Gmail

Dhaval Chheda <kiddo.dhaval@googlemail.com>

[Just JavaScript] 03. Values and Variables

1 message

Dan Abramov <dan@overreacted.io> 28 May 2020 at 23:23
To: Dhaval <kiddo.dhaval@gmail.com>

We'll kick off this module with a little code snippet.

```
let reaction = 'yikes';
reaction[0] = 'l';
console.log(reaction);
```

What do you expect it to do? We haven't covered this yet so it's okay if you're not sure. **Try to answer it using your current knowledge of JavaScript.**

Now I want you to take a few moments and write down your exact thinking process for each line of this code, step by step. Pay attention to any gaps or uncertainties in your existing mental model, and write them down too. If you have any doubts about it, try to articulate them as clearly as you can.

SPOILERS BELOW

Don't scroll further until you have finished writing.

...

...

...

...

...

...

...

...

...

...

Here's the answer. This code will either print `"yikes"` or throw an error depending on whether you are in strict mode. **It will never print `"likes"`.**

Yikes.

Primitive Values Are Immutable

Did you get the answer right? This might seem like a trivia question, the kind that people ask in JavaScript interviews but that doesn't come up much in practice. Even so, it illustrates an important point about primitive values.

I can't change primitive values.

I will explain this with a small example. Strings (which are primitive) and arrays (which are not — they're objects!) have some superficial similarities. An array is a sequence of items, and a string is a sequence of characters:

```
let arr = [212, 8, 506];
let str = 'hello';
```

You can access the first array item similarly to how you would access a string's first character. It almost feels like strings are arrays (but they're not!):

```
console.log(arr[0]); // 212
console.log(str[0]); // "h"
```

You can change an array's first item:

```
arr[0] = 420;
console.log(arr); // [420, 8, 506]
```

So intuitively, it's easy to assume that you can do the same to a string:

```
str[0] = 'j'; // ???
```

But you can't.

Here's an important bit that we need to add to our mental model. A string is a primitive value. And that means a great deal!

All primitive values are immutable. "Immutable" is a fancy Latin way to say "unchangeable". Read-only. You can't mess with primitive values. At all.

If you attempt to set a property on a primitive value, be it a number or a string or something else, JavaScript won't let you do that. Whether it will silently refuse your request or error depends on which mode your code is running in.

But stay assured that this will never work:

```
let fifty = 50;
fifty.shades = 'gray'; // No!
```

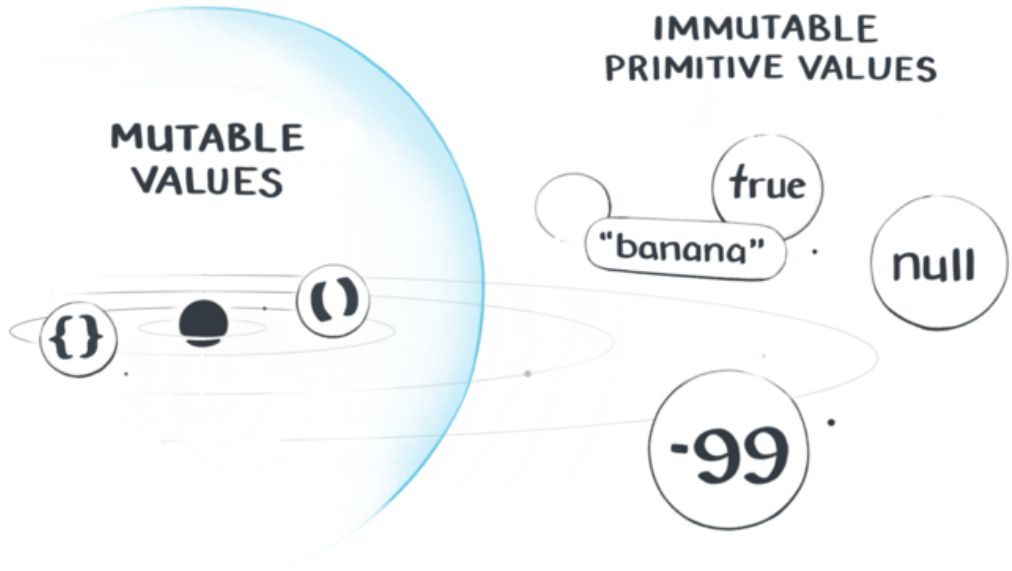
Like any number, `50` is a primitive value, and you can't set properties on it.

Can't touch this.

In my JavaScript universe, all primitive values exist in the outer circle further from my code — like distant stars. This reminds me that even though I can

refer to them from my code, I can't change them. They stay what they *are*.

I find it strangely comforting.



A Contradiction?

