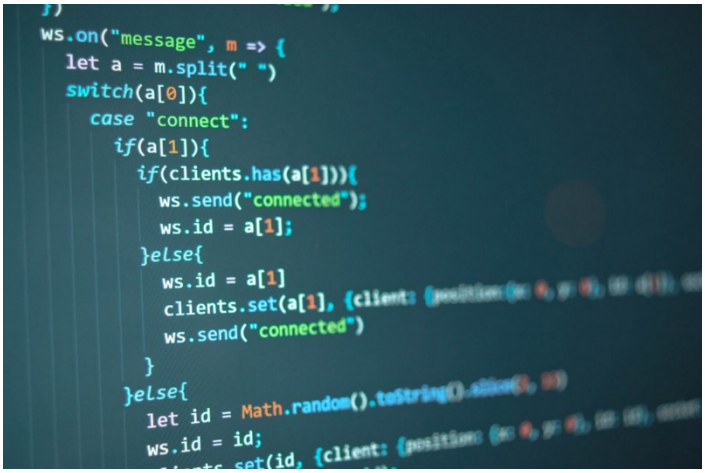
Javascript Introduction

javaScript: The Programming Language of the Web

Christian Grewell Follow
Feb 6 · 9 min read



JavaScript makes the web interactive

INTRODUCTION

One of the hardest things to learn in programming is not the syntax, but how to apply it to solve real world problems. I want you to start thinking like a programmer—this generally involves looking at descriptions of what your program needs to do, working out what code is needed to achieve those things, and how to make them work together.

These labs are designed to give you experience with programming syntax and practice. The more you build stuff with code, the better you'll get at it. I can't promise that you'll develop "programmer brain" in five minutes, but you'll have many opportunities to practice thinking like an 'entrepreneurial programmer' throughout the course.

In this lab, you'll be introduced to the process of writing code across a variety of environments, and a healthy dose of basic Javascript concepts and syntax.

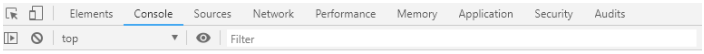
- You'll use your text editor (VSCode)to write the JavaScript code.
- You'll use a Web Browser (then eventually your own computer) to execute your project code.

JAVASCRIPT VALUES

In order to make useful programs, we must be able to take the billions of bits in our computer, and separate them in a way that allows us to create `values` .

```
30;
```

That's a value (it's also an expression which we'll get to later). If you try and run this in an environment, you'll get nothing, well not exactly nothing. The node runtime environment created this value (a number, 30) for maybe a 1/10,000,000th of a second. Then it was gone.



you just wrote your first JavaScript code!

Let's make the computer work for us. Open up your browser (or VIM if you are a *hacker*), type `console.log(10);`

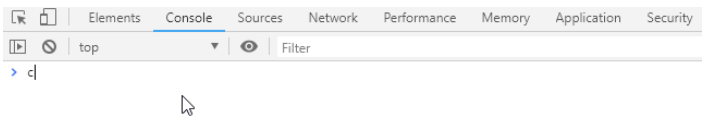
```
console.log(10);
//10
```

```
console.log(9.123456);
//9.123456
```

The first console.log returns a **number**. The second also returns a **number**. Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

Try this:

```
console.log(9999999999999999);
```



Javascript uses 64 bits (0/1 pairs) to store numbers. This is a huge number, ^{264}. How big is that? Well, we're coding now, can you make the computer do this for you?

. . .

```
CHALLENGE #1: make a program to calculate 2^26
you can Google this if you want!

note: I've added some blank space before the solution so
that you can choose not to peek.

--solution below (don't peak!)--

SOLUTION:
console.log(2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2);

console.log(Math.pow(2, 64)); //better way...
```

Actually, that's not exactly the maximum value possible. It's actually, it's closer to 9 quadrillion when you factor in the extra bit to denote negative or positive, and the bits needed to store the location of the decimal point. This is called a signed number, specifically, if we're not talking about decimals, but integers (whole numbers). Then the actual number is only:

```
9,223,372,036,854,775,807
```

That's 9 quadrillion, still a **big number**.

ARITHMETIC IN JAVASCRIPT

Let's do some simple arithmetic in JavaScript.

1. `console.log(20 + 15 * 4);`
2. `console.log((20 + 15) * 4);`

Javascript has the following *obvious* operators :

- Addition (+)
- Subtraction (—)
- Multiplication (*)
- Division (/)

In addition, it has a few others, specifically the **modulus** operator, which returns a remainder:

```
console.log(101 % 100);
```

STRINGS

Another value type in Javascript is the `string` value. There are a few different ways to write them (one is especially powerful, you'll see).

```
"I love dumplings" -- double quotes

'I love dumplings' -- single quotes

`I love dumplings` -- backticks
```

If we write a simple program to log our favorite food `console.log(`I love dumplings.`);` we get the result: `I love dumplings.`

```
$node dumplings.js

I love dumplings

but I also love

tabs!
```

How could we make the computer output the text as formatted above?

The Escape Character `\`

Javascript has something called an *escape character* that tells the program you'd like to do things like start a new line, or add a tab.

