# The Ultimate JavaScript Master Series

# Part 1

## 01 - Getting Started - 01 Welcome

This will teach the fundamentals of programming and Javascript.

## 01 - Getting Started - 02 What is JavaScript?

What is JavaScript

What can you do with it?

Where does JavaScript code run?

What’s the difference between JavaScript vs ECMAScript?

What is JavaScript?

It is one of the most popular and widely used programming languages in the world. It is growing faster that any other programming languages. (Ah… Java and JavaScript are not the same. Der).



Big companies like Netflix, Walmart, and Paypal build Entire applications around JavaScript. Average salary is $72,000 per year. (According to glassdoor.com?)

You can work as a front end developer, or a back end developer, or a full-stack Developer.

What can you Do with JavaScript? For a long time, javascript was only used in browsers to build interactive webpages. “Some developers refer to javascript as a toy language. But those days are gone because of huge community support and investments by large companies like facebook and google.”

You can build full blow web or mobile apps, as well as real-time networking apps like chats and video streaming services, command-line tools, or even games.

Where does JavaScript Code run? It was originally designed to run only in browsers. Every browser has a “JavaScript Engine” that can execute javascript code. E.g., the javascript engines in firefox and chrome are Spidermonkey and v8. In 2009 an engineer named Ryan daul, took the opensource JavaScript Engine in chrome, and embedded it inside a C++ program. He called the program Node.

Node is a C++ program that includes googles V8 JavaScript engine. Using this, we can run JavaScript code Outside a browser. We can pass our JavaScript code to Node for execution. This means, with JavaScript, we can build the back end for our web and mobile applications.

Javascript code can be run inside a browser or in node. They both provide a ‘runtime?’ environment for our javascript code.

What is the difference between JavaScript and ECMAScript?

ECMAScript is just a Specification. JavaScript is a programming Language that Conforms to this specification.

We have a programming language called ECMA, which is responsible for defining standards. They take care of this ECMAScript Specification.

The first version of ECMAScript was released in 1997. In 2015, ECMA has been working on annual releases of a newer specification. E.g., in 2015, they release ES2015/ES6. This specification defined Many new features for JavaScript.

Every browser has a JS engine, and we can write code here without an other tools. Let’s inspect a chrome window.

Select the Console Tab. This is our JavaScript Console. “We can write any Valid JS code here”.

*console.log('Hello World');*

*console.log('Hello World');*

*VM451:1 Hello World*

*undefined*

We can see the hello world Message on the console. (The VM451:1 is an artifact).

We can also write mathematical expressions here.

*2+2*

*2 +2*

*4*

alert(‘yo’)



## 01 - Getting Started - 03 - Setting Up the Development Environment

VSCode, Sublime Text, and Atom are all code Editors. Mosh prefers VSCode.

We will also install Node, from Nodejs.org. You don’t Need Node to execute JS, because you can execute it inside a browser. But it’s good to have node, because you can use it to install Third Party Libraries.

Create a folder, e.g., js-basics, and drag and drop in VSCode.

Let’s add a new file:



“Now you don’t really need to know html in order to take this course, but if you want to be a front end developer, you should know your html well.”

Make a boilerplate doc !. We’ll use this as a host for our JS code. Save.

Open index.html with live server.

In order to write JS, we need a script element. There are two places where we can add this. In the head section, or the body section.

The best practice is to put the script element at the end of the body section, After all the ‘existing?’ elements.

Why is this a best practice? One reason is that the browser parses this file from top to bottom. If you put the script element in the head, there would be a lot of JS code there, and your browser may get busy parsing and executing that JS code and it won’t be able to render the Content of the page. This will create a bad user experience. The user sees a white or blank webpage while your browser is busy parsing and executing your javascript code.

The second reason is that almost always the code between script elements needs to talk to the elements on this web page. For example we may wish to show or hide some elements. So by adding the code here at the end of the body section we’ll be confident that all these elements will be rendered by the browser.

There are exceptions to this rule. Sometimes you are using third party code that has to be placed in the head section. But these are exceptions. As a best practice, you should add your JavaScript code at the end of the body section.

Let’s write some code:

 <body>

    <h1>Hello World</h1>

    <script>

console.log('Hello World');

    </script>

  </body>

The highlighted code is a statement. A statement is a piece of code that expresses an action to be carried out. In this case, we want to log *log* a message *‘Hello World’*, on the *console*.

All statements in JavaScript should be terminated by a semicolon ;

console.log('Hello World');

What we have here in between single “code?” is called a string. A string is a sequence of characters.

In JavaScript we also have this notation: // We can add two slashes and go to sleep this represents a comment.

<script>

        // Comment

        console.log('Hello World');

    </script>

Here we can add some description to our code and this description is ignored by the JavaScript engine. It is not executed. It is purely for documenting the code when you want to explain to other developers why you have written the code this way. You don’t want to explain what the code does because that should be clear in the code itself. (Mosh highlights the console statement). We want to explain Why’s and Hows.

  <h1>Hello World</h1>

    <script>

      // This is my first JS code.

      console.log("Hello World");

    </script>

Let’s open our console again the browser. alt ctrl i

## 01 - Getting Started - 05 - Separation of Concerns

In real world applications we have thousands or even millions of lines of code. Therefore writing in the script element is oftentimes not practical. We don’t want to write all the code inline here. We want to extract and separate our JS code from our html code.

Mosh utilizes a metaphor in which we recognize that bedrooms stores your bed and your clothes. We don’t store are clothes in the kitchen. This is what we call the Separation of Concerns. We want to separate HTML which is all about content from JavaScript which is all about behavior. How should your webpage behave? What should happen when we hover our mouse over a given element? Maybe something should pop up or be hidden. We’ll use JavaScript to implement behavior.

In VSCode let’s start a new file will call index.js. Let’s cut our JS code from our html file, and paste it in index.js. In this application we have a single file, a single JavaScript file. In a real world application we have hundreds or even thousands of JavaScript files. We’ll eventually learn how to combined these files into a bundle and ‘serve’ that bundle to a ‘client’.

