**Node JS and Javascript**

**Training**

**By Suresh Chouksey**

**Index**

1. Introduction of Javascript

2. Scope

3. Closures

4. Set

5. Map

6. Array Methods

7. Typescript

8. ES6 Arrow Function(=>)

**Introduction of Javascript**

**JavaScript** (**JS**) is a lightweight, interpreted or JIT compiled programming language with first-class functions. Most well-known as the scripting language for Web pages, many non-browser environments also use it, such as node.js and Apache CouchDB. JS is a prototype-based, multi-paradigm, dynamic scripting language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

The standard for JavaScript is ECMAScript. As of 2012, all modern browsers fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3. On June 17, 2015, ECMA International published the sixth major version of ECMAScript, ECMAScript 2015. Since then, ECMAScript standards are on yearly release cycles. This documentation refers to the latest draft version, which is currently ECMAScript 2019.

Do not confuse JavaScript with the Java programming language. Both "Java" and "JavaScript" are trademarks or registered trademarks of Oracle in the U.S. and other countries. However, the two programming languages have very different syntax, semantics, and uses.

Ref: <https://developer.mozilla.org/bm/docs/Web/JavaScript>

**Scope:**

we have two types of scope, we have two types of execution context. They are a global execution context and a function execution context.

The global context is always running. In the case of a browser environment, it only stops when the browser is closed.

Parameters to a function are implicitly defined. They are “local” to that function’s scope. These declared variables are “hoisted”, taken to the top of the scope that they belong to.

var a='global';

function myfun(){

//var a After variable hoisting

a='local';

console.log('inside',a);

var a;

}

myfun()

console.log('outside',a);

**Closures:**

Closures are the best form of privacy for functions and variables. This is evident in the use of many module patterns. A module pattern returns an object to expose a public API. It also keeps other methods and variables private. Closures are used in event handling and callbacks.

1. Closures are functions that have access to variables from another function's scope. This is accomplished by creating a function inside another function.

2. A Closure is a function that returns another function.

3. A Closure is an implicit, permanent link between a function and its scope chain.

function firstName(first){

function fullName(last){

console.log(first + " " + last);

}

return fullName;

}

var name = firstName("Mister");

name("Smith") // Mister Smith

name("Jones"); //Mister Jones

The inner function fullName( ) is accessing the variable, first, in its outer scope,

firstName( ). Even after the inner function, fullName, has returned, it still has access to that variable. How is this possible? The inner function’s scope chain includes the scope of its outer scope.

**Variable capturing quirks**

Take a quick second to guess what the output of the following snippet is:

**for** (**var** i = 0; i < 10; i++) {  
 setTimeout(**function**() { console.log(i); }, 100 \* i);  
}

= 10,10,10,10....

Expected Output: 0,1,2,3,4,

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture i at each iteration:

**for** (**var** i = 0; i < 10; i++) {  
 // capture the current state of 'i'  
 // by invoking a function with its current value  
 (**function**(i) {  
 setTimeout(**function**() { console.log(i); }, 100 \* i);  
 })(i);  
}

var sum = function(a){

return function(b){

return function(c){

return a+b+c;

}

}

}

sum(1)(2)(3)

6

**SET:**

Ref: <https://medium.freecodecamp.org/lets-learn-about-set-and-its-unique-functionality-in-javascript-5654c5c03de2>

**So what is Set, you ask?**

“The Set object lets you store unique values of any type, whether primitive values or object references.”, MDN.

Set removes duplicate entries.

All valid ways to initialize set.

const newSet1 = new Set();

const newSet2 = new Set(null);

const newSet3 = new Set([1, 2, 3, 4, 5]);

Set utilities/methods

add, like its name suggests, adds new entries to the newly initialized Set const. If at any time a duplicate value gets added to the set, it will be discarded using strict equality.

const newSet = new Set();

newSet.add("C");

newSet.add(1);

newSet.add("C");

// chain add functionality

newSet.add("H").add("C");

newSet.forEach(el => {

console.log(el);

// expected output: C

// expected output: 1

// expected output: H

});

**Map:**

The **Map** object holds key-value pairs. Any value (both objects and primitive values) may be used as either a key or a value.

