Abramov <dan@overreacted.io> 29 May 2020 at 10:19
To: Dhaval <kiddo.dhaval@gmail.com>

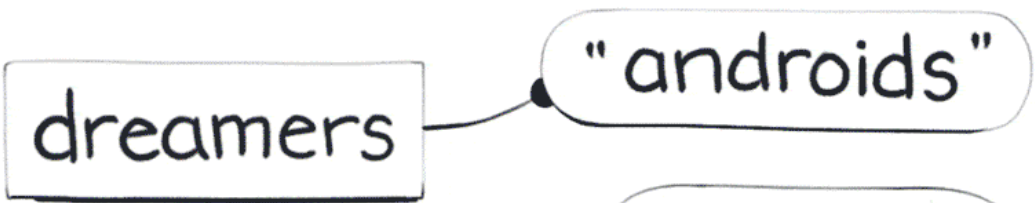
In this module, we'll take a closer look at the JavaScript world and the values in it. But before we can get to that, we need to address the elephant in the room. *Is the JavaScript world even real?*

The JavaScript Simulation

I live on my small asteroid inside the JavaScript universe.

When I ask the JavaScript world a question, it answers me with a value. I certainly don't come up with all these values by myself. The variables, the wires, the values — they all populate my world. The JavaScript world around me is absolutely real to me — just as the world you live in is real to you.

But sometimes, there is a moment of silence before the next line of code. An idle tick before the next function call. A glitch in the Matrix. During those moments, I see visions of a world that's much bigger than mine.



In the world that appears to me, there are no variables and values. No expressions and no literals. Instead, there are quarks, and atoms, and electrons, and water, and life. Perhaps, you are familiar with this world?

There, sentient beings called “humans” use special machines called “computers” to simulate *my* JavaScript universe. Some of them do it for amusement. Some of them do it for profit. Some of them do it for no reason at all. At their whim, my whole world gets born and dies a trillion times a day.

Maybe my JavaScript world isn’t so real, after all.

This means there are two ways to study it.

Studying From the Outside

One way to study my JavaScript world would be to *study it from the outside*.

Perhaps, you might focus on how a simulation of my world — a JavaScript engine — “really” works. For example, you might learn that this string of text — *a value in my world* — is a sequence of bytes stored inside a silicon chip.

This approach puts our mental focus on the physical world of people and computers. Some tutorials take this approach. But my approach is different.

Studying From the Inside

We will be studying the JavaScript world *from the inside*. Transport yourself mentally into the JavaScript universe and stand next to me. We will observe its laws and perform experiments, like physicists do in the physical universe.

We will learn about the JavaScript world for what it is — without thinking about how it's implemented. This is similar to how physicists can talk about properties of stars without answering the question whether the *physical* world is real. It doesn't matter! We can still describe it on its own terms.

Our mental model does not attempt to answer questions like "How is a value represented in the computer memory?" The answer changes all the time! In fact, the answer to this question changes even while your program is running. If you heard of a simple explanation about how JavaScript "really" represents numbers, strings, or objects in memory, it is most likely wrong.

To me, each string is a value. Not a "pointer" or a "memory address" — but a *value*. **In my universe, a value is good enough.** Don't allow "memory cells" and other low-level metaphors to distract you from building an accurate high-level mental model of JavaScript. It's turtles all the way down anyway!

If you're coming from a lower-level language, set aside your intuitions about "passing by reference", "allocating on stack", "copying on write", and so on. These models of how a computer works often make it *harder* to be confident what can or cannot happen in a JavaScript program. We'll look at some of the lower level details, but only where it really matters. They can serve as an *addition* to our mental model, rather than its foundation.

Instead, the foundation of our mental model is that our world is full of values. Each value belongs to one of a few built-in types. Some of them are primitive, which makes values of those types immutable. Variables are "wires" pointing from names in our code to values. And we'll keep building on that foundation.