I have just demonstrated that primitive values are read-only — or, in the parlance of our times, immutable. Here's a snippet to test your mental model.

```
let pet = 'Narwhal';
pet = 'The Kraken';
console.log(pet); // ?
```

Like before, write down your thinking process in a few sentences. Don't rush ahead. Pay close attention to how you're thinking about each line, step by step. Does immutability of strings play a role here, and what role does it play?

SPOILER ALERT

...

...

...

...

...

...

...

...

...

...

...

If you thought I was trying to mess with your head, you were completely right! The answer is `"The Kraken"` — immutability of strings doesn't play a role.

Don't despair if you got it wrong and haven't seen it coming! These two last examples may definitely seem like they're contradicting each other.

This is an important realization.

When you're new to a language, it might be tempting to ignore contradictions. After all, if you chase every contradiction, you'll get into a rabbit hole too deep to learn anything. But now that you are committed to building a mental model, you need to question contradictions. They reveal gaps in mental models.

Variables Are Wires

Let's look at this example again.

```
let pet = 'Narwhal';
pet = 'The Kraken';
console.log(pet); // "The Kraken"
```

We know that string values can't change because they are primitive. But the `pet` variable *does* change to `"The Kraken"`. What's up with that?

This might seem like it's a contradiction, but it's not. We only said it's the primitive *values* that can't change. We didn't say anything about *variables*!

As we refine our mental model, we might need to untangle related concepts.

Variables are not values.

Variables point to values.

In my universe, a variable is a wire. It has two ends and a direction: it starts from a name in my code and it ends pointing at some value in my universe.

For example, I can point the `pet` variable at the `"Narwhal"` value:

```
let pet = 'Narwhal';
```

There are two things I can do to a variable after that.

Assigning a Value to a Variable

One thing I can do is to *assign* some other value to my variable:

```
pet = 'The Kraken';
```


```
let pet = 'Narwhal'
pet = 'The Kraken'
```

All I am doing here is instructing JavaScript to point the "wire" on the left side (my `pet` variable) at the value on the right side (`'The Kraken'`). It will keep pointing at that value unless I re-assign it again later.

Note that I can't just put *anything* on the left side:

```
'war' = 'peace'; // Nope. (Try it in the console.)
```

The left side of an assignment must be a "wire". For now, we only know that variables are "wires". But there is another kind of "wire" we'll talk about in a later module. Perhaps, you can guess what it is? (Hint: it involves square brackets or a dot, and we've already seen it a couple of times.)

There's also another rule.

The right side of an assignment must be an expression. It can be something simple, like `2` or `'hello'`, or a more complicated expression — for example:

```
pet = count + ' Dalmatians';
```

Here, `count + ' Dalmatians'` is an expression — a question to JavaScript. JavaScript will answer it with a value (for example, `"101 Dalmatians"`). From now on, the `pet` "wire" will start pointing to that value.

If the right side must be an expression, does this mean that numbers like `2` or strings like `'The Kraken'` written in code are also expressions? Yes! Such expressions are called *literals* — because we *literally* write down their values.

Reading a Value of a Variable

I can also *read* the value of variable — for example, to log it:

```
console.log(pet);
```

That's hardly surprising.

But note that it is not the `pet` *variable* that we pass to `console.log`. We might say that colloquially, but we can't really pass *variables* to functions. We pass the current *value* of the `pet` variable. How does this work?

It turns out that a variable name like `pet` can serve as an expression too! When we write `pet`, we're asking JavaScript a question: "What is the current value of `pet`?" To answer our question, JavaScript follows the `pet`'s "wire", and gives us back the value at the end of this "wire".

So the same expression can give us different values at different times!

Nouns and Verbs

Who cares if you say “pass a variable” or “pass a value”? Isn’t the difference hopelessly pedantic? I certainly don’t encourage interrupting your colleagues to correct them — or even yourself. That would be a waste of everyone’s time.

But in your mind you need to have clarity on *what you can do* with each concept. You can’t skate a bike. You can’t fly an avocado. You can’t sing a mosquito. And you can’t pass a variable — at least, not in JavaScript.

Here’s a small example of why these details matter.

```
function double(x) {
  x = x * 2;
}

let money = 10;
double(money);
console.log(money); // ?
```

If we thought `double(money)` was passing a *variable*, we could expect that `x = x * 2` would double that variable. But that’s not how it works. We know that `double(money)` means “figure out the *value* of `money`, and then *pass that value* to `double`”. So the answer is `10`. What a scam!

What are the different JavaScript nouns and verbs in your head? How do they relate to each other? Write down a short list of the ones you use most often.