**Objects and maps compared**

Objects are similar to Maps in that both let you set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. Because of this (and because there were no built-in alternatives), Objects have been used as Maps historically; however, there are important differences that make using a Map preferable in certain cases:

* The keys of an Object are Strings and Symbols, whereas they can be any value for a Map, including functions, objects, and any primitive.
* The keys in Map are ordered while keys added to object are not. Thus, when iterating over it, a Map object returns keys in order of insertion.
* You can get the size of a Map easily with the size property, while the number of properties in an Object must be determined manually.
* A Map is an iterable and can thus be directly iterated, whereas iterating over an Object requires obtaining its keys in some fashion and iterating over them.
* An Object has a prototype, so there are default keys in the map that could collide with your keys if you're not careful. As of ES5 this can be bypassed by using map = Object.create(null), but this is seldom done.
* A Map may perform better in scenarios involving frequent addition and removal of key pairs.

Ref: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map>

**Array Methods**

**forEach**

forEach dont return anything. it just run the callback functionf or each element of the array

[1,2,3].forEach(function(elem,index,arr){

Console.log(elem)

})

O/p: 1,2,3

**Map**

returns new array by executing the callback function for each elements of the array

[1,2,3].map(function(elem, index, arr){

return elem \* elem;

})

o/p: [1,4,9]

**filter**

if the condition is true for an element, element is picked for the return array

[1,2,3,4,5,6,7,8].filter(function(elem, index, arr){

return index % 2 ==0;

});

= [1,3,5,7]

**Some**

if any of element in the arrya fulfil the condition, some will return true else will return false

[1,2,3,4].some(function(elem, index, arr){

return elem>3

});

= true;

**Every**

it will return true if every elements in the array fulfil the condition. else return false

[1,2,3,4].every(function(elem, index, arr){

return elem>3

});

= false;

[1,2,3,4].every(function(elem, index, arr){

return elem>0

});

= true

**Reduce**

;

reduce cache the value. for example u want to get the sum of all the element. alternatively, you can initially.

[1,2,3,4].reduce(function(elem, index, arr){

return sum + el;

});

=10;

[1,2,3,4].reduce(function(elem, index, arr){

return sum + el;

},100);

=110;

Ref: <http://thatjsdude.com/jsConcepts/concepts/array.html#array_method>

**TypeScript**

TypeScript starts from the same syntax and semantics that millions of JavaScript developers know today. Use existing JavaScript code, incorporate popular JavaScript libraries, and call TypeScript code from JavaScript.

TypeScript compiles to clean, simple JavaScript code which runs on any browser, in Node.js, or in any JavaScript engine that supports ECMAScript 3 (or newer).

Basic Type:

**Boolean**

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

**let** isDone: boolean = false;

**Number**

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

**let** decimal: number = 6;

**let** hex: number = 0xf00d;

**let** binary: number = 0b1010;

**let** octal: number = 0o744;

**String**

**let** color: string = "blue";

color = 'red';

You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

**let** fullName: string = `Bob Bobbington`;

**let** age: number = 37;

**let** sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;

This is equivalent to declaring sentence like so:

**let** sentence: string = "Hello, my name is " + fullName + ".\n\n" +

"I'll be " + (age + 1) + " years old next month.";

**Array**

**let** list: number[] = [1, 2, 3];

The second way uses a generic array type, Array<elemType>:

**let** list: Array<number> = [1, 2, 3];

**Tuple**

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

// Declare a tuple type

**let** x: [string, number];// Initialize it

x = ["hello", 10]; // OK// Initialize it incorrectly

x = [10, "hello"]; // Error

When accessing an element with a known index, the correct type is retrieved:

console.log(x[0].substr(1)); // OK

console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'

When accessing an element outside the set of known indices, a union type is used instead:

x[3] = "world"; // OK, 'string' can be assigned to 'string | number'  
console.log(x[5].toString()); // OK, 'string' and 'number' both have 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'

Union types are an advanced topic that we’ll cover in a later chapter.

**Enum**

A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

**enum** Color {Red, Green, Blue}

**let** c: Color = Color.Green;

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

**enum** Color {Red = 1, Green, Blue}

**let** c: Color = Color.Green;

Or, even manually set all the values in the enum:

**enum** Color {Red = 1, Green = 2, Blue = 4}

**let** c: Color = Color.Green;

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren’t sure what that mapped to in the Color enum above, we could look up the corresponding name:

**enum** Color {Red = 1, Green, Blue}

**let** colorName: string = Color[2];

console.log(colorName); // Displays 'Green' as its value is 2 above

**let declarations**

**let** hello = "Hello!";

The key difference is not in the syntax, but in the semantics, which we’ll now dive into.