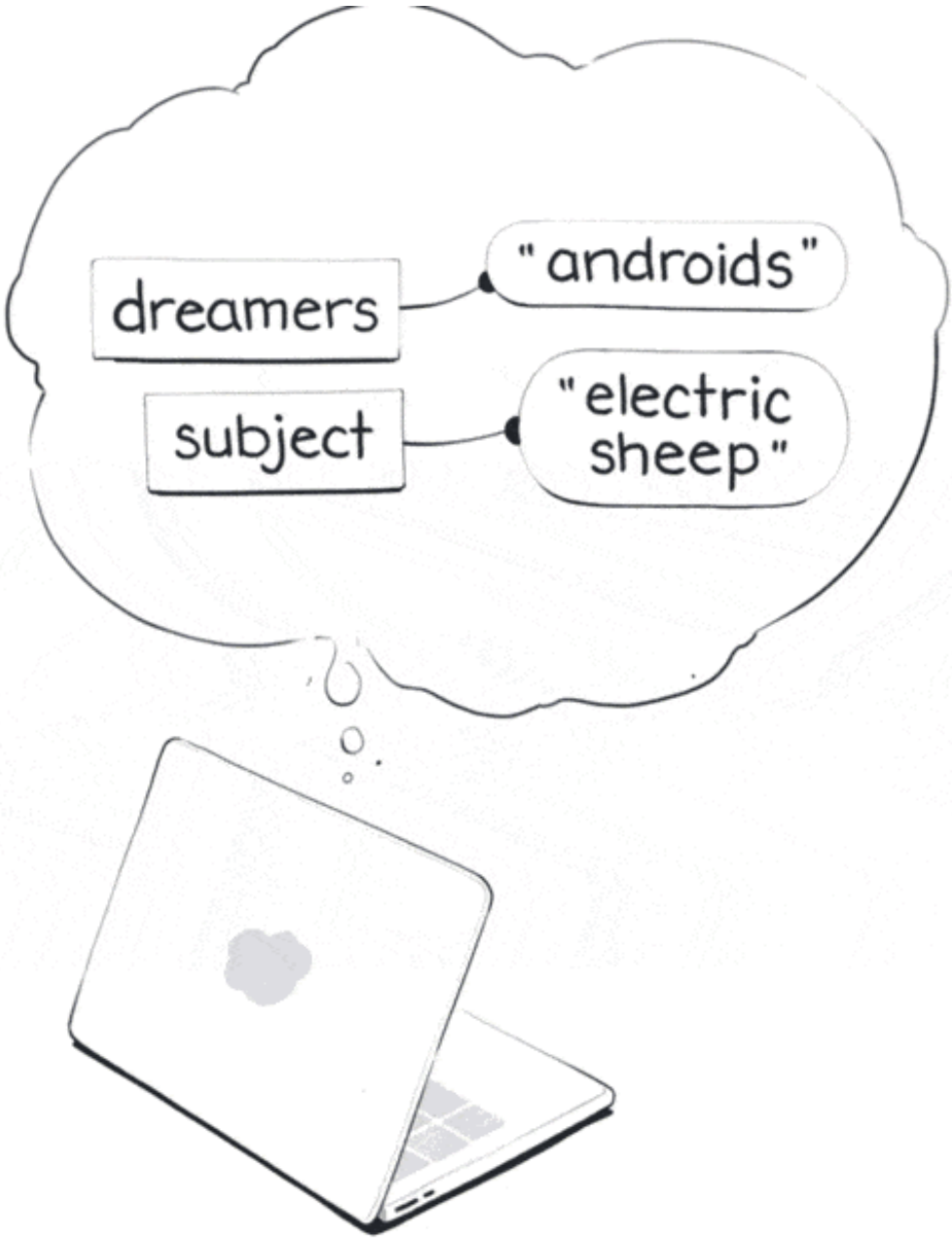
As for these strange visions, I don't pay as much thought to them anymore. I have wires to point, questions to ask, and functions to call. I better get to it!

The stars are bright when I look at them.

Are they still there when I blink?

I shrug.

"Implementation details."