Now that all are JavaScript code is in a separate file, we need to reference that file here.

Let’s add an attribute here (in our HTML document):

<script></script>

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

This tells the browser that are JavaScript code is in index.js.

When we open our browser, we note that the Hello World message is still up, which indicates that our code is still working. Huzzah!

## 01 - Getting Started - 06 - JavaScript in Node

We navigate to our js-basics folder. Then we type (in command prompt… not in node?)

*node index.js*

*C:\Users\Mr. Artifice\Desktop\js-basics>node index.js*

*Hello World*

We get the same messages on the consol. As we can see, node is a program that includes google’s V8 JavaScript engine. We can give it a piece of JavaScript code and it will execute that code for us just like we can in a browser. (This works in both command prompt and the node command prompt). So, node is a runtime environment for executing JavaScript code.

Here’s a tip from Mosh:

VSCode includes an integrated terminal, so you don’t have to open up a separate terminal window. Under view, you’ll find the “Terminal” option. Note that our terminal is pointing to the same folder where we created our files… You don’t have to explicitly navigate to this folder. (Make sure you have the index selected).

Here you can type node index.js and get the same output. (I have a big error message from my previous shell manipulations).



In this course we will no longer work with node. Node is a separate topic for which mosh has devoted a course with 14 hours of content.

## 02 - Basics - 01 - Variables - 5.36

Variables are one of the most fundamental concepts in JavaScript or any other programming language.

In programming we use a variable to store data temporarily in a computer’s memory.

We store our data somewhere, and give that memory location a name:

Memory

|  |  |  |  |
| --- | --- | --- | --- |
| Variable |  |  |  |

Variable Name.

With this Name, we can read the data at the given location in the future.

Metaphor!!!

Imagine putting items in various boxes, and labeling each box. Now, you can readily find your stuff. A variable is like this box. What we put inside this box is the value we assign to a variable; that’s the data. And the label that we put on the box is the Name of our Variable.

In index.js let’s declare a variable. In the old days before ES6 we used the *var* keyword to declare a variable.



However, there are issues with VAR as we will discover later in the course.

After ES6, the best practice is to use the *let* keyword to declare a variable.

let

Let’s give this variable and name or an identifier. This is like the label we put on a box. We’ll call it *name* and terminate it with a semicolon.

let name;

Let’s laud this on the console and see what happens.

*console.log(name);*

let name;

console.log(name);

In the console, we see *undefined.* (I also have loads of error messages, for whatever reason).



d

By default variables that we define in JavaScript… Their value is undefined.

We can optionally initialize this variable:

let name;

console.log(name);

= a string, which is a sequence of characters.

let name = 'Mosh';

console.log(name);

We can use single or double quotes. It is more common to use single quotes for declaring strings in JavaScript.



Now we see Mosh on the console.

let name = 'Mosh';

console.log(name);

We have declared a variable called *name* and we have set it to this (‘Mosh’) value, to this string.

There are a few rules for naming the variables.

They cannot be a reserved keyword. For example we can’t use the keyword *let*. If we tried to use one of these names, we will get an error.

Note this red underline:



This is indicating that this is not a valid identifier.

Our second rule is that the name should be meaningful. The name should give some clue as to what the purpose of the variables are. What kind of data are we storing at that memory location. Always use meaningful and descriptive names.

The third rule is that they cannot start with a number.

The fourth rule is that they cannot contain a space or a hyphen.

For example, *let firstName .* Mosh is using camel notation, so the first letter of the first word is lowercase, and the first letter of every word after should be uppercase.

Camel notation is the convention used in JavaScript to name are variables.

The fifth rule for variable names is that they are case sensitive. E.g., the following variables are different:

*let firstName;*

*let FirstName;*

The sixth rule is that if you want to declare multiple variables there are two ways to do this. You can declare them on one line and separate them using a comma… e.g.,

*let firstName, lastName;* (In this case Mosh has not initialized either of these variables. They are both undefined).

We can optionally initialize one or both of them. E.g.,

*let firstName = ‘Mosh’, lastName;* (lastName is undefined) or… Up

*let firstName =’Mosh’, lastName = ‘Hamedani’;*

But the modern, best practice is to declare each variable on a single line. Like so:

*let firstName =’Mosh’;*

*let lastName = ‘Hamedani’;*

## 02 - Basics - 02 - Constants

Let’s make a variable called interest rate:

*let interestRate = 0.3.;*

This is the initial value; we can always change it later.

let interestRate = 0.3;

interestRate = 1;

If we log this on the console, we’ll see the new value… right?



There it is. 1 on the console.

However, there are real world situations in which we don’t want the value of that variable to change… Because otherwise it’s going to cause all kinds of bugs in the application. So, we use a Constant instead of a variable.

The value of the variable changes, but the value of a constant.

So, let’s change *let*, to *const*:

const interestRate = 0.3;

interestRate = 1;

console.log(interestRate);

When we save these changes, we’ll see an error in the console on line two.



If we click the error circled above:



We can see the line in code where this error occurred. So… we cannot reassign a constant.

The best practice is that if you don’t need to reassign, constant is your best choice. If you need to reassign a variable, use let.

## 02 - Basics - 03 - Primitive Types

What are the kind of values we can and assigned to a variable? We have seen strings… but we have more types.

In JavaScript we have two categories of types:

1. Primitives AKA Value Types
2. Reference Types.

For Primitives, we have:

* Strings
* Number
* Boolean
* undefined
* null

E.g.,

let name = 'Mosh';

We have a variable called name, which is set to a string. This: “*‘Mosh’*” is what we call a String literal. This is a fancy name for a string.

Let’s declare a variable and set it to a number.

let age = 30;

This is what we call a number literal.

Let’s declare a Boolean. A Boolean can be either true or false.

Let is approved to be true:

let isApproved = true;

We use Boolean in situations where we want to have some logic. E.g., If the order is approved, it needs to be shipped. So, the value of Boolean variable can be true or false. (Both true and false are reserved keywords, so they cannot be variable names.)