- `\'` single quote
- `\"` double quote
- `\\` backslash
- `\n` new line
- `\r` carriage return
- `\t` tab
- `\b` backspace
- `\f` form feed

If you want to add a new line in the middle of a string, you could write something like `console.log('I am a little teapot, short and stout\nhere is my handle, here is my spout');`

. . .

```
CHALLENGE #2: make a program to output the following text:

I love dumplings

but I also love

    tabs!
```

```
--solution below (don't peak!)--
```

```
SOLUTION:
console.log('I love dumplings\nbut I also love\n\ttabs');
```

The Backtick ``

Creating string values with backticks `` `hi!` `` come with more than a few benefits. For one, you don't need to use the escape character to complete the challenge about. You can just write this with backticks formatted as you'd like:

```
> console.log('I love dumplings\nbut I also love\n\ttabs');
  I love dumplings
  but I also love
      tabs
```

STRING INTERPOLATION + CONCATENATION

You can also perform arithmetic on string values. well, technically not arithmetic, but you can add strings together:

```
console.log('my favorite' + 'cat' + 'is' + 'concat');
console.log('my favorite' + ' cat' + ' is' + ' con' + 'cat'
);


//my favoritecatisconcat
//my favorite cat is concat
```

Strings also have methods that you can call on to perform operations on them.

. . .

```
CHALLENGE #3:
write a program that outputs a NUMBER value for the length
of the string 'abaracadabara'
```

```
start here: https://developer.mozilla.org/en-US/


--solution below--
```

```
SOLUTION:
console.log('abaracadabara'.length); //13
```

Back-tick quotes are very powerful, we've already seen how they can be used to avoid having to use escape characters. They can also do something called *template literals* and *string interpolation*

What happens when you run this code?

```
console.log(`half of 200 is ${200/2}`);
```

The `typeof` Operator

The `typeof` operator returns a string indicating that tells us the type of value

```
console.log(typeof 45);
//number
```

```
console.log(typeof 'Bob Ross')
//string
```

BOOLEANS

Booleans are for when you want to distinguish between two values. In Javascript, these are `true` and `false`.

```
console.log(10 < 9);
//false
```

```
console.log('Bob Ross' < 'Ross Bob');
//true
```

OPERATORS

`<` less than

`>` greater than

`<=` less than or equal

`>=` greater than or equal

`===` equal to

`!==` not equal to

`==` truthy

`!=` falsey

And some logical operators:

`&&` and

`||` or

`!` not

```
console.log('Bob Ross' != 'Bob Moss');
//true
```

```
console.log("5" == 5);
//true
```

The example above is because JS is extremely forgiving. It's doing something called *type coercion*, which is a fancy way of saying JavaScript is trying to interpret what you really mean

```
console.log("4" + 2);
//42
```

```
console.log("4" * 2);
//8
```

```
console.log(true == 1);
//true
```

```
console.log(true === 1);
//false
```

```
console.log('5' !== 5);
//true
```

This is why we say the two operators `==` and `!=` are truthy and falsey respectively. In general, if you don't want accidental type coercion, you should use `===` and `!==`, which will make sure to match on the `TYPEOF` the value as well.

LOGICAL OPERATORS (and `&&` or `||` and not `!`)

The and `&&` operator will evaluate to `true` if all statements are true:

```
console.log(true && false);
//false
```

```
console.log(true && true);
//true
```

The or `||` operator will evaluate to true if one of the statements is true

```
console.log(true || false);
//true
```

```
console.log(false || false);
//false
```

The not `!` operator will flip the value that you give it

```
console.log(!true);
//false
```

```
console.log(!(1 === 2));
//true
```

The Ternary Operator

So far, we've been working with binary operators, they evaluate as `true` or `false`. The third, and very useful operator, is the TERNARY operator

```
console.log(true ? “Bears” : “Sharks”);
//Bears
```

```
console.log(false ? “Bears” : “Sharks”);
//Sharks
```

Syntax format: `x ? y : z` where **`x`** `= what you want to evaluate`, **`y`** `= the value or statement if` **`true`**, and **`z`** `the value or statement if` **`false`**.

The ternary operator evaluates a statement and if true, returns whatever is after the ?, otherwise it returns whatever is after the :

```
console.log(true ? 2 + 2 : 4 + 4);
//4
```

EXPRESSIONS AND STATEMENTS

Expressions in Javascript are essentially sentence fragments. You can add many expressions together to form a statement, just like you can add sentence fragments together to form a sentence

```
!true;
//this is technically an expression. Not a good one, but and
expression.
```

BINDINGS / VARIABLES

The bread and butter of your programs, they give you the ability to grab a hold of the bits in your computer and assign names to them.

The *concept* of bindings is the most important thing you can possibly learn. In JavaScript, bindings are used to point to values—bits of 1s and 0s that make up your program. You can create bindings to not just to values, but to entire functions and objects in your program.

You create bindings in JavaScript using the *let, const, or var* expressions. In general, avoid using the *var* keyword to create a binding.

In general, you use the `const` expression when you're sure the value won't be reassigned. However, a mutable variable is often needed, particularly as a counter when we need to loop through something.

