# A Quick JavaScript Introduction

The JavaScript language (JS for short);

* Is cross platform; because browsers are cross platform and the node runtime is cross platform. That makes JS cross platform too
* Is a scripting language
* Is an interpreted language; as opposed to compiled languages like Java, C, C++ or C# are
* Is object oriented; not in the way Java or C# is — JS uses prototypal inheritance, while ES6 introduced the “class keyword”, JS doesn’t have classes in the way that Java, C++ or C# would implement classes
* Is dynamically typed; so you can create variables of type ‘string’ and then later assign a number to it; there’s no good reason why a programmer will do this, but in JS, you certainly can
* Can be ran on the client-side (inside a browser), on the server side (like nodeJS). You can also use JS to write CLI programs (it needs nodeJS for this)
* Is not related to Oracle’s Java; they are two completely different languages
* Is an implementation of the ECMAScrip specification
* Has been written by Brendan Eich sometime in 1995; the lore, if it is to be believed, says he wrote it in 10 days

## How can you run JS

You can use JS any browser that has a JS engine. Safari, Edge and Google Chrome all use webkit and V8 engine while FireFox uses the SpiderMonkey engine. The basic idea is to reference a JS file in the HTML code, like so

<html>

<body>

<script src=”myProgram.js”></script>

</body>

</html>

You can run JS outside the browser by using nodeJS. The basic idea is to invoke the node runtime and pass the name of your JS program to it, like so;

node myProgram.js

## JavaScript REPL

The Read Eval Print Loop (REPL) tool is an interactive way to run JS statements (even blocks, like loops and conditions). It basically reads and evaluates whatever JS codes you write in it and then immediately prints out the results; which makes it a great way to learn JS constructs quickly.

**To use the REPL inside the browser**

Open the Developer Tools. You can do this from the main menu of Chrome then **View** > **Developer > Developer Tools**. Alternatively, you can use keyboard shortcuts like *CTRL+Shift+J* (on Windows) and *Command+Option+J* (on macOS) to view console, as shown in the picture below.

A screenshot of a cell phone

Description automatically generated

To use the **REPL in the command line**, just invoke the node runtime from anywhere you like; as shown in the example below.

**ted in** ~ node

Welcome to Node.js v12.8.1.

Type ".help" for more information.

> console.log("Hello World");

Hello World

undefined

> Math.pow(2,10);

1024

>

To exit the CLI REPL, press CTRL+D or type the following (while in the REPL)

process.exit(0);

## Basic Syntax

### Program entry point

There is no program entry point Unlike many languages, JS doesn’t require a function main. You can basically write any number of (valid) statements, it gets executed, one after another — generally, on the same order as they are defined in the source code.

var name = "John Doe";

var email = "jdoe@gmail.com";

console.log(`name: ${name} | email: ${email}`);

### Grammar and Types

JavaScript belongs to CFOL (C Family of Languages). If you used C, C++, C# or Java before, JS will look very familiar; and like most languages in CFOL, JS is case sensitive. Declarations like in the following example;

var Myvariable = 1;

var myvariable = 1;

Are okay; because myVariable is different from Myvariable. Even if their difference is simply the lowercase and uppercase M, this is enough for JS to consider them as two separate variables — so, be careful on this one.

In JS, instructions are called statements — sometimes you will see them terminated by semicolons (and sometimes you won’t). JS is quite forgiving if you forget the semicolon.

|  |
| --- |
| **NOTE** The only two times you need to use a semicolon on your statements are 1) if you write more than one statement in one line and 2) if you’re team or company requires that you use semicolon, no matter what — you know, part of coding convention |

Generally, you will see English characters in the code, but technically, JS can use any ISO-8859-1 characters; so, statements like the following, is allowed;

var Früh = "foobar";

The umlaut is a legal character to use in JS because it can use unicode characters. Whether this is a good idea to use, is something I’ll leave up to you.

### Comments

Commented lines are ignored by the interpreter, that makes them (comments) good for code documentation. There are two ways to comment your code in JS, here they are;

// a one line comment

/\* this is a longer,

\* multi-line comment

\*/

/\* You can't, however, /\* nest comments \*/ SyntaxError \*/

### Variables

You can declare variables using either one of the 3 keywords *var, let and const*.

var mLastname = “Doe”; // declares a variable and initializes it to a value

var mFirstname; // just declares a variable without initializing it

The *var* keyword is written to the left of variable name; in the above example, mLastname and mFirstname are the names of the variables (they are more formally referred to as *identifiers*).