(Are undefined variables ones that aren’t initialized? What is initialized?).

Ah, yes. If we do not initialize a variable, then by default it is undefined.

let firstName;

We can also explicitly set the variable to undefined:

let firstName = undefined;

However, that is not very common. In contrast, we have another key word, *null*.

let lastNames = null;

We use *null* in all situations that we want to Explicitly clear the value of a variable. E.g., We might want to present the user with a list of colors. If the user has no selection, we want to set the:

*selectedColor* variable to null:

let selectedColor = null;

In the future… if the user selects a color, we will reassign this variable to a color like ‘red’.

let selectedColor = 'red'

But then, if the user clicks red again, we want to remove the selection and set this back to null. We use null in situations where we want to clear the value of a variable.

These are the examples of primitives/value types.

let name = 'Mosh';  //This is a String Literal

let age = 30;  //Number Literal

let isApproved = true;  // Boolean Literal

let firstNames = undefined;  //undefined

let lastNames = null; //null

## 02 - Basics - 04 - Dynamic Typing

Something that separates JavaScript from other programming languages is that java script is a dynamic the language.

There are two types of languages:

* Static (statically-typed)
* Dynamic (Dynamically-typed).

In static languages, when we declare a variable, the Type of that variable is set and it cannot be changed in the future: *string name = ‘John’;*

Whereas in a dynamic language, the type of a variable can change at runtime: *let name = ‘John’;*

Let’s examine our code:

let name = 'Mosh';  //This is a String Literal

let age = 30;  //Number Literal

let isApproved = true;  // Boolean Literal

let firstName = undefined;  //undefined

let lastNames = null; //null

At the top, we have declared (let?) this name variable, and we have “set that to a string”. So, the type of name is currently a string, but it can change in the future.

Let’s go to our console, and execute some JavaScript code. We have a typeof operator, which we can use to check the type of variable.

So, we type *typeof* followed by our name variable *name*

*typeof name*



If we reassign name to a different value, like a number (*name = 1;*) and check it’s type:



The type is now changed to number. *‘number’*

This is what we call a dynamic language. Unlike static languages, the type of these variables will be determined at run time, based on the values we assign to them.

Let’s look at some more examples of the typeof operator. (typeof using other reserve key word).

(Command^^) To clear the console press ctrl L.

*typeof age*

‘number’

Type of age is a number.

Let’s change age to a floating point number:

*age = 30.1*

*30.1*

Note that when we look at typeof, age is still a number.



In JavaScript, unlike other programming languages, we don’t have two types of numbers: we don’t have floating point numbers and integers. All numbers are *oftype* number.

*typeof isApproved*

*‘boolean’*

*typeof firstName*

*‘undefined’*

“That’s funny, because the value of this variable (highlights the word undefined) is undefined, but its type is Also undefined”:

let firstName = undefined;

[Now it seems to me that the typeof is *not* undefined. Surely this is part of the following list of types?   
  
**JavaScript types**

* Boolean type.
* Null type.
* Undefined type.
* Number type.
* BigInt type.
* String type.
* Symbol type.

So then, why is this type undefined?]

“What does this mean? Well, earlier I told you that we have two categories of types. Primitives/Value Types, and Reference Types.

In the Primitive/Value Types we have:

* String
* Number
* Boolean
* undefined
* null

So… undefined is actually a Type… but it is also a Value.

let firstName = undefined;

“In this example, *because* we have set *firstName* to *undefined* as a value, it’s type is also undefined.”

How about *selectedColor*?

let selectedColor = null;

key Variable type

word

*typeof selectedColor*

*'object'*

The type of this Variable, is an Object.

## 02 - Basics - 05 - Objects

Reference Types

* Object
* Array
* Functions

An object in JavaScript and other programming languages is like an object in real life. For example, a person has a name, age, address, etc. These are the Properties of a person. The same concept exists and JavaScript.

When we’re dealing with multiple related variables, we can put these variables inside an object.

E.g., here we have two variables: name and age.

let name = 'Mosh';

let age = 30;

They are highly related; they are part of the representation of a person. So instead of declaring two variables, we can declare a person Object. Then instead of referencing these two variables, we can simply reference the person object. It makes for cleaner code.

*let person =* [an object literal] {};

The curly braces above are what we call an object literal.

let name = 'Mosh';

let age = 30;

let person = {

};

Between these curly braces, we add one or more key value pairs. The keys are what we call the properties of this object. In this case, we want the person object to have two properties, or two keys: name and age.

name: [after that, we set the value] ‘Mosh’ [add a comma], [another key value pair] age: 30

let name = 'Mosh';

let age = 30;

let person = {

    name: 'Mosh',

    age: 30

};

  name: 'Mosh',

Key Value who

Now, let’s log person on the console.

*console.log(person);*

console.log(person);

]



We see our person object above… “note the object literal syntax”. [He means the curly braces]. Between them we have a couple key value pairs. Age and Name are the Properties of the person object. 30 and Mosh are the Values of those properties… I think.

*age:30 Name: ‘mosh’*

Property:value Property: Value

There are two ways to work with these properties. Let’s say we want to change the name of this person. We’ll need to access the name property. One way is to utilize Dot Notation. By typing *person[dot]*., the properties are displayed.

It shows the age and the name properties.



Once we see the property that we wish to alter, we can enter it and do the following:

//Dot Notation

person.name = 'John';

We can also use the dot notation to read the value of a property. [In the console I think he means. we’ll read it in the console]

We change:

person.name = 'John';

console.log(person);

person.name = 'John';

console.log(person.name);

Now, in the console, it just says “John”.

Bracket Notation is the other way to access a property. We use square brackets [] instead of .

person [pass a string that determines the name of the target property]

The name of our target property will be name

person [‘name’]

person [‘name’] = ‘Mary’;



Which notation is better? Dot notation or bracket notation?

Dot notation is more concise, so that should be your default choice.

However, bracket notation has its own uses. Sometimes you don’t know the name of the target property until the runtime.

