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 cuántas 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 agregarlo 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.

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

const openingHours = {

[weekdays[3]]: {

open: 12,

close: 22,

},

[weekdays[4]]: {

open: 11,

close: 23,

},

[weekdays[5]]: {

open: 0, // Open 24 hours

close: 24,

},

}

**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 de dato. 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) {}

**MAPS ITERATION:**

**Otra forma de inicializar un Map:** pasandole un array de arrays, en cada array especificamos key y value.

const question = new Map([

  ["question", "What is the best programming language in the world?"],

  [1, "C"],

  [2, "Java"],

  [3, "Javascript"],

  ["correct", 3],

  [true, "Correct"],

  [false, "Try again!"]

]);

**Convertir un objeto en un map:** con Object.entries()

const hoursMap = new Map(Object.entries(restaurant.openingHours))

console.log(hoursMap); //Map(3) {'thu' => { open: 12, close: 22 },'fri' => { open: 11, close: 23 },'sat' => { open: 0, close: 24 }}

**ITERAR:**

for (const [key, value] of question) {

  if (typeof key === "number") {

    console.log(`Answer ${key}: ${value}`);

    //Answer 1: C, Answer 2: Java, Answer 3: Javascript

  }

}

**Convertir un map en un array:** con el spread operator.

console.log([...question]); //[[ 'question', 'What is the best programming language in the world?' ],[ 1, 'C' ],[ 2, 'Java' ],[ 3, 'Javascript' ],[ 'correct', 3 ],[ true, 'Correct' ],[ false, 'Try again!' ]]

**¿QUÉ ESTRUCTURA DE DATOS USAR?**

Preguntas que debemos hacernos:

1. **¿Solo necesitamos una lista simple de valores?**

Si es así, usaremos un **array** o un **set**.

Si necesitamos key-value, entonces usaremos un **object** o un **map**. Lo que nos permite tener una descripción del value, porque tenemos una key descriptiva.

**ARRAYS vs SETS:**

Debemos usarlos para listas simple de valores, cuando no necesitamos describir los valores.

|  |  |
| --- | --- |
| **ARRAY** | **SETS** |
| Almacenar valores en orden y puedan tener valores duplicados. | Usarlo cuando trabajemos con valores únicos |
| Cuando necesitemos manipular datos | Cuando la alta performance sea realmente importante, porque operaciones como buscar un elemento o borrarlo puede ser 10 veces más rápido en sets que en arrays. |
|  | Usarlo para remover valores duplicados de un array. |

**OBJECTS vs MAPS:**

Debemos utilizar esta estructura de key-values data, cada vez que necesitemos describir los valores usando claves.

|  |  |
| --- | --- |
| **OBJECTS** | **MAPS** |
| Es el más tradicional, pero está siendo muy abusado de su uso. | Ofrece una mejor performance. |
| Son fáciles de escribir y de acceder a los datos usando simplemente **.** o el operador de corchetes []. | Las keys pueden tener cualquier tipo de dato |
|  | Son fáciles de iterar. |
|  | Y fáciles de calcular el tamaño del Map. |
| Usar cuando necesitemos incluir funciones (methods), accedemos a ellas con la keyword **this**. Lo cual no tienen los Maps. | Usar cuando necesitemos mapear key y values. |
| Usar cuando trabajemos con un JSON. | Usar cuando necesitemos keys que no sean strings |

**MÉTODOS DE LOS STRING**

**length:** para ver la longitud de caracteres del string.

**indexOf(**pasar la letra o palabra que buscamos**):** para encontrar alguna letra o palabra.

**lastIndexOf(**pasar la letra o palabra**):** para encontrar la última letra que buscamos.

**.slice():** dentro de los paréntesis le pasamos el índice donde queremos que empiece a buscar, y si queremos un índice final donde termine de buscar. A su vez, el valor final no está incluido en el string. Devuelve un subString.

const airlane = "TAP Air Portugal";

console.log(airlane.slice(4));//Air Portugal

**.toLowerCase():** no necesita argumentos. Transforma todo el string en minúsculas.

console.log(airlane.toLowerCase());//tap air portugal

**.toUpperCase():** no necesita argumentos. Transforma todo el string en mayúsculas.

console.log(airlane.toUpperCase());//TAP AIR PORTUGAL

Ejemplo para combinar uppercase y lowercase:

const passenger = "jOnAS";

const passengerLower = passenger.toLowerCase();

const passengerCorrect = passengerLower[0].toUpperCase() + passengerLower.slice(1);

console.log(passengerCorrect); //Jonas

**.trim():** método para recortar los espacios que se encuentran al principio y final de un string.

const email = "hello@jonas.io";

const loginEmail = "   Hello@jonas.io   ";

const lowerEmail = loginEmail.toLowerCase();

const trimmedEmail = lowerEmail.trim();

console.log(trimmedEmail);//hello@jonas.io

// Podemos hacerlo en una sola línea

const normalizedEmail = loginEmail.toLocaleLowerCase().trim();

console.log(normalizedEmail);//hello@jonas.io

**.replace():** dentro de los paréntesis pasamos dos argumentos, el primero es el elemento que queremos reemplazar, y el segundo es el elemento por el cual lo reemplazamos.

