Programming Fundamentals

This is a tough section to teach because the needs of the student vary quite widely here. Some of you are programmers from other languages who are coming here to learn front end coding. Some of you are totally new to coding and JavaScript will be your first programming language. Just like you know Spanish already, learning Italian becomes a whole lot easier since you take the same concepts and express them with minor variances. This workshop is optimizing for those totally new to coding and thus we assume no prior knowledge. If you do know another progamming language, I invite you instead to skim this section in familiarize with how JavaScript looks. The next sections will also be aimed at you.

What is code?

A dumb question but its answer may surprise you. Code is for humans first and computers second. You can think of writing code as essentially writing notes on how to solve a particular problem that just happens to be in a way that computer can understand it.

Wait, why? The why is because you or someone else will have to go back and re-read that code some time in the future, maybe tomorrow or maybe in ten years (I've worked on code older than 10 years old, it's not fun.) As such, it's important to write code in a such way that it's easily understandable to someone with little context. You will spend far longer maintaining this code than you will writing it the first time. Be explicit. Be deliberate. The point here is not to be clever but to be simple and to communicate clearly. Code is communication.

Okay, given this, let's frame how this code works then. When you write code, the computer breaks it down into smaller pieces it can understand and then executes those one bit at a time. With JavaScript, only one thing is ever happening at a time (this is called being *single threaded* but that is not a term you need to know.) In general, this means it executes line 1, then line 2, then line 3, etc. Let's that in action:



1

const monthlyRent = 500;

2

​

3

const yearlyRent = monthlyRent \* 12;

4

console.log(yearlyRent);



1

6000

2

undefined

The first thing that happens above is that we declare a variable, monthlyRent. The const keyword is how we let JavaScript know we're declaring a variable. Variable names always have to know no spaces in them, which is why we squish the words "monthly rent" together. In order to make this more readable, we use what's called *camel casing*, so-called because the capital letters in the middle make it look like humps on a camel. You can also use other styles of captialization, there's no rule it must be camel case in JavaScript; everyone just happens to do camel casing in JavaScript.

Notice the ; at the end of every line. This semi colon lets JavaScript know you've completed your thought. Think of this as the period/full-stop of the the programming world.

Variable can be called *almost* anything. You **can't** use keywords. An example would be const const = 15. const is a keyword so it can't be used as a variable name. You do want to give your variables good names. Imagine we have a huge file and 200 lines below we see the variable named monthlyRent: we'll know instantly what this variable does and we won't have to go try to read the code to figure it out. Always, always, always use good variable names. Seriously. Put time into it. Naming things is hard and it's a big part of your job.

Okay, so after line one, I have a variable named monthlyRent that I can use as much as I want. In this case, it represents the number of 500 but it also semantically represents monthlyRent. Imagine if I had 1000 lines between where monthlyRent is declared and where yearlyRent is calculated. I could have just put 500 directly in yearlyRent but I don't because I now understand how that's calculated just by reading the code. Use variables. Use them everywhere. It makes your code way easier to read. Also, later, if my monthly rent changes, I can change it one place and everywhere I reference monthlyRent gets updated automatically. Powerful stuff.

Okay, I think calculate yearlyRent. I use the \* to represent multiplication. I'm also mixing variables and numbers which is just fine. I also could have said const yearlyRent = monthlyRent \* monthsInAYear; (assuming I put const monthsInAYear = 12; somewhere else) too and that would be a good idea. I would argue the two are roughly the same since it's obvious there are 12 months in a year. But you do what you think is most clear. That's your job.

console.log(yearlyRent); is going to print whatever is stored in yearlyRent to the JavaScript console. The JavaScript Console is a part of the dev tools. If you need help finding them, [see here](https://webmasters.stackexchange.com/questions/8525/how-do-i-open-the-javascript-console-in-different-browsers). We'll explain how it works in a bit but for now just know that anything you put between the parenthesises gets logged out to your JavaScript console.

Let's get this little snippet working in our browser. Make a new folder (I'll just a put it on my desktop) and add an index.html file with the following in it:

<!DOCTYPE html>

<html lang="en">

<head>

<title>JavaScript Experiments</title>

</head>

<body>

<h1>JavaScript Experiments!</h1>

<script src="./experiments.js"></script>

</body>

</html>

That script tag is going to let us load JavaScript code into out HTML page. So make another file in the same folder called experiments.js (it really can be called anything as long as the script tag matches it.) Then in the that JS file put our code from above:



1

const monthlyRent = 500;

2

​

3

const yearlyRent = monthlyRent \* 12;

4

console.log(yearlyRent);



1

6000

2

undefined

Now, if you open your **HTML** file, not the JS file, in your browser and open your console, you should see the number 6000 being printed. Congrats! You just wrote your first code!