[[So… person is the object, with a couple of properties. Dot notation allows us to select one (or more?) of those properties, and display (probably) or alter it]]

/\* let name = 'Mosh';

let age = 30;

The Object below allows us to eliminate the two variables above

let person = {

    name: 'Mosh',

    age: 30

};

then add

console.log(person);

Now both of the person object properties display in the console.

//Dot Notation

person.name = 'John';

console.log(person);

This displays just John in the Console window, as we have defined the person object (displayed by the console) as merely the name property (using dot notation), which we also redefine.

Next, we also employ bracket notation to define the properties that will be displayed by the console, and its altered values.

let person = {

    name: 'Mosh',

    age: 30

};

//Bracket Notation

person ['name'] = 'Mary';

console.log(person.name);

So... the object is person.

let person = {

    name: 'Mosh',

    age: 30};

Object has a name property.

name: 'Mosh',

The the variable Selection is created, with a value of name.

let selection = 'name';

The console will open the person object's name propery.

console.log(person.name);

The person object is defined with bracket notation as selection, with selection being set with the value of Mary.

The object Person = the value Name,

The variable Selection = the value Name,

The object Person = Selection,

Selection = Mary

The Object display's the name Mary in the console.

\*/

let person = {

    name: 'Mosh',

    age: 30

};

//Bracket Notation

let selection = 'name';

person [selection] = 'Mary';

console.log(person.name);

…However, bracket notation has its own uses. Sometimes you don’t know the name of the target property until the runtime. E.g., in our user interface, the user might be selecting the name of the target property. In that case, at the time of writing code, we don’t know what property we are going to access. That is going to be selected at runtime by the user.

So we might have another variable somewhere else like “let selection = ‘name’;”. *Let selection* [[that determines the name of the target Property that the user is selecting]] =’name’ and that [[mosh highlighted “let selection = ‘name’]]can change at runtime.

With this [[person [] = ‘Mary’;]] we can access that property using the bracket notation in a dynamic way:

*person [selection] = ‘Mary’;*

## 02 - Basics - 06 - Arrays - 4.18

Sometimes in your applications you might be dealing with a list of objects. For example, the list of products in a shopping cart. Or, the list of colors the user is selected. We utilize arrays to store such lists.

*let selectedColors*  Note that we aren’t using an indecipherable abbreviation like SC. We have a meaningful name. Let’s initialize this:

*let selectedColors =*

We’ll set this to an empty array. The square brackets are what we call *array literal*. They indicate an empty array.

*let selecteColors = [];*

We can initialize this array and add a couple of items.

let selectedColors = ['red', 'blue'];

Now, let’s log this on the console:

console.log(selectedColors)

let selectedColors = ['red', 'blue'];

console.log(selectedColors)



Note that our array has two elements. Also note that each element has an index, and that determines the position of that element in the array.

To access an element in an array, we use that index…

For example, suppose we want to display the first element in the array. We can use the square brackets, and then specify the index. Observe:

let selectedColors = ['red', 'blue'];

console.log(selectedColors[0])



Now, only the first item in the index displays.

The lengths of our arrays and the type of objects in them are dynamic, they can change. For example, we can add other elements besides red and blue and expand our array.

let selectedColors = ['red', 'blue'];

selectedColors[2] = 'green';

console.log(selectedColors[0])



Now we have an array with three elements. Thus, the length is dynamic. It is changeable. Also, the type of objects in this array is also dynamic. Most programming languages do not support mixing different types of objects in the same array.

Here, we will change one of the elements in the array from a colour to a number.

let selectedColors = ['red', 'blue'];

selectedColors[2] = 1;

console.log(selectedColors)



Now we have two strings, and a number.

So, the objects in the array as well as the size of the array is dynamic.

The technically an array is an object. Just like the personal object we previously defined, the array has a bunch of key value pairs or Properties that we can access using the dot notation.

If we examined the already in the console with the typeof function, this is our output:

*typeof selectedColors*

*‘object’*

Note, that this array is indeed an object.

console.log(selectedColors)

In VS code, we can examine the properties of this array/object using the dot notation. VSCode displays a number of properties to choose from for an array in JavaScript.

“Every time we declare an array using square brackets that array will automatically receive these properties. That we did not explicitly define them… they are simply magically inherited from somewhere else.” (We will learn about this later when we examine prototypes.)



Let’s examine the length property.

let selectedColors = ['red', 'blue'];

selectedColors[2] = '1';

console.log(selectedColors.length);

This property returns the number of items or elements in an array. See below.



An array is a data structure that we used to represent a list of items.

/\*

02 - Basics - 06 - Arrays

let selectedColor = []

The above Square brackets are called a literal array

let selectedColors = ['red', 'blue'];

console.log(selectedColors)

This displays both elements

let selectedColors = ['red', 'blue'];

console.log(selectedColors[0])

This displays the First element.

let selectedColors = ['red', 'blue'];

selectedColors[2] = 'green';

console.log(selectedColors)

This displays a Third element in the array.

\*/

let selectedColors = ['red', 'blue'];

selectedColors[2] = '1';

console.log(selectedColors.length);

## 02 - Basics - 07 - Functions

In the category of reference types we’ve learned about objects and arrays.

A function is a set of statements that performs a task or calculates a value.

Let’s declare function using the *function* key word. Let’s give function name, *greet*. Add parenthesis… that’s part of the syntax for declaring functions. () Then {}. What’s inside the curly braces is referred to as the body of the function. This is where we had all the statements to define some kind of logic in our application.

E.g., the logic in this function should be to display a message on the console.

function greet () {

    console.log('Hello World');

}

Note that we terminate our statement with a ;. But when we are declaring a function, we do not need to add a ; at the end, because we’re not declaring it like a variable like this: *let number = 1;*

function greet () {

    console.log('Hello World');

}; this semicolon is unnecessary.

We can call a function like so:

We add the name of the function *greet* with parenthesis *()* and a semicolon to indicate that this is a statement.

