The complete Javascript course

**Fundamentos parte 1:**

Una **expresión** es un fragmento de código que produce un valor. Ejemplo: 3 + 4.

Una **declaración** es un fragmento de código más grande que se ejecuta y que no produce un valor por sí mismo. Es como la estructura del código, por ejemplo, el if.

**Fundamentos parte 2:**

**Modo estricto:** es un modo especial que podemos activar en JS, el cual nos facilita escribir código seguro.

Para activarlo debemos colocar el principio del archivo script.js el siguiente string: “use strict”

**Functions:** están las declaradas y las expresadas. Un parámetro es lo que ponemos como nombre en los paréntesis, un argumento es el valor que le damos a ese parámetro.

Las funciones declaradas son aquellas que declaramos con function nombre (){}. A estas funciones las podemos llamar antes de ser declaradas.

En cambio, una función expresada la tenemos que guardar en una variable y va sin nombre la function.

**Ciclo for:**

En el ciclo for tenemos dos declaraciones que podemos agregar:

* **Continue:** la cual sirve para salir de la iteración actual del ciclo y continuar con la siguiente. Ejemplo:
* for (let i = 0; i < jonas.length; i++) {
* if (typeof jonas[i] !== 'string') continue;
* console.log(jonas[i], typeof jonas[i]);
* }
* **Break:** se utiliza para terminar completamente el ciclo. Ejemplo:
* for (let i = 0; i < jonas.length; i++) {
* if (typeof jonas[i] === 'number') break;
* console.log(jonas[i], typeof jonas[i]);
* }

**While for:**

No depende de una variable de contador. No sabemos cuantas iteraciones tendrá el ciclo for.

let dice = Math.trunc(Math.random() \* 6) + 1;

while (dice !== 6) {

  console.log(`You rolled a ${dice}`);

  dice = Math.trunc(Math.random() \* 6) + 1;

  if (dice === 6) console.log('Loop is about to end...');

}

**DEVELOPER SKILLS-03:**

**Prettier:** es una extensión que sirve para ordenar el código. Podemos hacer un archivo “.prettierrc” y abrir un objeto {} el cual le especificamos que configuraciones queremos que tenga, como por ejemplo que escriba doble comillas o simples, etc.

**Pasos para resolver problemas:**

1. Asegurarse de que estés entendiendo bien el problema. Hacer las preguntas correctas para conseguir un claro panorama del problema.
2. Divide y vencerás: hay que dividir el problema en pequeños problemas, porque de esta manera los pequeños problemas son más fáciles de resolver.
3. Si no sabemos resolver uno de estos pequeños problemas solamente tenemos que investigar. Google, Stack Overflow, MDN.

**Debugging:**

Poseemos aparte del console.log() otros tipos de console:

* Console.warn(): te da una advertencia del dato.
* Console.error() te da un error.
* Console.table() te muestra en una tabla.

Sirven para encontrar errores.

En las herramientas de programador en Google, en la ventana de “Source” podemos poner un breakpoint y evaluar hasta ahí cómo va el código para encontrar el error.

**GUESS MY NUMBER:**

Para hacer un numero aleatorio utilizamos el método **Math** con su propiedad **random**. Esto no provoca un numero aleatorio del 0 al 1, con decimales incluidos. Si queremos que llegue hasta el 20 lo multiplicamos por 20. Ej: Math.random()\*20

Ahora a todo esto para que nos de un número entero debemos encerrarlo con el método Math nuevamente, pero en su propiedad **trunc**, y sumarle 1. Ej: **Math.trunc(Math.random()\*20+1)**

**PRIMITIVES VALUES VS REFERENCES VALUES:**

Para fusionar dos objetos y que nos devuelva un objeto nuevo en el cual cambiemos sus datos podemos hacerlo de las siguientes maneras:

**De manera superficial:** es decir que si dentro de un objeto tenemos otro objeto este no podrá copiar ese sub-objeto. Por ejemplo:

const jessica = {

  firstName: 'Jessica',

  lastName: 'Williams',

};

const jessicaCopy = Object.assign({}, jessica);

jessicaCopy.lastName = 'Davis';

Este nuevo objeto llamado jessicaCopy poseerá todas las propiedades y valores del objeto jessica, pero le modificamos su apellido en “Davis”.

**De manera profunda:** copia todo lo que tenga el objeto. Por ejemplo:

No lo vi en la sección. Pero se usa una librería como LoDash.

Lo que pasa es que cuando al objeto jessica le asignamos un array de familia, y al hacer el nuevo objeto jessicaCopy modificamos ese array, el objeto original también se modica, y no es lo que queremos.