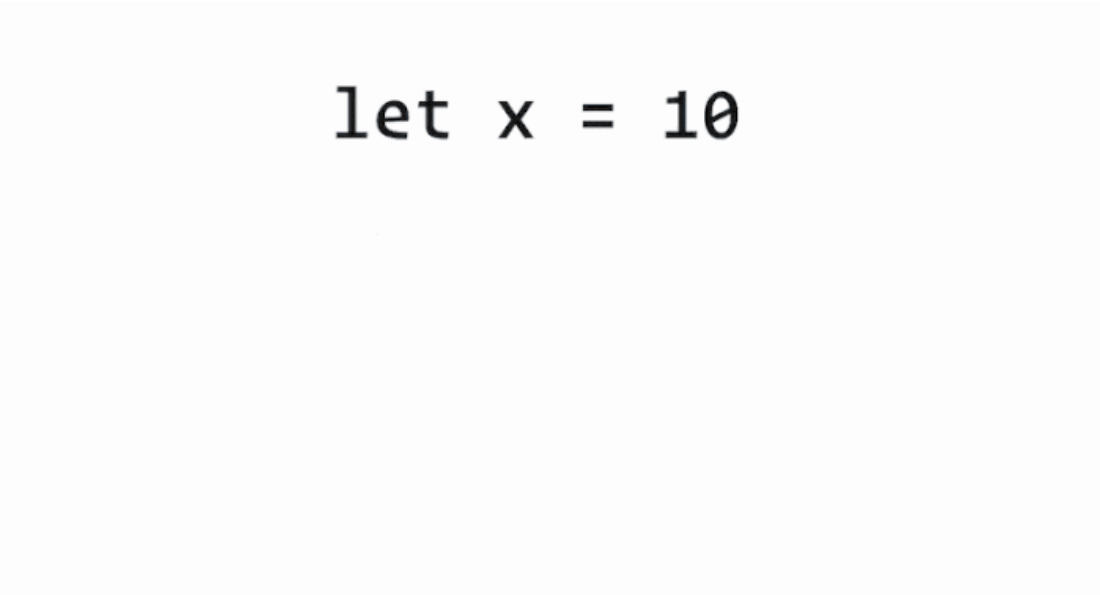
Putting It Together

Now let’s revisit the first example from *Mental Models*:

```
let x = 10;
let y = x;
x = 0;
```

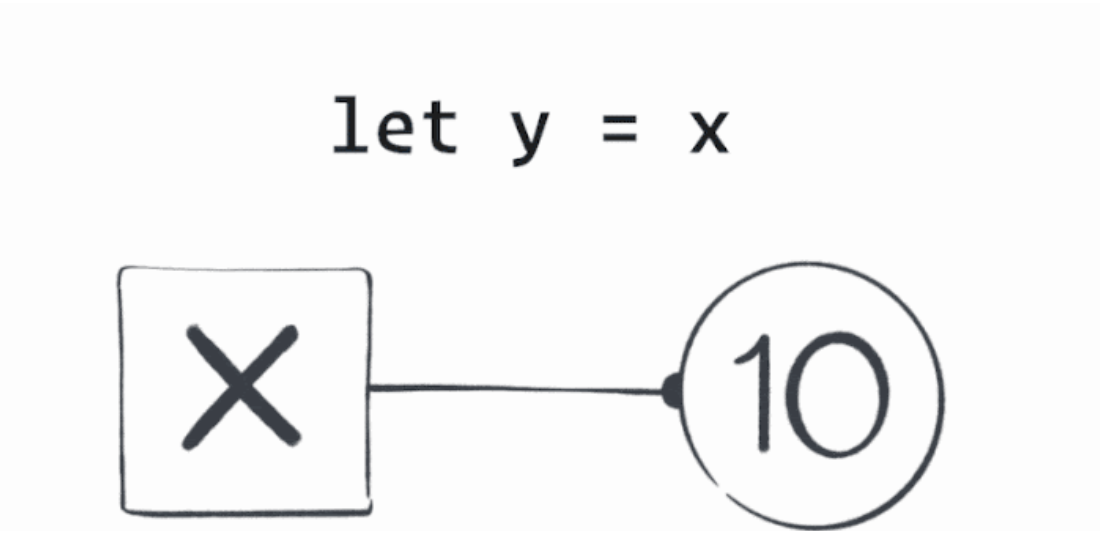
I suggest that you take a piece of paper or a drawing app and sketch out a diagram of what happens to the "wires" of the `x` and `y` variables step by step.