Types in JavaScript are;

* Number – double precision 64-bit which adheres to IEEE 754 specs. JS doesn’t care if a number is integer or float point; they’re all just *number*
* String – a sequence of unicode characters. String literals are created by surrounding characters or words either in double quotes, single quotes or back ticks
* Boolean – *true* and *false* are boolean literals
* Object
  + Function
  + Array
  + Date
  + RegExp
* Symbol (new in ES6)
* Undefined – any variable that has been declared but not yet defined, will have this data type
* Null

### Functions

Functions in JS can either be named or unnamed (anonymous). In either case, they are a collection of statements which may or may not produce a result when invoked. Example of a named function is as follows;

function sum(a,b) {

return a + b;

}

The name of the function (sum, in our example) comes after the keyword function. This function example takes on 2 parameters and returns a value. There are functions that don’t accept parameters, and there are also some that don’t return any values.

An example of a function that doesn’t take any parameter and does not return any value as well is as follows;

function doSomething() {

console.log(“doing something”);

. . .

}

Functions in JS can also be written as expressions, as such, they are written as follows;

let sum = function add(a,b) {

return a + b;

}

console.log(sum(1,2));

In this example, *sum* is a variable and we assigned a function definition to it. While the name of the function is technically *add(a,b),* that’s not how we should invoke it. We should invoke it by calling *sum(),* because sum is the variable that contains the definition of our function.

There are times when we omit function names, when we do, they are called anonymous functions. Let’s rewrite our sum example, using anonymous functions,.

let sum = function(a,b) {

return a + b;

}

console.log(sum(1,2));

|  |
| --- |
| **NOTE** When the first word in the function block is the word *function*, it’s a function declaration, otherwise, it’s a *function expression* |

When ES6 came out, it gave us a new way to define anonymous functions; we can now use the “fat arrow” operator. Our previous code example can now be rewritten as

let sum = (a,b) => {

return a + b;

}

In fact, we can write it in a much more concise way, like this;

let sum = (a,b) => a + b;

We can omit the curly braces because there is only one statement in the block; and because there isn’t any block, we can also omit the “return” keyword — naturally, if you need to write more than one statement in the function, you need to use to keep the pair of curly braces (the block).

### String

Yo can define string literals by enclosing letters or words in double quotes or single quotes.

let someString = “Hello World”;

let someOtherString = ‘Hello World’;

Either way is fine.

Starting with ES6, we can also use back ticks for creating string literals, like this;

let longString = `<html>

<body>

back ticks are not single quotes, if you are using an American keyboard, the

back tick char is located below the ESC key and to the left of numeral 1

</body>

</html>`;

The back tick is handy if you need to work with strings that spans multiple lines.

Finding things in a string

Split

Length

Trim

Touppercase

Tolowercase

### Math Operators

You can use the **+, -, /, \*** and **%**. They work exactly the way you expect them to. If you haven’t seen the % for a while, it’s the modulo operator; it gets you the remainder while the **/** gets you the quotient.

### Comparisson Operators

JavaScript has both strict and type–converting comparisons. A strict comparison (e.g., ===) is only true if the operands are of the same type and the contents match. The more commonly-used abstract comparison (e.g. ==) converts the operands to the same type before making the comparison. For relational abstract comparisons (e.g., <=), the operands are first converted to primitives, then to the same type, before comparison

Strings are compared based on standard lexicographical ordering, using Unicode values, for example;

console.log(1 == 1);

// expected output: true

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

// expected output: true

Features of comparisons:

* Two strings are strictly equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
* Two numbers are strictly equal when they are numerically equal (have the same number value). Nan is not equal to anything, including NaN. Positive and negative zeros are equal to one another.
* Two Boolean operands are strictly equal if both are true or both are false.
* Two distinct objects are never equal for either strict or abstract comparisons.
* An expression comparing Objects is only true if the operands reference the same Object.
* Null and Undefined Types are strictly equal to themselves and abstractly equal to each other.

The equality operator (==) converts the operands if they are **not of the same type**, then applies strict comparison. If **both operands are objects**, then JavaScript compares internal references which are equal when operands refer to the same object in memory.

1 == 1 // true

'1' == 1 // true

1 == '1' // true

0 == false // true

0 == null // false

var object1 = {'key': 'value'}, object2 = {'key': 'value'};

object1.key == object2.key //true

0 == undefined // false

null == undefined // true

There are other operators that you can use for comparisson like the > < !== !===. There are also logical operators like the && and ||, all of which, I’m (dangerously) assuming, you’ve seen before. If you have, they behave exactly as you think they do.

### Conditions

Like many other languages, JS supports the *if* statement.

if (condition) {  
 // block of code to be executed

// if the condition is true  
}

The condition in the code block needs to evaluate to either true or false. The condition is generally constructed as an expression using equality operators. Generally this block will behave the way you expect it to be; but you need to consider and remember that JS has the notion of “truthiness” and “falsiness”, and these aren’t strictly true or false — we’ll discuss truthiness and falsiness shortly.

An example on how to use if statement is shown below;

let hour = new Date().getHours();

if(hour < 18) {

console.log("Good day");

}

else {

console.log("Good evening");

}