**DESTRUCTURING ARRAYS:**

const array = [1, 2, 3];

const [a, b, c] = array;

console.log(a, b, c) // 1, 2, 3

**Si quiero saltarme un elemento y tomar el primero y el tercero por ejemplo sería así:**

const [first, , third] = array;

console.log(first, third)// 1, 3

**Para reasignar valores podríamos hacerlo de la siguiente manera:**

const array = [1, 2, 3];

const [main, secondary] = array;

console.log(main, secondary) // 1, 2

[main, secondary] = [secondary, main]

console.log (main, secondary) // 2, 1

Con esto estamos diciendo que main ahora tiene el valor de secondary, y secondary tiene el valor de main.

**Para destructurar un array que tiene a la vez un array dentro haríamos así:**

const nested = [1, 2, [3, 4]];

const [oneElem, , [thirdElem, fourElem]] = nested;

console.log(oneElem, thirdElem, fourElem)// 1, 3, 4;

**Default values:** para destructurar un array que no sabemos cuántos elementos tiene.

const [p = 1, q = 1, r=1] = [8, 9];

console.log(p,q,r); //8, 9, 1

Como no sabemos cuantos elementos puede tener un array le asignamos un valor por defecto. Entonces como el array no tiene un tercer elemento, “r” pasa a valer 1, por el valor que le dimos por defecto.

**DESTRUCTURING OBJECTS:**

A diferencia que en la destructuración con arrays, en los objetos no hay que dejar un espacio en blanco para saltearse el orden.

const restaurant = {

  name: 'Classico Italiano',

  location: 'Via Angelo Tavanti 23, Firenze, Italy',

  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],

  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],

  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {

    thu: {

      open: 12,

      close: 22,

    },

    fri: {

      open: 11,

      close: 23,

    },

    sat: {

      open: 0, // Open 24 hours

      close: 24,

    },

  },

};

Así destructuramos:

const {name, openingHours, categories} = restaurant

**Si queremos darles nuevos nombres a esas variables destructuradas haríamos así:**

const {name: restaurantName, openingHours: hours, categories: tags} = restaurant

console.log(restaurantName, hours, tags);

**Default values:**

const {menu = [], starterMenu: starters = []} = restaurant

console.log(menu, starters); //[], todo el objeto de starters...

**Mutating variables:**

let a = 111;

let b = 999;

const obj = {

  a: 23,

  b: 7,

  c: 14

}

({a, b} = obj);

console.log(a,b); //ahora a=23 y b=7

Hay que envolverlo entre paréntesis porque si arrancamos una sintaxis con {} Javascript espera un bloque de código.

**Nested objects:**

const {openingHours} = restaurant;

const {fri: {open, close}} = openingHours

console.log(open, close); //11, 23

**SPREAD OPERATOR:**

Extrae todos los elementos de un array.

const arr = [4, 5, 6];

const newArr = [1,2,3, ...arr]

console.log(newArr);//1,2,3,4,5,6

**En un string:**

const str = "joaquin"

const letters = [...str];

console.log(letters); //"j","o","a","q","u","i","n"

**En un objeto:**

const newRestaurant = {...restaurant, founder: "Joaquín"};

console.log(newRestaurant);//todo el objeto de restaurante, + founder: "Joaquín"

**REST PATTERN AND PARAMETERS:**

Reste pattern recopila los elementos que no se utilizan en la destructuración. El rest pattern debe ser el último elemento.

//  Rest because on LEFT of =

const [a, b, ...others] = [1, 2, 3, 4, 5]

console.log(a, b, others);//1, 2, [3, 4, 5]

const [pizza, pasta, ...otherfood] = [...restaurant.mainMenu, ...restaurant.starterMenu];

console.log(pizza, pasta, otherfood);//pizza, pasta, ['Risotto','Focaccia','Bruschetta','Garlic Bread','Caprese Salad']

**Object:**

const { sat, ...weekdays} = restaurant.openingHours;

console.log(weekdays); //muestra el objeto de los días thu y fri

**Function:**

function add (...numbers) {

  let sum = 0;

  for (let i = 0; i < numbers.length; i++) {

    sum += numbers[i]

  }

  console.log(sum);

}

add(5,4); //9

add(4,6,3,4); //17

add(2,6,7);//15

const x = [1,2,3]

// spread operator

add(...x)//6

**DIFERENCIA ENTRE SPREAD OPERATOR Y REST PATTERN:**

* Spread operator lo utilizamos cuando escribimos valores separados por coma.
* Rest pattern lo utilizamos cuando escribimos nombres de variables separadas por coma.