But why should you use `let` instead of `var` for these situations, if they both provide the basic mutability required of them?

`let` , is a signal that **the binding may be reassigned**, it also signals that the variable will be used **only in the block it's defined in (we'll get to blocks when we talk about functions in the next lab)**, which is not always the entire containing function.

confused? don't worry, in general you are going to be just fine using `const` *, and if you find the program has an error, you can switch to using* `let` *, just avoid* `var` *, it's not the future…*

So, how can we create a binding?

```
let catWeight = 30; //we use let here because this cat needs
to lose weight
```

```
console.log(catWeight);
//30
```

This statement creates a binding called 'catWeight' and points it to a location in memory with the following binary value: `00011110`

```
const numberOfCats = 4;
console.log(catWeight * numberOfCats); //30 * 4)
//120
```

. . .

```
CHALLENGE 4: Now that we have two bindings, and we remember
string interpolation, we can use these to create output.
```

```
Log a message to the console that tells someone how many
cats they have, and tells them how many bags to bring,
assuming one bag can carry 10 kilos.

--solution below--
```

```
console.log(`You have ${numberOfCats} cats. You need
${(catWeight*numberOfCats)/10} bags to carry them all!`);
```

You can also change bindings, and point them to a new value (including other bindings)

```
const numberOfCats = 10;
```

```
console.log(`You have ${numberOfCats} cats. You need
${(catWeight*numberOfCats)/10} bags to carry them all!`);
```

BINDINGS AND VALUES: THE RUNTIME ENVIRONMENT

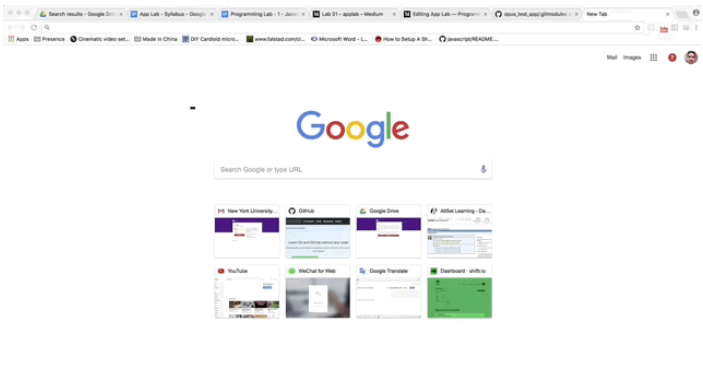
The environment that the program runs in also has a bunch of other pre-defined bindings. `console.log()` is actually a pre-defined binding, the console function has a method that returns whatever is in the log to your screen.

Web browsers also have a few other pre-defined bindings, things that help with showing prompts to a user, or responding to mouse clicks.

FUNCTIONS

We'll get into functions in a lot of detail in the next lab, but for now, know that they are a set of statements that perform tasks or calculates a value. In order to use a function, you need to create one, or you can use functions in your runtime environment (e.g., the browser that we've been using so far)

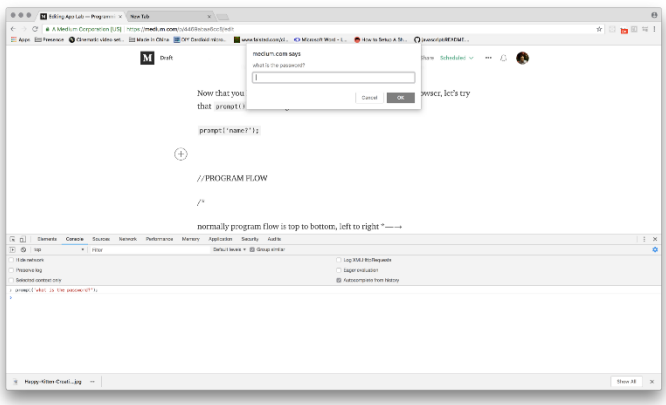
You've already been using a JavaScript function, `console.log` :



We can write JavaScript directly in the browser's runtime environment. Hax!

There are a slew of built-in functions that come with your browser's JavaScript runtime, for example, try typing this:

```
prompt('what is the password?');
```



The result of: prompt('what is the password?');

. . .

COMMENTING

So until now, we've just been rolling along writing some simple code, making the computer do things and generally getting along fine. You've probably noticed during our live exercises that I've been commenting some of the code in the following two ways.

`//` denotes a single line comment. These are good for when you have something terse to say

The other type of comment is a multi-line comment. These are great for big blocks of documentation or text that spans multiple lines.

```
/*
write your comment here
*/
```

It's required to start with the comment with a `/*` and end the comment with a `*/` otherwise the compiler will interpret your comment as code!

PSUEDOCODE

Pseduocode is a helpful way to start a program. Essentially you use plain English (or whatever language you'd like really) to outline what a program will do, trying to stay as detailed as possible and true to syntax forms.

```
//IF there is a bear and it is closer than 10 meters
     //IF we have bear spray
          //spray the bear
          //run
     ELSE IF it is a Brown or Grizzly Bear
          //play dead
     ELSE
          //raise your hands and shout
//ELSE
     //run away
```