JavaScript Calculator

The [last chapter](/1-environment/) ended with the following, in the console:

> console.log("Hello, World!");  
//--> Hello, World!

Note that I have included the > prompt in this example, along with the how the console responded to your single line of JavaScript, after the //-->. There’s a lot going on in just this one example, but, briefly, console.log() is a way of using JavaScript to tell the console to output something.[[1]](#footnote-22) Before we get too carried away with writing JavaScript, however, I think it might be useful to learn a bit about JavaScript’s history.

## Hello, World! Hello, JavaScript!

I mentioned a few details about JavaScript’s history in the [FAQs](%7B%7B%20site.baseurl%20%7D%7D/0-introduction/), but I’ll reiterate some of that now.

Programming languages inherit from previous programming languages. The writer(s) of a language will take parts they like from one language and mix it with parts they like from another, and then add their own, new additions that makes their language special. JavaScript is no different; in fact, JavaScript might be especially good at revealing its mixed parentage. But that means, concretely, that it helps to have a sense of the history of programming in general when understanding why JavaScript looks like it does today.

The [fourth episode of the BBC show *Connections* draws a line](http://www.dailymotion.com/video/x3dvbkg) to computer programming that begins with the Roman era [Barbegal aqueduct and mill](https://en.wikipedia.org/wiki/Barbegal_aqueduct_and_mill). Sadly, James Burke concludes his wry take on advances in technology with punch cards, but he was also filming *Connections* in 1978. Nevertheless, Burke sees that the key to how computers work is related to how they *organize information*. How to organize information is, not coincidentally, the first part of programming that we will learn in this chapter.

Speaking a few decades later, Douglas Crockford picks up the thread from where Burke ended it (punchcards) and moves us all the way up to JavaScript in two of his *Crockford on JavaScript* lectures, “[The Early Years](https://www.youtube.com/embed/JxAXlJEmNMg)” and the first 20 minutes of “[And Then There Was JavaScript](https://www.youtube.com/watch?v=RO1Wnu-xKoY).” Although the lectures are aimed at people with some programming knowledge, meaning that they get pretty geeky, what Crockford does well is hammer home two key points in the way computing has evolved over time:

* The people who should be the first to recognize the value of an innovation (like programmers) are often the last.
* Obsolete technologies fade away slowly.

The implications of both points reveal why JavaScript has its quirks and why those quirks are here to stay.

The short version of Crockford’s history boils down to the fact that JavaScript was written over ten days by one man, [Brendan Eich](http://en.wikipedia.org/wiki/Brendan_Eich). Netscape wanted a scripting language for their web browser, Navigator, and Eich delivered. At the time, Java was positioned to be “the language for the web,” as we would be running Java applets on websites, so Netscape called Eich’s language “JavaScript,” despite the fact that JavaScript inherited nearly nothing substantial from Java.

Once Microsoft put their *own* version of JavaScript in Internet Explorer, it became clear that in order for JavaScript to be useful across the entire web, it would have to be standardized. This took time, and, what’s more, standardizing involved a process of maintaining compatibility. As a result, JavaScript, as Crockford is fond to point out, has a lot of bad parts that are here to stay. Nevertheless, he adds, it also has some *very good* parts. After all, back in 2001 Crockford called JavaScript “[the world’s most misunderstood programming language](http://javascript.crockford.com/javascript.html).” Yet now programmers are focusing more on these good parts (it helps that Crockford even wrote a book [about JavaScript’s good parts](http://shop.oreilly.com/product/9780596517748.do)), which is part of what makes JavaScript so popular.

As Crockford likes to point out, JavaScript is many things to many people, allowing itself to be used in three different programming paradigms. It can be a [procedural](https://en.wikipedia.org/wiki/Procedural_programming) language. It can be used as a [functional](https://en.wikipedia.org/wiki/Functional_programming) language. Or it can be used as an [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) language. Or all three in the same application. These are details that go beyond the scope of this course, but they gesture towards the idea that JavaScript is very flexible and very forgiving. This is part of why it’s so easy to learn. But it’s also why it’s so easy for programmers to cause trouble with it.[[2]](#footnote-35)

Now that we’ve got a bit of an introduction to the language’s history and context out of the way, we can go back to the JavaScript console you learned to open in the [last chapter](/1-environment/) and start learning the language itself.

## Basic data types

We hear the word “data” every day, but what, precisely, does it mean? If you had to draw a picture of “data,” what would it look like? We can say “data is information,” but doesn’t that just pass the buck, because now we have to ask what “information” means?

It maybe helps to remember that the word “data” is the plural version of the Latin word “datum.” A datum is a single piece of information, abstract as that sounds. It can be a number. It can be string of letters. It can be the answer to the question “true or false?” It can also be the *lack* of information.

Data can also be more complex. A datum can be a list or collection of other pieces of data, or it can be a process that takes some data and generates new data based on that data.