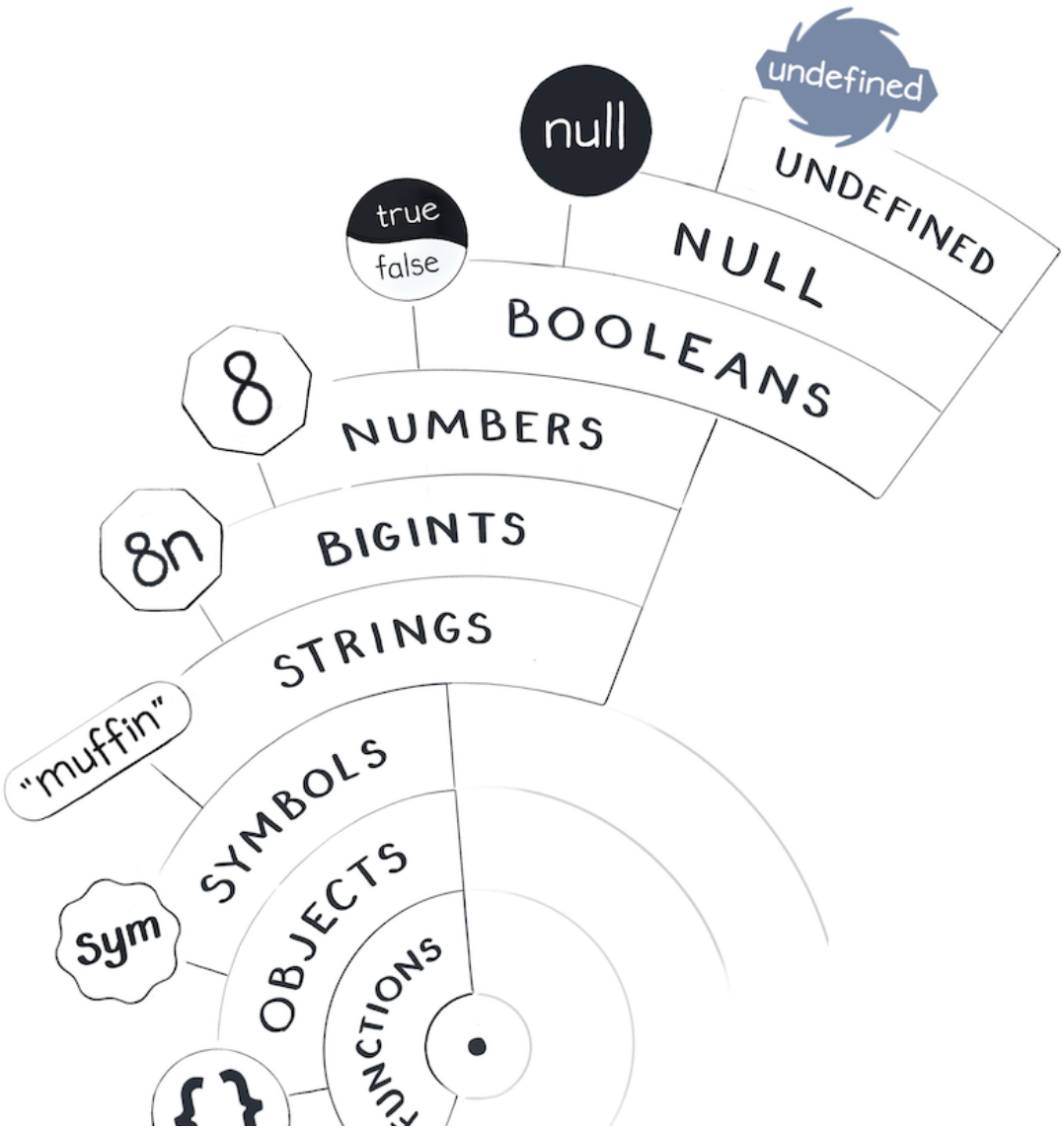
Counting the Values

Counting the Values

Count von Count was my childhood role model. If you're not familiar with him from the Sesame Street, his favorite pastime is counting things. Today, Count von Count will join us in counting every value in the JavaScript universe.

You might wonder: what's the point of *counting* values? We're not in an arithmetics class, are we? The essence of counting is to distinguish things from one another. You can only say there are "two apples" when you clearly see that they're two *distinct* apples. Distinguishing values from one another is key to understanding *equality* in JavaScript — which will be our next topic.