**SHORT CIRCUITING (&&, ||):**

* Pueden usar cualquier tipo de dato.
* Retornan cualquier tipo de dato.
* Hacen el short-circuiting.

El operador || devolverá el primer valor verdadero que haya, en caso de que no haya un booleano.

**3 || “joaquin” \\ devuelve el 3.**

El short-circuiting del operador && corta cuando el primer valor es falso.

**0 && “Joaquin” \\ devuelve 0**

Si el primer valor es verdadero continua la evaluación y devuelve el último valor.

**7 && “joaquin” \\ devuelve “joaquin”**

**THE NULLISH COALESCING OPERATOR (??)**

Este operador trabaja con la idea de valores NULL en lugar de valores FALSY.

Los valores nullish son NULL o UNDEFINED. No incluye el 0, ni un string vacío “”.

restaurant.numGuest = 0;

const guestCorrect = restaurant.numGuest ?? 10;

console.log(guestCorrect);// el valor da 0

**LOGICAL ASSIGMENTS OPERATORS**

const rest1 = {

  name: "Capri",

  numGuests: 20

}

const rest2 = {

  name: "La Piazza",

  owner: "Giovanni Rossi"

}

rest2.numGuests = rest2.numGuests || 10;

rest1.numGuests = rest1.numGuests || 10;

**Operador de asignación de órdenes (||=):** Podemos escribir la línea anterior de una manera más rápida. Este operador asigna un valor a una variable si esa variable es actualmente falsa.

// Operador de asignación de órdenes

rest1.numGuests ||= 10;

rest2.numGuests ||= 10;

**Operador de asignación nullish (??=):** asigna un valor a una variable si esa variable exacta es actualmente nullish.

// Operador de asignación nullish

// Si el rest1 tuviera de numGuests = 0, las líneas anteriores darían de resultado 10 porque 0 es un valor falso

rest1.numGuests ??= 10;

rest2.numGuests ??= 10;

**Operador de asignación AND(&&=):** asigna un valor a una variable si es actualmente TRUTY.

// Operador de asignación AND

rest1.owner &&= "ANONYMOUS";

rest2.owner &&= "ANONYMOUS";

**THE FOR OF LOOP**

// FOR OF LOOP

const menu = [...restaurant.mainMenu, ...restaurant.starterMenu];

for (const item of menu) console.log(item);//Pizza Pasta Risotto Focaccia Bruschetta Garlic Bread Caprese Salad

**Si queremos también el índice utilizamos .entries():**

for (const item of menu.entries()) {

  console.log(item); //[ 1, 'Pasta' ][ 2, 'Risotto' ][ 3, 'Focaccia' ][ 4, 'Bruschetta' ][ 5, 'Garlic Bread' ][ 6, 'Caprese Salad' ]

}

**Podemos destructurar:**

for (const [i, el] of menu.entries()) {

  console.log(`${i + 1}: ${el}`);//1: Pizza 2: Pasta 3: Risotto 4: Focaccia 5: Bruschetta 6: Garlic Bread 7: Caprese Salad

}

**ENHANCED OBJECT LITERALS**

1. En el Javascript moderno si tenemos un objeto por fuera de otro objeto y queremos agragerlo adentro del objeto ya no hace falta escribir por ejemplo openingHours: openingHours. Con tan solo escribirlo una sola vez el sistema ya lo reconoce.
2. Para escribir funciones ya no hace falta escribirla con un nombre de propiedad: function… Ahora con escribir el nombre de la function ya está. Ejemplo: order () {…}
3. Ahora podemos calcular los nombres de las propiedades en lugar de tener que escribirlos de forma manual y literal.
4. const weekdays = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"];
5. const openingHours = {
6. [weekdays[3]]: {
7. open: 12,
8. close: 22,
9. },
10. [weekdays[4]]: {
11. open: 11,
12. close: 23,
13. },
14. [weekdays[5]]: {
15. open: 0, // Open 24 hours
16. close: 24,
17. },
18. }

**OPTIONAL CHAINING (?)**

if (restaurant.openingHours && restaurant.openingHours.mon) {

  console.log(restaurant.openingHours.mon.open);

}

// WITH OPTIONAL CHAINING

console.log(restaurant.openingHours.mon?.open);

Si esto no existe devuelve automáticamente UNDEFINED.