**So, in my terms… the function *greet ()* is all one unit, both the word and the parenthesis. Inside, we execute the log command. So… add you have to do is add a *greet ()* function [[with a semicolon]] and you are executing the function. So… empty parenthesis are part of the function, along with the name.**

function greet () {

    console.log('Hello World');

}

greet();

Our function can have inputs, and these inputs can change how the function behaves.

Let’s say we’d like to display the name of a person instead of hello world. We can add a variable in between parentheses. We refer to this variable as a parameter. ^^

function greet (name) {

    console.log('Hello World');

}

greet();

Thus, our greet function has one parameter called name. “essentially name is like a parameter that is only meaningful inside of this function. So inside of this function we can work with this name variable but it will not be accessible outside of this function”

Name is an input to this function. So, instead of displaying hello world, we can delete world and add a plus to concatenate two strings… e.g., name:

function greet (name) {

    console.log('Hello' + name);

}

greet();

“When calling the greet function, we need to pass a value for the name variable, or name parameter more accurately”.

So, we can pass John between the brackets. We refer to this as an argument.

function ***greet (name)*** {

    console.log('Hello' + name);

}

greet('John');

So, name is a parameter of the greet function. And John is an argument to the greet function. ^^ “That’s one of the things that a lot of programmers don’t know. They don’t know the difference between a parameter and an argument.”

“So a parameter is what we have here at the time of ***declaration***. The argument is the actual value we supply for that parameter.

function greet (name) {

    console.log('Hello' + name);

}

greet('John');



Why does Mosh have a space?



Ah haha. Add a space after hello.

We can reuse this function, with a different input. We can copy the argument line, paste it, and add a name.

function greet (name) {

    console.log('Hello ' + name);

}

greet('John');

greet('Mary');

Now we have two different messages on the console.



A function can have multiple parameters. We can add more parameters by using a comma. E.g., lastName.

function greet (name, lastName) {

    console.log('Hello ' + name);

}

greet('John');

greet('Mary');

Now we can add this new parameter to console.log. Note that we added an extra space in the quotes. Also, note that parameters aren’t put into quotes. Presumably only the strings? Also, we can easily add spaces to strings, but we cannot do that with parameters. We had to add empty quotes to incorporate spaces.

function greet (name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John');

greet('Mary');

“when calling this greet function, we should pass another argument for the last name.” Let’s see what happens if we don’t do this.

((I got a different error message previously because last name was not capitalized. Remember that this stuff is case sensitive))



The default value of variables in JavaScript is undefined which is why hello john and hello Mary are undefined. “so because we did not pass a value for the last name, by default it’s undefined.”

For greet, let’s pass another argument. We’ll add a comma, and a last name in quotes. “we don’t need the second call to the greet function”. We’re deleting the second call to the greet to function, which is Mary. ((apparently values are assigned two parameters in consecutive order))

/\*

function greet () {

    console.log('Hello World');

}

greet();

function greet (name) {

    console.log('Hello World');

}

greet();

Here we concatenated two strings (hello and name) with a plus, and provided an arguement/function Name

function greet (name) {

    console.log('Hello ' + name);

}

greet('John');

Here we copied our function, and put in a different argument.

function greet (name) {

    console.log('Hello ' + name);

}

greet('John');

greet('Mary');

Here we add a last name, parameter, but we did not pass an argument (assign a value) to it, which is why it is (and displays as) literally 'undefined' in the console.

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John');

greet('Mary');

Lastly, below, we have passed another argument, "Smith", which is assigned to the parameter lastName

\*/

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', 'Smith');

## 02 - Basics - 08 - Types of Functions

Mosh disapproves of all the concatenations listed below. But he says we’ll worry about fixing them later.

console.log('Hello ' + name + ' ' + lastName);

The function below is performing a task. Its task is to display something on the console.

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', 'Smith');

Let’s do a function that calculates a value.

Let’s create a function and call it square. This function should take a parameter. We’ll call it number.

function square(number) {

}

We want to calculate the square of a given number. Aka, number \* number.

“we need to return this value to whoever is calling this function. For that we use the return key word” We cannot have another variable called return because it is another reserved key word.

function square(number) {

    return number \* number;

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

function square(number) {

    return number \* number;

greet('John', 'Smith');

Instead of calling to greet function, will call the square function.

Square… and we pass 2.

square(2);

This returns a value. We can use add value to initialize a variable. E.g.:

let number = square(2);

((This reads square of two, not square times two… I think))

Now we can display this on the console.

console.log(number);

function square(number) {

    return number \* number;

}

//We were calling the greet function.  Now we are calling the square function.

let number = square(2);

console.log(number);

Soo... we'll call the function Square.  Square has a number parameter.  We will return number times number.

Our variable says that number will equal square function (N times N), with the number parameter being 2.



We don’t have to declare a second variable, if all we want to do is display the square of two (square(2)) on the console.

let number = square(2);

console.log(number);

To this:

console.log(square(2));

“First it will call this function [[square(2)]], it will get a value, and then pass that value to console.log.”



Mosh says we have Two function calls. Square(2) is one function call, and console.log() is Another function call, because it has a parenthesis. ^^ (Function call being the thing that requests info from a function?)

console.log()

We are calling the log function which is defined somewhere and passing an argument ((between the brackets)). We could pass a simple string like hello: *console.log(‘Hello’);* Or we can pass an expression. That expression can be a call to Another function… “like square of 2”: *console.log(square(2));*

“For now, all I want you to take away is that a function is a set of statements that either performs a task for calculates and returns a value.”  
  
/\*

This function is Performing a task.

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', 'Smith');

function square(number) {

    return number \* number;

}

//We were calling the greet function.  Now we are calling the square function.

let number = square(2);

console.log(number);

Soo... we'll call the function Square.  Square has a number parameter.  We will return number times number.

