# JavaScript

* JavaScript is a high level, object oriented, multi paradigm programming language.
* High-level: Resources (like memory) are managed by the language itself, resources are not managed manually as in low level languages, as C.
* Garbage-collected: Algorithm inside the JavaScript engine which automatically removes all the unused objects from the computer memory.
* Interpreted or just-in-time compiled: PENDING
* Multi-paradigm: Supports procedural programming, object-oriented programming and functional programming.
* Prototype-based object-oriented: Almost everything in JavaScript are objects, even arrays are objects and those have access to its methods (push, shift, etc.) due to the prototype inheritance.
* First class functions: Functions are treated as variables, we can pass them into other functions as parameters or return them as the output of a function.
* Dynamic: Dynamically typed language. No data definitions, types become known at runtime. Data type of variables are automatically changed on reassignment.
* Single-threaded: JavaScript runs in one single thread, so it can only do one thing at a time.
* Non-blocking event loop: Takes long running tasks, executes them in the “background” and puts them back in the main thread once they are finished.
* Variable names can only contain numbers, letters, underscores, or the dollar sign.
* As a convention, write constant variable names with uppercase, like const PI = 3.14
* == operator (loose) can introduce bugs to the code, it is best to ALWAYS use the === operator (strict) and if type coercion is needed, then do it manually instead of expecting the == operator to do it automatically.
* Expression: produces a value (1 + 2) or (2 < 3)
* Statement: Full sentence (const a = ‘sale’;)
* Ternary operator (? :) is an expression, different to an if else statement, which is a statement.
* JavaScript updates will never break old revisions, NEVER BREAK THE WEB, all updates are backwards compatible.
* Strict mode: Type ‘use strict’; in the first line of the script, helps to prevent issues that can appear if no strict mode is activated, recommended to always use it.
* Functions inside an object are called methods and methods inside an object have access to the ‘this’ keyword, ONLY METHODS. ‘this’ is writable and readable, FYI.
* Arrow functions do not have access to its own ‘this’ keyword, only normal functions ‘function() {}’.
* Arrays are also objects, that is why they have their own methods.
* ‘continue’ advances to the next iteration and ‘break’ gets you out of the loop.
* Besides from ‘console.log’, we also have ‘console.warn’ and ‘console.error’.
* ‘debugger;’ can be used to open the debugger tool in the browser, being that line the breakpoint.
* The CSS ‘box-sizing’ property allows us to include the padding and border in an element's total width and height.
* Document Object Model ‘DOM’: Structured representation of html documents. Allows JavaScript to access html elements and styles to manipulate them.
* Top level code is the code that is NOT inside a function and will be executed in the global execution context.
* One execution context is created per function, and those contexts together with the global execution context make the ‘call stack’.
* Inside an execution context you can find
* Variable environment: let, const and var declarations | Functions | arguments object
* Scope chain: References to variables that are located outside the current function.
* ‘this’ keyword: Special variable PENDING
* The JavaScript engine is formed by the call stack and the memory heap.
* Block scope applies only for variables declared with let or const. Variables declared with var are function scope (or global). Functions are also block scoped (only in strict mode).
* Hoisting: Makes some types of variables accessible/usable in the code before they are actually declared. “Variables lifted to the top of their scope”. Before execution, code is scanned for variable declarations, and for each variable, a new property is created in the variable environment object.
* ‘this’ keyword/variable: Special variable that is created for every execution context (every function). Takes the value of (points to) the “owner” of the function in which the ‘this’ keyword is used.
* ‘this’ is NOT static. It depends on how the function is called, and its value is only assigned when the function is actually called. ‘this’ does NOT point to the function itself and also NOT its variable environment.