Podemos agregar todos los optional chaining que queramos evaluar

Ejemplo:

console.log(restaurant.openingHours?.mon?.open);

const days = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"];

for(const day of days) {

  const open = restaurant.openingHours[day]?.open ?? "closed";

  console.log(`On ${day}, we open at ${open}`);

}

**Se puede utilizar para verificar métodos:**

console.log(restaurant.order?.(0,1) ?? "Method does not exit");

Primero verifico si el método existe, y luego lo ejecuto.

**Como también para verificar si un array está vacío:**

const users = [{name:"joaquin"}];

console.log(users[0]?.name ?? "User is empty");//joaquin

**LOOPING OBJECTS**

**Recorrer las propiedades(keys):** Utilizamos Object.keys()

for (const day of Object.keys(restaurant.openingHours)){

  console.log(day);//thu, fri, sat

}

**Recorrer los valores(values):** Utilizamos Object.values()

const values =Object.values(restaurant.openingHours);

console.log(values);//[{ open: 12, close: 22 },{ open: 11, close: 23 },{ open: 0, close: 24 }]

**Recorrer keys y values:** Utilizamos Object.entries()

const entries = Object.entries(restaurant.openingHours);

console.log(entries);//[[ 'thu', { open: 12, close: 22 } ],[ 'fri', { open: 11, close: 23 } ],[ 'sat', { open: 0, close: 24 } ]]

for (const [key, {open, close}] of entries) {

  console.log(`On ${key} we open at ${open} and close at ${close}`);

  //On thu we open at 12 and close at 22. On fri we open at 11 and close at 23. On sat we open at 0 and close at 24

}

**SETS**

Los **sets** son una colección de valores únicos. Eso significa que un **set** nunca puede tener duplicados.

Para crear un set debemos escribir **new Set(**le pasamos un iterable, como un array, string**)**

const orderSet = new Set(["pizza", "pasta", "pasta", "pizza", "risotto"])

console.log(orderSet);//pizza, pasta, risotto

Como te das cuenta, elimina los duplicados.

**Set.size:** da la cantidad de elementos que posee el set.

const orderSet = new Set(["pizza", "pasta", "pasta", "pizza", "risotto"])

console.log(orderSet.size); // 3

**Set.has(**preguntamos por el elemento que buscamos**):** para ver si un elemento está dentro del set. Nos devuelve un booleano.

console.log(orderSet.has("pizza")); //true

**Set.add(**pasamos el elemento que queremos agregar**):**

orderSet.add("Garlic Bread");

console.log(orderSet);//'pizza', 'pasta', 'risotto', 'Garlic Bread'

**Set.delete(**pasamos el elemento que queremos borrar**):**

orderSet.delete("risotto")

console.log(orderSet);//'pizza', 'pasta', 'Garlic Bread'

**Set.clear():** sirve para eliminar todos los elementos de un set.

**Podemos iterar sobre un set:**

for (const order of orderSet) console.log(order); //pizza, pasta, Garlic Bread

**Transformar un set en un array:** usamos el spread operator

const staff = [“Waiter”, “Chef”, “Manager”, “Waiter”, “Chef”, “Waiter”];

const staffUnique = […new Set(staff)];

console.log(staffUnique); //[ ‘Waiter’, ‘Chef’, ‘Manager’ ]

**MAPS:**

Un map es una estructura de datos que podemos usar para asignar valores a claves. La diferencia entre un map y objeto, es que en los maps las keys pueden tener cualquier tipo. En los objetos, las keys son básicamente strings.

Al igual que antes con los sets, los maps se crean de la misma manera:

**new Map()**

**Llenar el Map: map.set(**keyName, value**)**

const rest = new Map();

rest.set("name", "Classico Italiano");

rest.set(1, "Firenze, Italy");

console.log(rest);//{ 'name' => 'Classico Italiano', 1 => 'Firenze, Italy' }

El .set() nos devuelve el Map actualizado, por lo que podemos seguir haciendo .set() al terminar algo seteado, ejemplo:

rest

  .set(2, 'Lisbon', 'Portugal')

  .set('categories', ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'])

  .set('open', 11)

  .set('close', 23)

  .set(true, 'We are open')

  .set(false, 'We are closed');

**Map.get(**pasamos el nombre de la key**):** para leer datos de un map.

console.log(rest.get("name"));//Classico Italiano

console.log(rest.get(true));//We are open

**Map.has(**keyName**):** determina si el Map tiene cierta key. Devuelve un booleano.

console.log(rest.has("categories"));//true

**Map.delete(**keyName**):** borra una key.

rest.delete(2);//borro la segunda ubicación del restaurante

**Map.size:** para ver cuántos elementos tenemos.

console.log(rest.size);//8

**Map.clear():** borra todo del Map.

rest.clear();

console.log(rest);//Map(0) {}