# 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’;*