Our variable says that number will equal square function (N times N), with the number parameter being 2.  There is an order of operations aspect.  The same term can have different values on the page... because sometimes that term is being acted upon, and other... things are receiving that finished output

function square(number) {

    return number \* number;

}

let number = square(2);

console.log(number);

\*/

//Performing a task

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

function square(number) {

    return number \* number;

}

console.log(square(2));

## 03 - Operators - 01 - JavaScript Operators - .40

We utilize operators along with variables and constants to create expressions. With expressions we can implement logic and algorithms.

Kinds of operators:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

## 03 - Operators - Comparison Operators - 2.01

We use comparison operators to compare the value of a variable with something else.

Here we have the x set to 1.

*let x = 1;*

Let’s look at the first comparison operator that is greater than. *console.log(x > 0);*

“So we want to check of X’s greater than zero”

let x = 1;

console.log(x > 0);

The console displays this as true.

“So the result of an expression that includes a comparison operator is a Boolean… It’s true or false”.

console.log(x >= 1);



We also have less than, and less than or equal to:

let x = 1;

console.log(x > 0);

console.log(x >= 1);

console.log(x < 1);

console.log(x <= 1);



We refer to the above operators as relational operators.

We also have equality operators. We can check and see if X is equal to one:

console.log(x === 1);

If we want to see if x is not equal to a given value, replace the equal sign with an exclamation mark.

console.log(x !== 1);



These are all the comparison operators in JavaScript.

let x = 1;

//Relational

console.log(x > 0);

console.log(x >= 1);

console.log(x < 1);

console.log(x <= 1);

//Equality

console.log(x === 1);

console.log(x !== 1);

## 03 - Operators - 05 - Equality Operators

Previously we learned about the equality operator. E.g.,

// Strict Equality Operator (Type + Value)

console.log(1 === 1);

In JavaScript we have another equality operator that is indicated by two equals signs.

// Lose Eqaulity Operator

console.log(1 == 1);

The strict equality operator ensures that the values on either side of this operator have the same type and value.

For example on both sides of this operator there are two numbers. Both their type and their value are equal.

console.log(1 === 1);

Suppose we change one of the numbers to a string, like so:

console.log('1' === 1);

This expression is going to evaluate to false. We are comparing a string to a number. The types do not match.

Whereas if we compare a lose equality operator that features a number and a string like so:

// Lose Eqaulity Operator

console.log(1 == 1);

console.log('1' == 1);

The result is still true.

In this case, the operator ((the two equals signs)) looks at the value on the left side, observes that it is a string, and converts the numeral on the right to a string as well.

What if on the left side we have a Boolean?

console.log(true == 1);

The operator will automatically convert the value on the right side to a Boolean.

The console will also read this as true.

“Here’s what you need to take away: the strict equality operator ensures that both values have the same type and the same value. The lose equality operator does not care about the types matching; if the types don’t match it will convert that type on the right side to match the type on the left side. Then, it will only check if the values are equal”

Generally, you will use the strict equality operator because it is more precise and accurate.

/\*

// Strict Equality Operator (Type + Value)

console.log(1 === 1);

console.log('1' === 1);

// Lose Eqaulity Operator

console.log(1 == 1);

\*/

// Lose Eqaulity Operator

console.log(1 == 1);

console.log('1' == 1);

console.log(true == 1);

## 03 - Operators - 06 - Ternary Operator

Let’s examine the Ternary or conditional operator, which is incidentally one of Mosh’s favorites.

Consider the following problem:

// If a customer has more than 100 points,

// they are a 'gold' customer, otherwise,

// they are a 'silver' customer.

Let’s start by declaring a variable to keep track of the number of points.

let points = 110;

Now let’s declare another variable called type, that will represent the type of customer.

*let type =*

And here’s where we use the Ternary or *conditional* operator. First let’s start with a condition. We want to compare the number of points with 100.

So, we use the comparative operator here:

*let type = points > 100*

This expression produces a Boolean. The result of this expression is either true or false.

Depending on the result, we’ll set the type variable to a different value. Let’s add a question mark:

*let type = points > 100 ?*

If this expression evaluates to true, that means this is a gold customer… So we want to set type to gold.

let points = 110;

let type = points > 100 ? 'gold' :

If the expression evaluates to false up, we want to set the type to:

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

Let’s log type on the console.

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

The console displays gold.

let points = 90;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

Now the console displays silver.

/\*

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

\*/

// If a customer has more than 100 points,

// they are a 'gold' customer, otherwise,

// they are a 'silver' customer.

let points = 90;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

## 03 - Operators - 07 - Logical Operators - 5.30

We use logical operators to make decisions based on multiple conditions. In JavaScript we have three kinds of logical operators:

* Logical and
* Logical or
* Not

We’ll start with Logical AND, which is indicated with two ampersands.

Here is the basic rule of thumb: logical returns TRUE if both operands are TRUE. Let’s see an example.

console.log(true && true)

We have two operands, and they are both true. So, the result of evaluating this expression will be true.

If either of these are false, the result will be false.

console.log(false && true)

The console outputs false. The console output false if one or both are false.

What is a real world to use case for this operator? Suppose we want to build an application for approving loans. We want to see if the applicant has high income and a good credit score.

Let’s declare a couple variables:

let highincome = true;

let goodCreditScore = true;

Here we are dealing with two conditions. We need to know that the applicant is true for both high income and credit score. That’s where we use the logical And.

Let’s declare another variable:

let eligibleForLoan =

This is where we use the logical and operator:

let eligibleForLoan = highIncome && goodCreditScore;

Now it’s long this on the console.

let eligibleForLoan = highIncome && goodCreditScore;

let highIncome = true;

let goodCreditScore = true;

let eligibleForLoan = highIncome && goodCreditScore;

console.log(eligibleForLoan);

The console outputs true.

Now let’s examine the logical or.

Logical OR is indicated by two vertical lines. This returns true if one or both of the operands are TRUE.

Less edit the logical and and turn it into a logical or, and change one of the operands to false.

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

Lastly, let’s look at the NOT operator. The knot operator is indicated by an exclamation mark.

If the applicant is not eligible for loan, we want to consider their application as refused.