Numbers, Strings, and Booleans

So far we've just dealt with numbers. Let's go further and start working with words and characters. In programming, we refer to these things are strings, as in a string of one-letter characters. An example of this would be



1

const myName = "Brian Holt";

2

console.log(myName);



1

"Brian Holt"

2

undefined

You can see I use the " (double quote) to enclose everything I want to be in the string. In JavaScript you can also use ' (single quote) and ` (back tick) as well to demarcate strings.

Strings, as you may imagine, are everywhere in programming. We're constantly keeping track of names, addresses, names of products, cities, etc. and thus constantly need strings.

Let's go further. Strings let you connect them together through string concatenation. If I want to be able to greet someone based on their name, I might have something like this:



1

const firstName = "Brian";

2

const lastName = "Holt";

3

​

4

const sentence = "Hello " + firstName + " " + lastName + "! How are you!?";

5

const sentenceWithTemplate = `Hello ${firstName} ${lastName}! How are you!?`;

6

​

7

console.log(sentence);

8

console.log(sentenceWithTemplate);



1

"Hello Brian Holt! How are you!?"

2

"Hello Brian Holt! How are you!?"

3

undefined

The first way is the old way. We can use the + to tell JavaScript to connect two strings. Notice how we have insert the space between firstName and lastName. The computer only does exactly what you tell it to do. If you don't insert that space, it doesn't get put there.

The second line is the new way of doing this. JavaScript got a large update in 2015 and it made things a lot easier. Now you can use the back tick (notice the first uses a double quote, you must use back ticks to do template strings) to do template strings. If you do that, anything inside of ${yourVariableHere} gets output in the string. Cool, right?

Booleans

Sometimes you just need a simple true or false. These are where booleans are useful. Something like a light switch's state is best represented by a boolean. A light is either on (true) or off (false). You'd have something like const lightIsOn = true;. Useful and you'll see them everywhere.

Number

Some languages separate integers (whole numbers, like 1, 2, 3, 4, 500, 1000) and floats (1.2, 3.14159, 14.01, etc.) differently but not JavaScript. JavaScript just has one type of number, Number. A number is a number.

Control Flow

Sometimes I want to modify the flow of how my program works, or in other words, some time I only want to run code if some condition is true. This is where if statements are very useful. Imagine if we tried this.



1

const skyIsBlue = true;

2

​

3

if (skyIsBlue) {

4

 console.log("The sky is blue!");

5

} else {

6

 console.log("The sky is … not blue?");

7

}



1

"The sky is blue!"

2

undefined

In the above example, the condition inside of the parens is evaluated and if it's true, the first block is run and the second is skipped. If it is false, the second block is run and the first block is skipped. Paste that code into your experiments and play with it. You also do not have to have an else block. Okay, let's go further.



1

if (2 + 2 === 4) {

2

 console.log(

3

   "Oh thank god, the fundamental principles of mathematics still hold true."

4

);

5

} else {

6

 console.log("Uh, panic?");

7

}



1

"Oh thank god, the fundamental principles of mathematics still hold true."

2

undefined

You can put any expression (a technical terms, means anything you can stick on the right side of an equal sign, we'll explore it more as we go) inside of the if statement. In this case, we are asking, is two plus two still equal to four. If this is true (I hope so) then again the first block will be run. If not, the second will be.

Let's talk about === for a second. If you use just one = in JavaScript, it means **is assigned to**. So when we have const isBrianCool = true; you can verbalize that as "The variable isBrianCool is assigned to true". Thus we can't use that inside of the if statement because that's not we mean. We're trying to ask a question, not assign something. We're trying to ask "is two plus two equal to four." Enter the triple equals. Triple equals is the same as asking "is this equal to that." We use the triple equals instead of the double equals because double equals does a lot of funny business that usually we don't want it to do. It does what's called coercion and we'll talk about that below. But in an example 2 == "2" but it does not 2 === "2". String 2 is double equal to number 2 but string 2 is not triple equal to number 2.

There's also !==. This is asking "is this not equal to that". Lastly you can ask with numbers > >= < <= as well to ask if numbers less than or greater than too. For another example:



1

const friendsAtYourParty = 10;

2

​

3

if (friendsAtYourParty === 0) {

4

 console.log("Cool, now I have a lot of nachos to myself.");

5

} else if (friendsAtYourParty >= 4) {

6

 console.log("Perfect amount to play some Mario Kart.");

7

} else {

8

 console.log("Wooooo turn on the dance music!");

9

}



1

"Perfect amount to play some Mario Kart."

2

undefined

This also demonstrates the else if if you have more than just two different conditions.