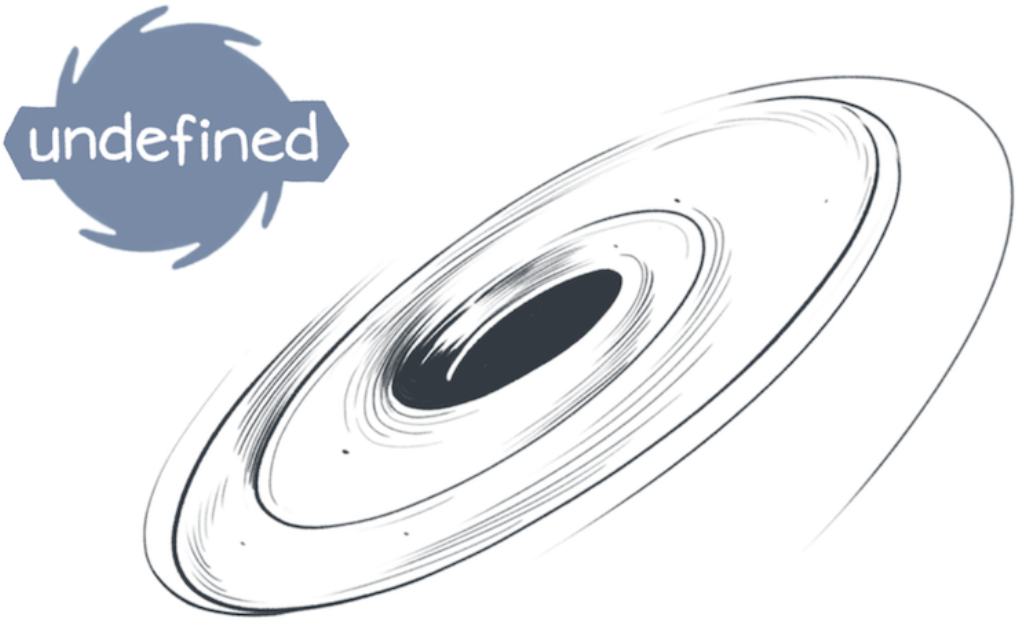
Like Virgil guided Dante through the nine circles of Hell, Count von Count will accompany us through the "celestial spheres" of JavaScript to meet different values: Booleans, Numbers, Strings, and so on. Consider it a sightseeing tour.



Undefined

We'll start our tour with the Undefined type. Count von Count will be pleased to know that **there is only one value of that type — `undefined`**.

```
console.log(typeof(undefined)); // "undefined"
```



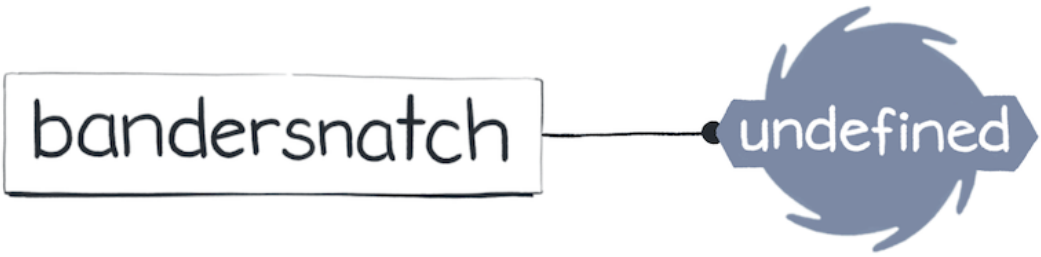
It's called undefined so you might think it's not there — but it *is* a value, and a very real one! Like a black hole, `undefined` is grumpy and can often spell trouble. For example, reading a property from it will break your program:

```
let person = undefined;
console.log(person.mood); // TypeError!
```

Oh, well. Luckily, there is only one `undefined` in the entire JavaScript universe. You might wonder: why does it exist at all? In JavaScript, it represents the concept of an *unintentionally* missing value.

You could use it in your own code by writing `undefined` — like you write `2` or `"hello"`. However, `undefined` *also* commonly "occurs naturally". It shows up in some situations where JavaScript doesn't know what value you wanted. For example, if you forget to assign a variable, it will point to `undefined`:

```
let bandersnatch;
console.log(bandersnatch); // undefined
```



Then you can point it to another value, or to `undefined` again if you want.