Let’s declare another variable:

*let applicationRefused*

Let’s use the NOT operator:

*let applicationRefused = !*

We apply it on eligibleForLoan:

let applicationRefused = !eligibleForLoan;

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log(eligibleForLoan);

If the eligibleForLoan is true, the not operator will convert that to false. So, whatever we give it, it will give us the opposite. So if eligibleForLoan is true, this will be converted to False, which means applicationRefused will Also be false.

In this scenario, applicationRefused is always the opposite of eligibleForLoan. That’s where we use the not operator.

So… let’s try changing both conditions to False:

let highIncome = false;

let goodCreditScore = false;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

//NOT (!)

let applicationRefused = !eligibleForLoan;

And… let’s add a label to console.log:

console.log('Eligible', eligibleForLoan);

((When I test this in the console, it gives the result “Eligible false”, instead of just false.))

Let’s do another console.log, and add another label:

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused')

And we log the applicationRefused variable:

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused', applicationRefused);

let highIncome = false;

let goodCreditScore = false;

let eligibleForLoan = highIncome || goodCreditScore;

console.log('Eligible', eligibleForLoan);

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused', applicationRefused);



/\*

//Logical AND (&&)

//Returns TRUE if both operands are TRUE

console.log(true && true)

We have two operands, and they are both true.  So, the result of evaluating this expression will be true.

console.log(false && true)

The console outputs false.  The console output false if one or both are false.

let highIncome = true;

let goodCreditScore = true;

let eligibleForLoan = highIncome && goodCreditScore;

console.log(eligibleForLoan);

This outputs as true.

// Logical OR (||)

// Returns TRUE if one or both of the operands is TRUE

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log(eligibleForLoan);

This will return the opposite value it receives.  In the above case, the applicant Is eligible for the loan, which means eligibleForLoan returns false, which means applicationRefused is Also false.

\*/

// Logical OR (||)

// Returns TRUE if one or both of the operands is TRUE

let highIncome = false;

let goodCreditScore = false;

let eligibleForLoan = highIncome || goodCreditScore;

console.log('Eligible', eligibleForLoan);

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused', applicationRefused);

/\* Basically, when highIncome and goodCreditScore are false, then eligibleForLOad is false.

And the console.log will display the value, along with a label.

If !eligibleforloan has a false input value, it will flip it to true, and applicationRefused will be true.

Console.log will show the label string, and that applicationRefused is True. \*/

## 03 - Operators - 08 - Logical Operators with Non-booleans - 5.53

In JavaScript unlike many other programming languages we can use logical operators with Non-booleans values.

false or true, aka

false || true

The above is a true expression.

How about:

false || ‘Mosh’



How about:

false || 1

We get an output of 1.

What we realize is that the result of a logical expression is not necessarily true or false. That depends on the value of the operands we have.

In the first example our second operands is true, which is why we get true back.

In our second example, our second Operand is a string, which is why we get a string back.

In the third example our second Operand is a number which is why we get a number back.

So when our JavaScript engine tries to evaluate this expression it looks at each operand. If that operand is not a Boolean true or false, it will try to interpret it as what we call truthy or falsy.

Falsy is not the same as Boolean false.

The values of false are:

* Undefined
* Null
* The number zero 0
* Boolean false
* Empty string ‘’
* Not a number NaN groups

Not a number is a special value in JavaScript. Mathematical calculations that do not produce a valid number, this value, not the number, is returned.

If we use any of these Falsy values in a logical expression, they will be treated as Falsy, which is kinda like a Boolean false, but it is not exactly the same.

Anything that is not Falsy, is truthy.

In our second example, our second operand is a string with four characters. ((‘Mosh’)). It’s not an empty string. It’s not falsely. So it’s truthy. Because with the logical/or operator, the result will be true if One of the operands is true. The first operands is false (false || ‘Mosh’), and so the JavaScript engine examines the second operand, which it finds to be not falsy, and therefore truthy.

The same goes for the third example. (false || 1) The 1 is not a Boolean true… It’s truthy. That’s why the value of that operands is returned.

What if we have a triple expression?

false || 1 || 2

The console outputs 1.

With the logical/or operator, it will return a result as soon as it finds an operand that is truthy. The first truthy operand discovered is the one the console will output. You could have a dozen more operands… it will only return the first truthy operand it encounters. This is called Short-Circuiting.

When do we actually use non-Boolean operators?

Suppose we are building an application, and somewhere the user has to pick a color, or were going to use a default color. E.g., the color of the T-shirt they want to buy.

Let’s declare a few variables:

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

Let’s log this on the console:

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

The console output is “red”. Because, our user has selected a color.

If the user has not selected a color, then you get the obvious result:

let userColor = undefined;

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

Console output is ‘blue’.

“This is the power of using the logical or operator between non-Boolean’s. With this technique we can provide default values.”

/\*

Falsy (false) :

Undefined

null

0

boolean false

empty string ''

Not a Number NaN

Anything that is not Falsy -> Truthy

E.g.:

 false || ‘Mosh’

returns: 'Mosh'

false || 1

We get an output of 1.

false || 1 || 2

The console outputs 1.

With the logical/or operator, it will return a result as soon as it finds an operand that is truthy.

You could have a dozen more operands… it will only return the first truthy operand it encounters.  This is called Short-Circuiting.

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

The output is Red, because user color is selected

\*/

let userColor = undefined;

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

## 03 - Operators - Bitwise Operator

These operators are more on the theoretical side and not something we'd use on a daily basis. This section of the lecture is for the enthusiasts, a.k.a. autists. ((Mosh encourages us to skip this lesson if we like)).

We humans use the decimal system to represent numbers. “12345”. In computers these numbers are stored in the binary format.

E.g., the number one in the decimal system is represented by seven zeros “00000001”. We have eight digits, each digit is what we call a bit. Above we have eight bits which represent one byte of information.

Number two equals six zeros and then one zero. “00000010”.

“There are mathematical formulas behind these for converting numbers between decimal and binary systems”. If you don’t want to learn the mathematics, simply Google decimal to binary converters.