These six possibilities (and more exist) make up the six most important **data types** in JavaScript, namely **number**, **string**, **boolean**, **null** / **undefined**, **array** / **object**, and **function**. We’ll only be working with the first four types, the basic types, in this chapter.

### Number

Numbers are precisely what they sound like—numbers. In JavaScript, it’s easy to use numbers, because you just type them as numbers. Numbers can have decimal points if they’re not integers. Try typing numbers into the console and see what happens:

> 9;  
//--> 9  
> 1.5;  
//--> 1.5  
> console.log(9);  
//--> 9

Your console may look a bit different in what it returns (including undefined after console.log(9);), but we can ignore that for now.

But note also that I type a ; at the end of every command. JavaScript is not very picky with semicolons, but it’s better to include them as a matter of habit. The semicolon comes at the end of a “statement,” which is a single instruction given to the console. 9; just means “the number 9.” console.log(9); means “log the number 9 to the console.”

### String

Strings are a bit trickier than numbers at first glance. They are any string of characters enclosed by double quotes.[[3]](#footnote-39) Have you used a string already in this course? Yes, "Hello, World!" is a string containing the characters Hello, World!. We can issue statements with strings in much the same way we did with numbers:

> "Hello, World!";  
//--> "Hello, World!"  
> console.log("Hello, World!");  
//--> Hello, World!  
> "World, I just wanted to say \"Hi!\" 9 times!";  
//--> "World, I just wanted to say \"Hi!\" 9 times!"  
> console.log("World, I just wanted to say \"Hi!\" 9 times!");  
//--> World, I just wanted to say "Hi!" 9 times!

Notice how strings can have double quotes inside of them, as long as you use a \ (backslash) beforehand. What do you type if you want a backslash in the string? Notice also that when we use console.log(), the result does not have surrounding double quotes or the backslashes.

### Boolean

A boolean is a value that is either true or false. They are named after [George Boole](https://en.wikipedia.org/wiki/George_Boole), an English mathematician, and they are astonishingly useful in programming, because they allow programs to make decisions based on values, much like you might make a decision on where to get dinner based on a series of truth values. For example, say you eat burritos every day for dinner except Fridays, when there is a deal on falafel. You ask yourself the question, “Is today Friday?” If the answer is “yes” (true), then you go to the falafel joint. If the answer is “no” (false), you get a burrito.

Since booleans are only ever true or false, you refer to them in JavaScript using those two words:

> true;  
//--> true  
> false;  
//--> false  
> 9 === 9;  
//--> true  
> 9 === "Hello, World!";  
//--> false

Notice that true and false do *not* have double quotes around them. What would happen if they did? I have also introduced you to === in this code snippet. That is the “strict equal” **comparison operator**, and it is a way of asking the console a question.[[4]](#footnote-42) 9 === 9; is the same as “Is the number nine the same thing as the number nine?” Because the answer is “yes,” the console responds true. In the next example, we’re asking the question, “Is the number nine the same thing as the string ‘Hello, World!’?” Since the answer is “no,” the console responds false.

Between numbers, strings, and booleans, we can start using JavaScript as a calculator, but one type remains.

### Null/Undefined

Sometimes we deal with a lack of information, instead of information we understand as a datum. There are two main ways JavaScript understands that lack, through an object called null and a data type called undefined.

null is an object that indicates the lack of information. Say you’re filling out a questionnaire online and you left the last few questions unanswered but still pressed “Submit.” The answered questions would be strings or numbers, probably. But how should the computer understand the unanswered questions? The number zero doesn’t seem right. Nor does a blank string. In this case, it makes sense to register that lack of answers with null.

undefined, on the other hand, is for information that we do not yet have. Using the questionnaire example again, all of the answers are undefined until you press “Submit” and send them to the computer, where it can then decide what to do with them, by turning them into strings, numbers, or just null.

The distinction is tricky, but now you’re definitely ready to start add up some numbers in JavaScript.

## Using JavaScript as a calculator

You learned a comparison operator in the previous section, ===, and you’ll learn a few more in this section. You’ll also learn all five arithmetic operators. In fact, let’s start with them:

> 2 + 3;  
//--> 5  
> 2 - 3;  
//--> -1  
> 2 \* 3;  
//--> 6  
> 2 / 3;  
//--> 0.6666666666666666

As you can see, the arithmetic operators give numbers as answers. The comparison operators, like ===, respond with true or false:

> 2 + 3 \* 6 === (2 + 3) \* 6;  
//--> false  
> 2 + 3 \* 6 < (2 + 3) \* 6;  
//--> true  
> 2 > 2;  
//--> false  
> 2 >= 2;  
//--> true  
> 2 < 2;  
//--> false  
> 2 <= 2;  
//--> true

Programmers have figured out nifty ways to get around the difficulty of typing ≥ and ≤!