Don't get too hung up on its name. It's tempting to think of `undefined` as some kind of variable status, e.g. "this variable is not yet defined". But that's a completely misleading way to think about it! In fact, if you read a variable that was *actually* not defined (or before the `let` declaration), you will get an error:

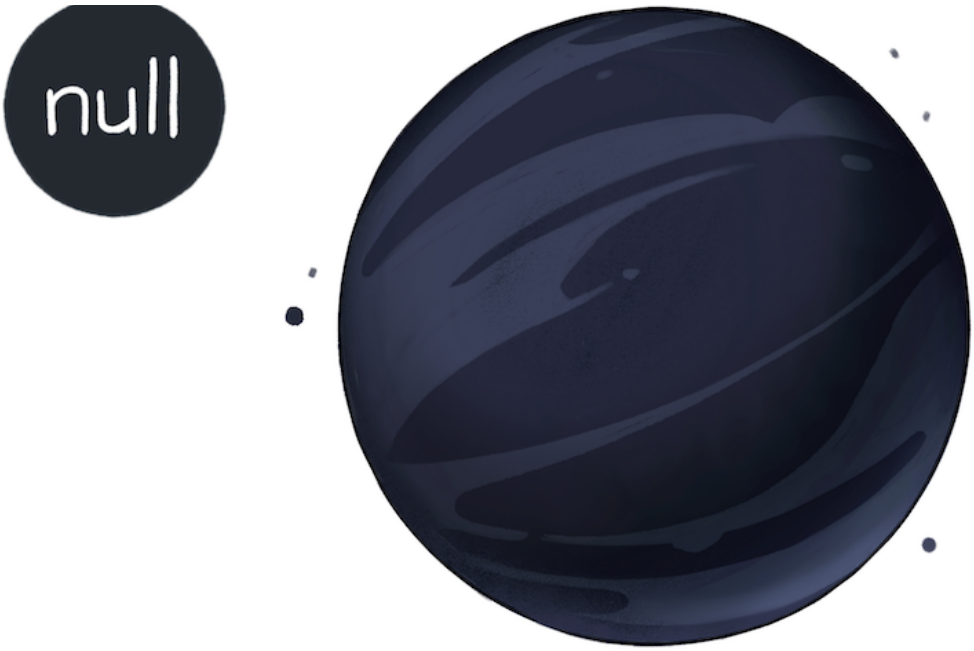
```
console.log(jabberwocky); // ReferenceError!
let jabberwocky;
```

That has nothing to do with undefined.

Really, `undefined` is a regular primitive value, like `2` or `"hello"`.

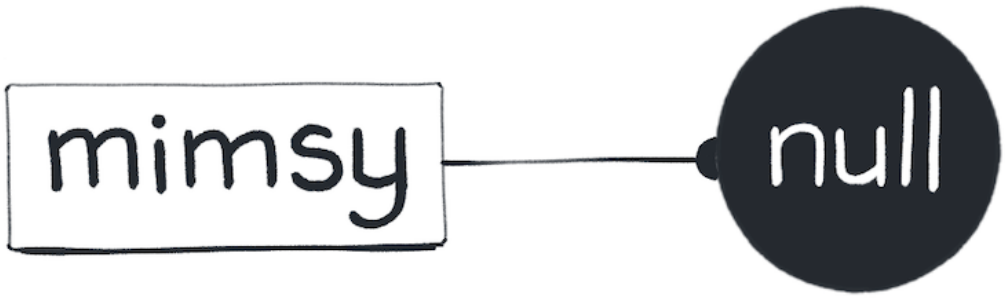
Handle it with care.

Null



You can think of `null` as `undefined`'s sister. It behaves very similarly. For example, it will also throw a fuss when you try to access its properties:

```
let mimsy = null;
console.log(mimsy.mood); // TypeError!
```



Similarly to `undefined`, **`null` is the only value of its own type**. However, `null` is also a liar. Due to a bug in JavaScript, it pretends to be an object:

```
console.log(typeof(null)); // "object" (a lie!)
```