Loops

Okay so now what if I want do one thing multiple times? I could do something like this



1

let friendsAtYourParty = 0;

2

friendsAtYourParty = friendsAtYourParty + 1;

3

friendsAtYourParty = friendsAtYourParty + 1;

4

friendsAtYourParty = friendsAtYourParty + 1;

5

friendsAtYourParty = friendsAtYourParty + 1;

6

friendsAtYourParty = friendsAtYourParty + 1;

7

friendsAtYourParty = friendsAtYourParty + 1;

8

friendsAtYourParty = friendsAtYourParty + 1;

9

friendsAtYourParty = friendsAtYourParty + 1;

10

friendsAtYourParty = friendsAtYourParty + 1;

11

friendsAtYourParty = friendsAtYourParty + 1;

12

console.log(friendsAtYourParty);



1

10

2

undefined

That's annoying though. I wish there was a better way. Before we explore that, let's chat about this example a tad more.

We used let instead of const. Things that are const cannot be reassigned later. In general I find this be of minor help but others do not so I leave you to make your own judgement call. In general one should try to follow the "principle of least power." You should always choose the least powerful "thing" to accomplish whatever you're trying to do. Things with less power tend to be simpler and simple things are less prone to having or causing bugs. We instead use let here because you can see on the subsequent lines we do reassign friendsAtYourParty to be a different number. If you used const your code would crash because const won't let you do that. Thus here we use let. There's another one called var that is the old way of doing JavaScript. There are differences but I don't see a reason to use var at all anymore. It behaves more similar to let.

Okay, so, we want to do this better, let's explore a few ways to do that using loops.



1

let friendsAtYourParty = 0;

2

while (friendsAtYourParty < 10) {

3

 friendsAtYourParty = friendsAtYourParty + 1;

4

}

5

console.log(friendsAtYourParty);



1

10

2

undefined

This is a while loop. The first part works similar to an if statement: as long as what's inside that statement is **true** it will continue running and re-running the body (what's between the { }) until that statement is false. Once that statement is false, it'll break the loop and continue on. This case, we add 1 to friendsAtYourParty until it's 10, and then the next loop, when it's 10, it'll stop because 10 is not less than 10.

Also, let's just show you a few shortcuts for adding one to a thing



1

let friendsAtYourParty = 0;

2

friendsAtYourParty = friendsAtYourParty + 1;

3

friendsAtYourParty += 1;

4

friendsAtYourParty++;

5

++friendsAtYourParty;

6

console.log(friendsAtYourParty);



1

4

2

undefined

Those four lines are equivalent. They all do the exact same thing: they add one to the existing total. The second one, the plus-equals line, you can put any number there and it'll add that amount to total, so friendsAtYourParty += 15; would add 15 to the total. It also works with -= (subtraction,) as well as \*= (multiplication,) /= (division,) and \*\*= (exponent.) Two last two lines (++ before or after) just signify add one. They more-or-less mean the same thing (there's a subtle different of *when* it adds one that should never matter to you) but suffice to say everyone in the JavaScript community *always* does the ++ after; I've never seen anyone do it before in JavaScript.

Okay, so now let's see a second kind of loop to achieve the same effect as above.



1

let friendsAtYourParty = 0;

2

for (let i = 0; i <= 10; i++) {

3

 friendsAtYourParty++;

4

}

5

console.log(friendsAtYourParty);



1

11

2

undefined

This is a for loop which is likely the most common kind of loop. Inside the parens are three statements and you need all of them. The let i = 0; is you defining your control variable that will control the loop. For some reason people always use i, not sure why. It's just that way. It really could be anything. The second statement i <= 10 is just like the while loop, is that's the statement that is as soon as it's false it breaks the loop. The last statement, i++ is that happens at the end of every loop. In our case, we increment the control variable i so that it creeps closer to the end of the loop each time.

An important note: in coding, we start counting from 0. In English, we count 1, 2, 3, 4, 5, etc. but in coding we count 0, 1, 2, 3, 4, etc.. So the fifth element of a string is index 4 (where index is how we'd refer to where that item is in the string). Index 0 is the first element. It's weird but you get used to it and it makes a lot of things easier.

Sometimes, if you mess up what's inside the control condition for the loop, you'll get a runaway loop that'll never complete. This is called an **infinite loop** and it'll lock up and crash your code. Something like this:

let friendsAtYourParty = 1;

while (friendsAtYourParty > 0) {

friendsAtYourParty = friendsAtYourParty + 1;

}

console.log(friendsAtYourParty);

Since you're adding one to friendsAtYourParty each time, and the loop will continue each time until it's less than zero, that condition will never happen. Thus it'll continue going until it crashes your code. Be careful of these. Nasty bugs.