* Method: this = <Object that is calling the method>
* Simple function call: this = undefined
* Arrow functions: this = <this of surrounding fuction (lexical this)>
* Event listener: this = <DOM element that the handler is attached to>
* new, call, apply, bind: <PENDING>
* Only primitive values are stored directly in the call stack, that is why a variable declared with const cannot be changed. On the other hand, objects and functions are stored in the memory heap and only a reference to that value is stored in the call stack, that is why if you declare a variable to be equal to another object variable, the new variable points to the same reference in the memory heap.
* Deep copies of objects can be made with external libraries.
* The spread operator works on all iterables, NOT in objects.
* Spread operator can only be used when building an array or when we pass values into a function. BUTTT since es2018, spread operator can also be used for objects.
* The spread and the rest syntax both look exactly the same, but they work in opposite ways. The spread operator is used where we would otherwise write values separated by a comma, on the other hand, the rest pattern is used where we would otherwise write variables separated by a comma.
* The Nullish Coalescing operator (??) introduced in es2020 evaluates only the nullish (null, undefined, ) values, different to the || that evaluates falsy values.
* Logical assignment operators were introduced in es2021 (||=) (&&= ) (??=)
* Since es2020 we also have the Optional Chaining operator ‘?.’
* Loop through objects:
* Object.keys(object) returns an array with the key names of the object.
* Object.values(object) returns an array with the values of the object.
* Object.entries(object) returns an array with the key names and the values of the object [key, value].
* Sets were introduced to javascript on es6, a Set (new Set()) is a collection of unique values. NOTE: length property is called size for Sets | .includes() method is called .has() for Sets | As order is irrelevant for Sets, then the push() or unshift() method for Sets is called add() | For the same reason, shift() or pop() methods for Sets is called delete() | .clear() to empty the Set.
* Maps were introduced to javascript on es6, a Map (new Map()) is similar to an object, but with the difference that the keys on a Map can be anything, a string, number, object, function, etc. Methods are .set() which not only adds an element to the map, but also returns the updated Map | .get() | .has() | .delete() | .clear()
* WeakMap and WeakSet? CHECK THIS LATER
* JavaScript has First-Class functions, which means that functions are values, so they can be stored as variables, sent to another function as a parameter or returned by another function (High-order function).
* .name property of a function might be useful for debugging purposes.
* The .call() and .apply() are methods for functions, what these methods do is call/apply the function but defining explicitly the where the ‘this’ keyword points at. The first argument of the method is the object to which the ‘this’ keyword should point at and the rest of the arguments are the same arguments that need to be sent to the function (in a comma separated way for the call method and inside an array for the apply method).
* The .bind() is another method for functions, similar to call and apply, but with bind you only attach the ‘this’ keyword to the new function without calling it. Also with bind you can hardcode the arguments passed to the new function. This method is useful with DOM eventListeners, as in those the ‘this’ keyword points to the DOM element, so the function needs to be binded to the object to which it wants to be applied.
* Immediately Invoked Function Expressions (IIFE) are used to invoke functions without storing them into variables, they are also used to have private variables, also to avoid polluting the global namespace.
* A function has access to the Variable Environment (VE) of the execution context in which it was created. CLOSURE: VE attached to the function, exactly as it was at the time and place the function was created. A closure gives a function access to all the variables of its parent function, even after the parent function has returned. The function keeps a reference to its outer scope, which preserves the scope chain throughout time.
* The forEach method for arrays, maps and sets is similar to the for of loop, but with an easier syntax, access to the current array at each iteration with the downside that there is no break or continue in that method.
* Array data transformations:
* .map() returns a new array containing the values returned after applying the callback function on each array value.
* .filter() returns a new array containing the array elements that passed a specified test condition (true). NOTE: The return value of the callback function should be a Boolean, return true if the elements needs to be added to the new array or false if it does not.
* .reduce() bolis (reduces) all the array elements down to one single value (e.g. adding all elements together). The return value of the callback function should be the new value of the accumulator after each iteration. [].reduce((accumulator, element, index, array) => accumulator + element, initialAccumulatorValue)