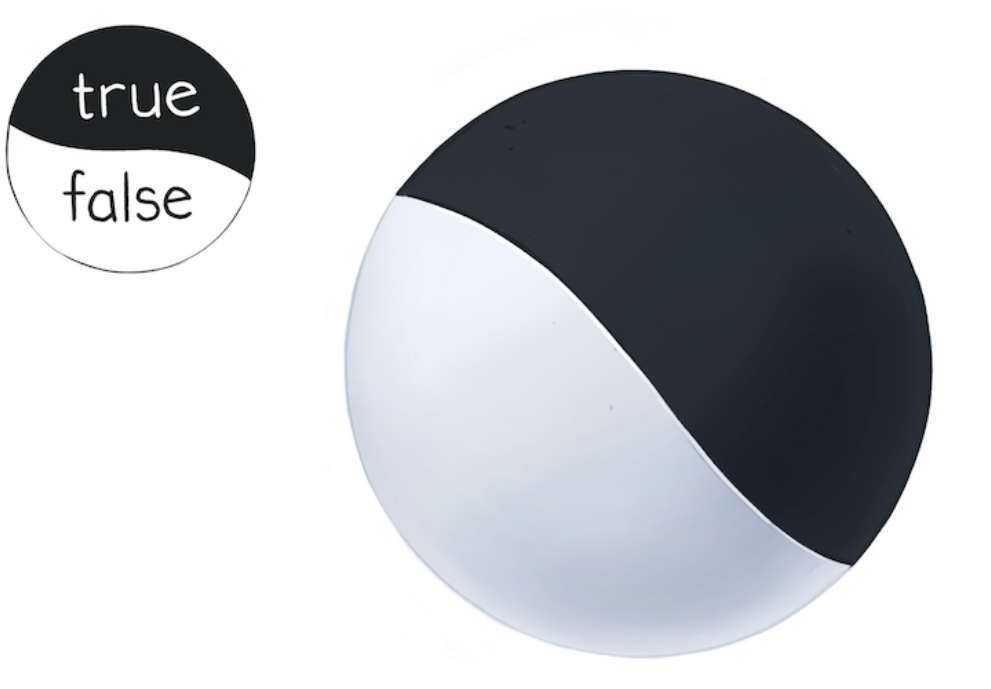
You might think this means `null` is an object. Don't fall into this trap! It is a primitive value, and it doesn't behave in any way like an object. Unfortunately, `typeof(null)` is a historical accident that we'll have to live with forever.

In practice, `null` is used for *intentionally* missing values. Why have both

`null` and `undefined`? This could help you distinguish a coding mistake (which might result in `undefined`) from valid missing data (which you might express as `null`). However, this is only a convention, and JavaScript doesn't enforce this usage. Some people avoid both of them as much as possible!

I don't blame them.

Booleans



Like day and night, **there are only two boolean values: `true` and `false`.**

```
console.log(typeof(true)); // "boolean"
console.log(typeof(false)); // "boolean"
```

We can perform logical operations with them:

```
let isSad = true;
let isHappy = !isSad; // The opposite
let isFeeling = isSad || isHappy; // Is at least one of
them true?
```

```
let isConfusing = isSad && isHappy; // Are both true?
```

Count von Count would like to check your mental model now. Open a sketching app or take a piece of paper, and sketch out the variables, the values, and the wires between them for the above snippet of code.

SPOILERS BELOW

Don't scroll further until you have finished sketching.

...

...

...

...

...

...

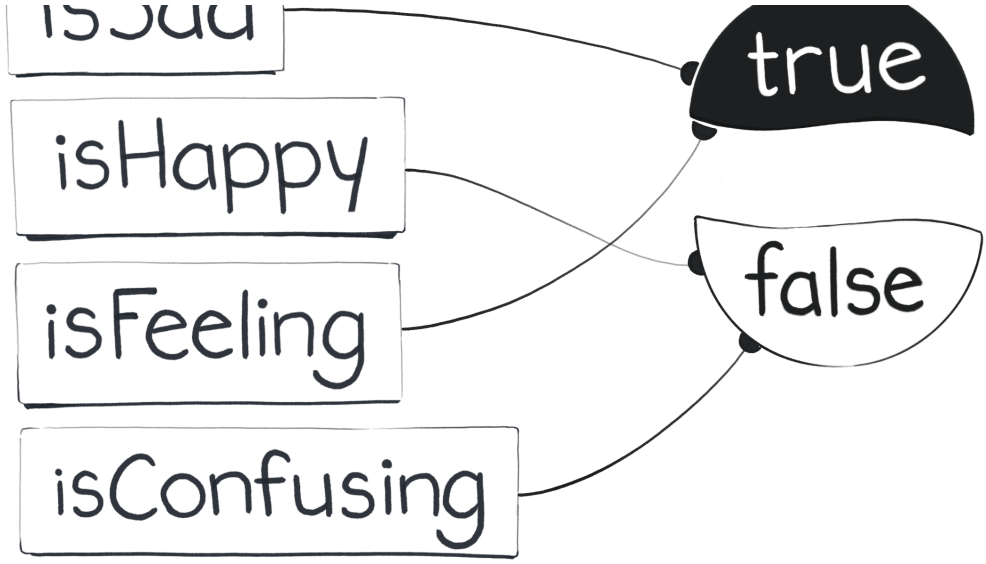
...