The first line doesn't do much:



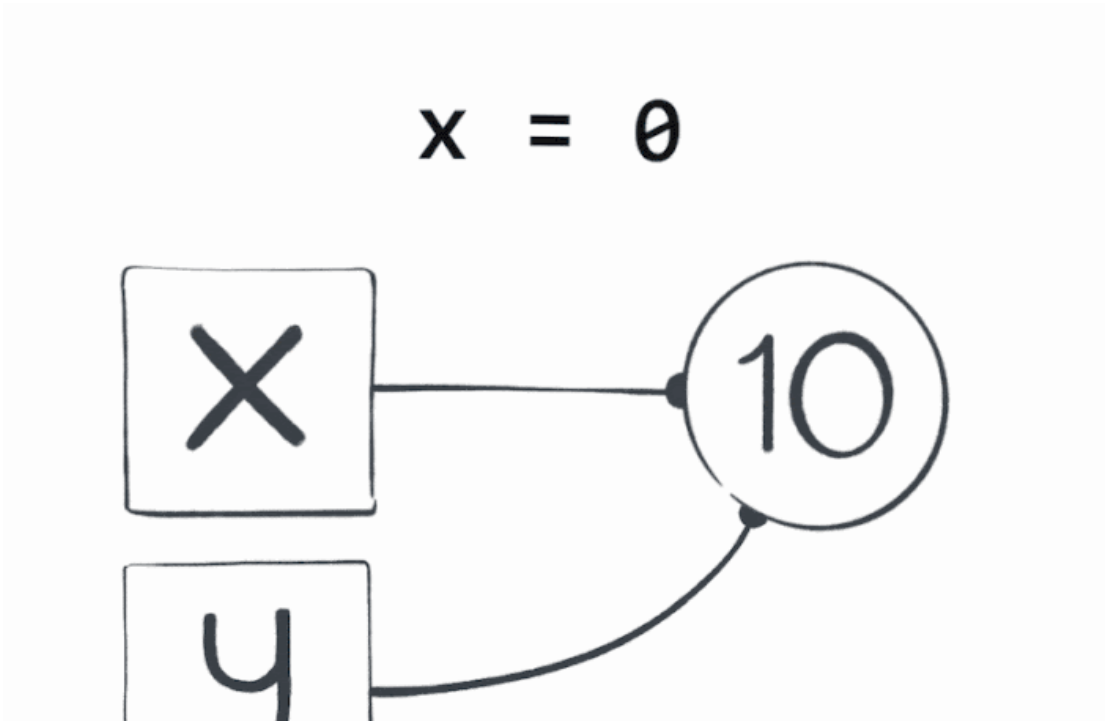
- Declare a variable called `x`.
  - *Make a wire for the `x` variable.*
- Assign to `x` the value of `10`.
  - ***Point `x`'s wire to the value `10`.***

The second line is short, but it does quite a few things:



- Declare a variable called `y`.
    - *Make a wire for the `y` variable.*
- Assign to `y` the value of `x`.
    - Evaluate the expression: `x`.
        - *The "question" we want to answer is `x`.*
        - ***Follow the `x`'s wire — the answer is the value `10`.***
    - The `x` expression resulted in the value `10`.
    - Therefore, assign to `y` the value of `10`.
    - ***Point `y`'s wire to the value `10`.***

Finally, we get to the third line:



- Assign to `x` the value of `0`.
    - ***Point `x`'s wire to the value `0`.***

At the end, the `x` variable points to the value `0`, and the `y` variable points to the value `10`. Note that `y = x` did not mean point `y` to `x`". We can't point variables to each other! **Variables always point at *values*.** When we see an assignment, we "ask" the right side's value, and point the left side's "wire" at it.

I mentioned in *Mental Models* that it is fairly common to think of variables as boxes. The universe we're building is not going to have any boxes at all. **It only has wires!** This might seem a bit annoying. Why can't we just "put `0` and `10` values *into* the variables rather than *pointing* variables to them?

Using wires is going to be very important for explaining numerous other concepts, like strict equality, object identity, and mutation. We're going to stick with wires, so you might as well start getting used to them now!

My universe is full of wires.

Recap

- **Primitive values are immutable.** There's nothing we can do in our code to affect them or change them in any way. They stay what they are. For example, we can't set a property on a string value because it is a primitive value. Arrays are *not* primitive, so we *can* set their properties.
- **Variables are not values.** Each variable *points to* a particular value. We can change *which* value it points to by using the = assignment

We can change *which* value it points to by using the = assignment operator.

- **Variables are like wires.** A “wire” is not a JavaScript concept — but it helps us imagine how variables point to values. There’s also a different kind of “wire” that’s not a variable, but we haven’t discussed it yet.
- **Look out for contradictions.** If two things that you learned seem to contradict each other, don’t get discouraged. Usually it’s a sign that there’s a deeper truth lurking underneath.
- **Nouns and verbs matter.** We’re building a mental model so that we can be confident in what *can* or *cannot* happen in our universe. It’s fine to be sloppy in casual speech, but our thinking needs to be precise.

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 likely familiar with the concept of variables, 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