Strings also have an operator +. It comes in very handy in web development:

> "I had a thought, but… " + "Oh yeah, I remember. Falafel on Fridays!";  
//--> "I had a thought, but… Oh yeah, I remember. Falafel on Fridays!"

What happens when you add a string to a number? How about a number to a string? Why?

A JavaScript calculator doesn’t sound terribly interesting, so let’s add one more wrinkle to it by introducing variables.

## What if the calculator understood variables?

Everything we have been doing above is fun for about three seconds. It is interesting to test edge cases (what happens when you multiply a number with a boolean?), of course, as that is a good way to understand the assumptions the language is making. But we’re building websites, not calculators.

Nevertheless, getting a bit of flexibility with the console is useful. Let’s expand on that, then, with the let statement, which lets us define variables. Type along in the console.

> let burrito;  
> burrito = "Basically the best food around.";  
> console.log(burrito);  
//--> Basically the best food around.

In the first line, you **defined** a variable, burrito. In the second, you **assigned** to the variable the string, “Basically the best food around.”[[5]](#footnote-46)

Then you tell the console to log the variable burrito, and it logs its value, Basically the best food around.

Any of the data types you have learned about already you can assign to a variable, and we can define multiple variables at once. It’s generally good practice to define all your variables at the top, so you know what you will be working with in the future.

> let magicNumber, secretNumber;  
> magicNumber = 9;  
> secretNumber = 10;  
> secretNumber + magicNumber;  
//--> 19  
> secretNumber === magicNumber;  
//--> false  
> secretNumber > magicNumber;  
//--> true  
> secretNumber = secretNumber + 1;  
//--> 11  
> console.log(secretNumber);  
//--> 11

These variables persist only for the duration of the console. If you close the console, then you’ll lose them. But the variables are also mutable, as you can see. secretNumber starts out assigned to the number 10, but then it becomes assigned to the number 11.

Let’s play a bit more with assigning variables:

> let tipRate, bill, billPlusTip;  
> tipRate = 0.20;  
> bill = 10.00;  
> billPlusTip = bill + (tipRate \* bill);  
> console.log(billPlusTip);  
//--> 12  
> let question, burritos, answer;  
> question = "What is the best food around?\n";  
> burritos = "Delicious burritos";  
> answer = burritos + " are clearly the best!";  
> console.log(question, answer);  
//--> What is the best food around?  
//--> Delicious burritos are clearly the best!

There are two new things in this example: \n can be used to make a new line to make a new line, and now you see that console.log() can take multiple values in the parentheses, separated by commas.

Finally, this discussion of variables allows me to introduce one more operator, the typeof operator. If you get a variable, sometimes you don’t know what kind of data type it is. Yet as we have seen with +, it behaves differently depending on the data. Continuing with the variables in the previous example:

> typeof question;  
//--> "string"  
> typeof magicNumber;  
//--> "number"  
> typeof 2;  
//--> "number"  
> let isItTrue;  
> isItTrue = 1 === 1;  
> typeof isItTrue;  
//--> "boolean"  
> typeof badTastingBurrito;  
//--> "undefined"

As you can see, this example makes use of variables defined in earlier exercises, but it also refers to variable that has not yet been assigned (or even defined).

## Exercises

1. Get console.log() to log a string that includes a backslash.
2. What happens when you surround true or false with double quotes? Are they still booleans?
3. What happens when you add a string to a number? What about the reverse? Why?
4. Use the statements to find billPlusTip above but have the response from console.log() be “You should pay $12 because the service was good.”

## Footnotes

1. More precisely, console is a JavaScript object (similar to window or document, as we’ll see later) that refers to the console you have opened in your browser. .log() is a “method” specific to the console object that outputs whatever is in sent as a parameter (or, inside the ()). In our example, we sent the console a snippet of text, "Hello, World!", and it returned it to us. [↑](#footnote-ref-22)
2. In this course, most of the JavaScript will be procedural, which encourages a step-by-step way of thinking through a solution. Once we move to making maps with Leaflet, the more object-oriented aspects will emerge. JavaScript’s functional personality is its most powerful and appealing, but it strikes me as the most difficult to understand and teach. [↑](#footnote-ref-35)
3. Yes, JavaScript permits using single quotes as well, but you should use double quotes exclusively. [↑](#footnote-ref-39)
4. JavaScript has a less strict equality comparison operator, but its behavior can be unexpected. It is considered one of JavaScript’s bad parts, and I won’t be teaching it in this course. [↑](#footnote-ref-42)
5. let is a statement that is not supported in all JavaScript consoles. If typing the above causes the console to complain Uncaught SyntaxError: Unexpected identifier, then you have one of those older consoles. The solution is, luckily, straightforward. For the duration of this course, where I instruct you to use let, you can use var, instead. [↑](#footnote-ref-46)