...

...

...

Check your answer against this picture:



First, verify that `isHappy` points to `false`, `isFeeling` points to `true`, and `isConfusing` points to `false`. (If you got different answers, there is a mistake somewhere along the way — walk through each line step by step.)

Next, verify that **there is only one `true` and one `false` value on your sketch**. Count von Count insists that this is important! Regardless of how booleans are stored in the memory, *in our mental model* there are only two of them.

Numbers

So far, we counted exactly four values: `null`, `undefined`, `true`, and

So far, we counted exactly four values: `null`, `undefined`, `true`, and `false`.

Hold on, as we will add eighteen quintillion, four hundred and thirty-seven quadrillion, seven hundred and thirty-six trillion, eight hundred and seventy-four billion, four hundred and fifty-four million, eight hundred and twelve thousand, six hundred and twenty-four more values to our mental model!

I am, of course, talking about numbers:

```
console.log(typeof(28)); // "number"
console.log(typeof(3.14)); // "number"
console.log(typeof(-140)); // "number"
```

At first, numbers might seem unremarkable. Let's look closer!

A Math for Computers

JavaScript numbers don't behave exactly the same way as regular mathematical numbers do. Here is a snippet that demonstrates it:

```
console.log(0.1 + 0.2 === 0.3); // false
console.log(0.1 + 0.2 === 0.30000000000000004); // true
```

This might look very surprising! Contrary to a popular belief, this doesn't mean that JavaScript numbers are broken. This behavior is common in different programming languages. It even has a name: *floating point math*.

You see, JavaScript doesn't implement the kind of math we use in real life. Floating point math is "math for computers". Don't worry too much about this name or how it works exactly. Very few people know about all its subtleties, and that's the point! It works well enough in practice that most of the time you won't think about it. Still, let's take a quick look at what makes it different.

Colors and Numbers

Have you ever used a scanner to turn a physical photo or a document into a digital one? This analogy can help us understanding JavaScript numbers.

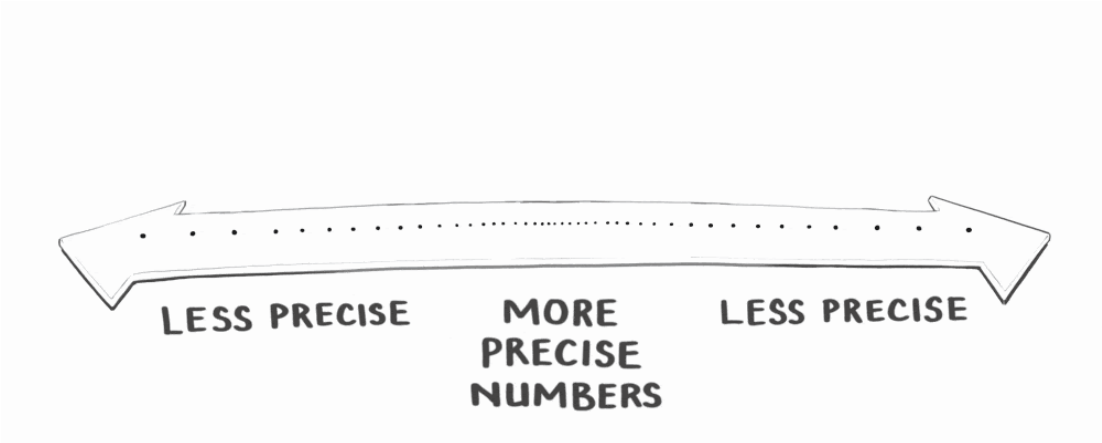
Scanners usually distinguish at most 16 million colors. If you draw a picture with a red crayon and scan it, the scanned image should come out red too — but it will have the *closest* red color our scanner picked from those 16 million colors. So if you have two red crayons with ever so slightly different colors, the scanner might be fooled into thinking their color is exactly the same!