**Block-scoping**

When a variable is declared using let, it uses what some call lexical-scoping or block-scoping. Unlike variables declared with var whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or for-loop.

**function** **f**(input: boolean) {

**let** a = 100;

**if** (input) {

// Still okay to reference 'a'

**let** b = a + 1;

**return** b;

}

// Error: 'b' doesn't exist here

**return** b;

}

Variables declared in a catch clause also have similar scoping rules.

**try** {

**throw** "oh no!";

}**catch** (e) {

console.log("Oh well.");

}

// Error: 'e' doesn't exist hereconsole.log(e);

Another property of block-scoped variables is that they can’t be read or written to before they’re actually declared.

a++; // illegal to use 'a' before it's declared;

**let** a;

Something to note is that you can still capture a block-scoped variable before it’s declared.

**function** **foo**() {

// okay to capture 'a'

**return** a;

}

// illegal call 'foo' before 'a' is declared// runtimes should throw an error here

foo();

**let** a;

**Re-declarations and Shadowing**

With var declarations, we mentioned that it didn’t matter how many times you declared your variables; you just got one.

**function** **f**(x) {

**var** x;

**var** x;

**if** (true) {

**var** x;

}

}

In the above example, all declarations of x actually refer to the same x, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, let declarations are not as forgiving.

**let** x = 10;

**let** x = 20; // error: can't re-declare 'x' in the same scope

The variables don’t necessarily need to both be block-scoped for TypeScript to tell us that there’s a problem.

**function** **f**(x) {

**let** x = 100; // error: interferes with parameter declaration

}

**function** **g**() {

**let** x = 100;

**var** x = 100; // error: can't have both declarations of 'x'

}

That’s not to say that block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

**function** **f**(condition, x) {

**if** (condition) {

**let** x = 100;

**return** x;

}

**return** x;

}

f(false, 0); // returns '0'

f(true, 0); // returns '100'

**Block-scoped variable capturing**

When we first touched on the idea of variable capturing with var declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an “environment” of variables. That environment and its captured variables can exist even after everything within its scope has finished executing.

**function** **theCityThatAlwaysSleeps**() {

**let** getCity;

**if** (true) {

**let** city = "Seattle";

getCity = **function**() {

**return** city;

}

}

**return** getCity();

}

Because we’ve captured city from within its environment, we’re still able to access it despite the fact that the if block finished executing.

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

setTimeout(**function**() { console.log(i); }, 100 \* i);

}

Output: 0,1,2,3,4....

**ES6 Arrow Functions**

Arrow functions (also called “fat arrow functions”) are undoubtedly one of the more popular features of ES6. They introduced a new way of writing concise functions.

Here is a function written in ES5 syntax:

function timesTwo(params) {  
 return params \* 2  
}

timesTwo(4); // 8

Now, here is the same function expressed as an arrow function:

var timesTwo = params => params \* 2

timesTwo(4); // 8

**Variations**

1. **No parameters**

If there are no parameters, you can place an empty parentheses before =>.

() => 42

**2. Single parameter**

With these functions, parentheses are optional:

x => 42 || (x) => 42

**3. Multiple parameters**

Parentheses are required for these functions:

(x, y) => 42

**4. Statements (as opposed to expressions)**

Here is an example of the arrow function used with an if statement:

var feedTheCat = (cat) => {  
 if (cat === 'hungry') {  
 return 'Feed the cat';  
 } else {  
 return 'Do not feed the cat';  
 }  
}

**5. “Block body”**

If your function is in a block, you must also use the explicit return statement:

var addValues = (x, y) => {  
 return x + y  
}

**6. Object literals**

If you are returning an object literal, it needs to be wrapped in parentheses. This forces the interpreter to evaluate what is inside the parentheses, and the object literal is returned.

x =>({ y: x })

Ref: <https://medium.freecodecamp.org/when-and-why-you-should-use-es6-arrow-functions-and-when-you-shouldnt-3d851d7f0b26>