“Bitwise operators in JavaScript or any other programming languages are similar to logical operators but they work on the individual bits of a number.”

E.g.:

console.log(1 | 2);

Note that we have a single vertical line, that represents a bitwise or. ((1 or 2, 1 | 2)) ((Logical ors have double vertical lines)).

“When we apply the bitwise or between one and two, this is what’s going to happen…”

“I add R as the result here (R =). Now, this operator here is going to look at each of these bits in a vertical way. ((Like adding sums?)). If either of these bits is one, the result will be one. Otherwise it will be zero”

1 = 0000001

2 = 0000010

R = 0000011

“If you convert this binary number (00000011, the sum of 0000001 and 00000010) to a decimal number, the result will be three”.

The console displays 3.

Bitwise and is similar. We use a single ampersand instead of a vertical line.

If both numbers are one, one will be returned, otherwise the result will be zero.

1 = 0000001

2 = 0000010

3 = 0000011

R = 0000000

console.log(1 | 2); // Bitwise Or

console.log(1 & 2); // Bitwise And

The console outputs 3 and 0.

Suppose you want to implement an access control system. The user can have the following permissions “Read, Write, Execute”.

We can use one byte of information (or eight bits) to determine the kind permission of a user can have.

E.g., we can use five zeros 00000 (the first 5 bits are irrelevant), for the last three bits, if the user has permission we will use 1, otherwise we will use 0.

E.g., read only permission would look like this: 00000100.

00000100 Read

00000010 Write

00000001 Execute

This is where we use bitwise operators.

Let’s implement an access control system using bitwise operators.

Let’s define a constant called read permission, and set it to a decimal number that is equivalent to a binary number 00000100, i.e. 4.

00000100 Read

00000010 Write

00000001 Execute

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

Let’s declare a variable like my permission and initially set that to 0.

let myPermission = 0;

Now let’s give ourselves extra permissions.

myPermission = myPermission | readPermission | writePermission;

((Ah ha. So, my permission equals 0, **OR** it equals readPermission, aka 4, or writePermission, aka 2.

But… what is the point of adding the numbers together? I don’t get it.))

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

console.log(myPermission);

6 is our output.

“Technically we don’t care about this decimal number. We can use the bitwise and operator to see if I have a given permission.”

For example:

Let’s declare a variable message, and use the ternary operator. The conditional operator.

“We start with a condition”: = **()** ((When working with ternary/conditional operators above, no parenthesis were used))

“We take myPermission”

Let message = (myPermission)

“And use the bitwise & operator”

let message = (myPermission &)

“Along with the readPermission”

let message = (myPermission & readPermission)

“If this evaluates to true” let message = (**myPermission & readPermission**) “that means I have the read permission.”  
  
***((How does any of this evaluate as true or false? If they just aren’t zero?))***

“Let’s say we want to display a message like yes:

let message = (myPermission & readPermission) ? ‘yes’:

“Otherwise, we want to display no”

let message = (myPermission & readPermission) ? ‘yes’: ‘no’;

“Let me break this up into multiple lines so we can see clearly”

let message =

(myPermission & readPermission) ? ‘yes’: ‘no’;

“Here’s our Ternary Operator”.

Highlights left parenthesis?

“We have a condition”

let message =

**(myPermission & readPermission)** ? ‘yes’: ‘no’;

“If that evaluates to true, this value will be used”

let message =

(myPermission & readPermission) ? **‘yes’**: ‘no’;

“Otherwise, this other value will be used”.

let message =

(myPermission & readPermission) ? ‘yes’: **‘no’**;

“Finally, let’s log this message on the console.

Console.log(message);

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

Console outputs: yes

((Interesting. When you remove the yes: no, the console outputs 4. But, when you add that “?” and the yes: no [[gold: silver in the example way above]], it now hinges on the truth of the condition. If it evaluates to true, the first value (yes) will be used. So, perhaps the ? : make it a conditional/ternary… thing))

((If we remove readPermission from the myPermission variable, then the *let message* variable will output as false, and therefore no)).

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

“So here’s what I want you to take away. With the bitwise or operator we can add permissions. And with the bitwise & operator we can check to see if we have a given permission.”

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

“Of course this is just one real world use case for using the bitwise operators. There are other use cases, but as I told you before they are really not that common. So if this lecture was confusing, don’t worry about it.”

/\*

Example of a bitwise or with a single vertical line: console.log(1 | 2);

(Logical ors have double vertical lines ||)

// Bitwise Or  If Either of the bits are 1, the result will be 1.

console.log(1 | 2);

1 = 0000001

2 = 0000010

R = 0000011

// Bitwise And - If both numbers are one, one will be returned.  Otherwise, the result will be zero:

console.log(1 & 2);

1 = 0000001

2 = 0000010

3 = 0000011

R = 0000000

console.log(1 | 2);

console.log(1 & 2);

The console outputs 3 and 0.

These are the fundamentals.

//Read, Write, Exectute  (User features)

//00000 (the first five bits are irrelevant).  The remaining bits indicate that the user has permission if the value is one.

00000100 Read

00000010 Write

00000001

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission =0;

myPermission = myPermission | readPermission | writePermission;

console.log(myPermission);

((console displays 6))

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

Outputs to yes

\*/

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

## 03 - Operators - 10 - Operators Precedence

Complex expressions require you to consider the precedence of the operators.

E.g.,

let x = 2 + 3 \* 4;

Let’s ask the console for the answer:

console.log(x);

14

Of course, the multiplication expression is evaluated before the addition expression.

It is hard to memorize which operators have higher or lower precedence. So when working with complex expressions, you can use parenthesis to determine how these operators are applied. E.g.,

let x = (2 + 3) \* 4;

console.log(x);

Now our output is 20.

/\*

let x = 2 + 3 \* 4;

console.log(x);

The output is 14

let x = (2 + 3) \* 4;

console.log(x);

This output is 20.

\*/

let x = (2 + 3) \* 4;

console.log(x);