We can say that a scanner treats colors as having a limited precision.

Floating point math is similar. In real math, there is an infinite set of numbers. But in floating point math, **there are only 18 quintillion of them.** So when we write numbers in our code or do calculations with them, JavaScript picks the *closest* numbers that it knows about — just like our scanner does with colors.

In other words, JavaScript treats numbers as having a limited precision.

We can imagine all of the JavaScript numbers on an axis. The closer we are to `0`, the more precision numbers have, and the closer they "sit" to each other:



As we move from `0` in either direction, we start losing precision. At some point, even two closest JavaScript numbers stay further apart than by `1`:

```
console.log(Number.MAX_SAFE_INTEGER);      //
```

```
9007199254740991
console.log(Number.MAX_SAFE_INTEGER + 1); //
9007199254740992
console.log(Number.MAX_SAFE_INTEGER + 2); //
9007199254740992
console.log(Number.MAX_SAFE_INTEGER + 3); //
9007199254740994
console.log(Number.MAX_SAFE_INTEGER + 4); //
9007199254740996
console.log(Number.MAX_SAFE_INTEGER + 5); //
9007199254740996
```

Luckily, any **whole** numbers between `Number.MIN_SAFE_INTEGER` and `Number.MAX_SAFE_INTEGER` are exact. This is why `10 + 20 === 30`.

But when we write `0.1` or `0.2`, we don't get *exactly* `0.1` and `0.2`. We get the closest available numbers in JavaScript. They are almost exactly the same, but there might be a tiny difference. These tiny differences add up, which is why `0.1 + 0.2` doesn't give us *exactly* the same number as writing `0.3`.

If this is still confusing, don't worry. You can learn more about floating point math, but you already know more than I did when I started writing this guide! Unless you work on finance apps, you likely won't need to worry about this.

Special Numbers

It is worth noting that floating point math includes a few *special numbers*. You might occasionally run into `NaN`, `Infinity`, `-Infinity`, and `-0`. They exist because sometimes you might execute operations like `1 / 0`, and JavaScript needs to represent their result somehow. The floating point math standard specifies how they work, and what happens when you use them.

Here's how special numbers may come up in your code:

```
let scale = 0;
let a = 1 / scale; // Infinity
let b = 0 / scale; // NaN
```

```
let b = 0 / scale; // NaN
let c = -a; // -Infinity
let d = 1 / c; // -0
```

Out of these special numbers, `NaN` is particularly interesting. `NaN`, which is the result of `0 / 0` and some other invalid math, stands for “not a number”.

You might be confused by why it claims to be a number:

```
console.log(typeof(NaN)); // "number"
```

However, there is no trick here. From JavaScript perspective, `NaN` *is* a numeric value. It is not null, undefined, a string, or some other type. But in the floating point math, the *name* for that term is “not a number”. So it *is* a numeric value. It happens to be called “not a number” because it represents an invalid result.

It’s uncommon to write code using these special numbers. However, they might come up due to a coding mistake. So it’s good to know they exist.

To Be Continued

This module is split in two parts. We’ve reached the end of the Part 1. We’ll take a small break now. Let’s recap how many values we’ve counted so far!

- **Undefined**: Only one value, `undefined`.
- **Null**: Only one value, `null`.
- **Booleans**: Two values: `true` and `false`.
- **Numbers**: One value for each floating point math number.

We've also learned a few interesting facts about JavaScript numbers:

- **Not all numbers can be perfectly represented in JavaScript.** Their decimal part offers more precision closer to `0`, and less precision further away from it. We can say that their decimal point is "floating".
- **Numbers from invalid math operations like `1 / 0` or `0 / 0` are special.** `NaN` is one of such numbers. They may appear due to coding mistakes.
- **`typeof(NaN)` is a number because `NaN` is a numeric value.** It's called "Not a Number" because it represents the *idea* of an "invalid" number.

In the second part, we will continue our sightseeing tour. We will look at BigInts, Strings, Objects, and Functions — and try to count them all.

Exercises

This module also has exercises for you to practice!

Click here to solidify this mental model with a few short exercises.

Don’t skip them!

Even though you’re probably familiar with different types of values, these exercises will help you cement the mental model we’re building. We need this foundation before we can get to more complex topics.

Cheers,
Dan