const priceGb = "288,97£";

const priceUS = priceGb.replace("£", "$").replace(",", ".");

console.log(priceUS);//288.97$

**Métodos que devuelven booleans:**

const plane = "A320neo"

**Includes():** pasarle en los paréntesis el string que estemos buscando.

console.log(plane.includes("A320"));//true

**startsWith():** pasarle en los paréntesis el string con el que pensamos que empieza, nos devuelve true o false dependiendo si arranca con lo que escribimos como parámetro o no.

console.log(plane.startsWith("Air"));//false

**endsWith():** igual al anterior pero preguntando si el string termina con lo que le pasamos como parámetro.

console.log(plane.endsWith("neo"));//true

**.split():** nos permite dividir un string en múltiples partes en función de una cadena divisoria. Le pasamos como argumento un elemento divisorio. Nos devuelve un array.

console.log("a+very+nice+string".split("+"));//[ 'a', 'very', 'nice', 'string' ]

console.log("Joaquin Caggiano".split(" "));//[ 'Joaquin', 'Caggiano' ]

const [firstName, lastName] = "Joaquin Caggiano".split(" ");

console.log(firstName, lastName);//joaquin caggiano

**.join():** une un string, debemos pasarle como parámetro como lo queremos unir, con un espacio, un guion, etc.

Le pasamos un espacio:

const newName = ["Mr.", firstName, lastName.toUpperCase()].join(" ");

console.log(newName);//Mr. Joaquin CAGGIANO

Le pasamos un guion:

const newName = ["Mr.", firstName, lastName.toUpperCase()].join("-");

console.log(newName);//Mr.-Joaquin-CAGGIANO

Función para agregar mayúsculas a un nombre:

const capitalizeName = function(name){

  const names = name.split(" ");

  const namesUpper = [];

  for (const n of names) {

    // namesUpper.push(n[0].toUpperCase() + n.slice(1))

    namesUpper.push(n.replace(n[0], n[0].toUpperCase()))

  }

  console.log(namesUpper.join(" ")); //Joaquin Caggiano

}

capitalizeName("joaquin caggiano")

**.padStart():** sirve para agregar caracteres al principio de un string. El primer parámetro es la cantidad de caracteres que queremos agregar, y el segundo el carácter que vamos a agregar.

**.padEnd():** igual que el anterior pero agrega al final.

const message = "Joaquin";

console.log(message.padStart(20, "-").padEnd(25, "+"));

//-------------Joaquin+++++

Función para que se vean solo los 4 últimos dígitos de una tarjeta:

const maskCreditCard = function (number) {

  const str = number + "";//esto transforma el numero en un string;

  const last = str.slice(-4);

  return last.padStart(str.length, "\*");

}

console.log(maskCreditCard(134256457658));//\*\*\*\*\*\*\*\*7658

**.repeat():** nos permite repetir el mismo string múltiples veces. Le pasamos como argumento la cantidad de veces que queremos que se repita.

const messageAirPlane = "Todos los vuelos estan demorados.... ";

console.log(messageAirPlane.repeat(3)); //se repite el mensaje 3 veces

**MÓDULO 10: A CLOSER LOOK AT FUNCTIONS**

**DEFAULT PARAMETERS**

Los default values se pasan directamente en los parámetros de la function, y pueden contener cualquier expresión.

const bookings = [];

const createBooking = function (flightNum, numPassengers = 1, price = 199 \* numPassengers) {

    // Como se hacia antes de la nueva versión de JS

//     numPassengers = numPassengers || 1;

//     price = price || 199;

    const booking = {

        flightNum,

        numPassengers,

        price

    }

    console.log(booking);

    bookings.push(booking)

};

createBooking("LH123");//{ flightNum: 'LH123', numPassengers: 1, price: 199 }

createBooking("LH145", 3, 1000);//{ flightNum: 'LH145', numPassengers: 3, price: 1000 }

createBooking("HR356", 4)//{ flightNum: 'HR356', numPassengers: 4, price: 796 }

**FIRST-CLASS and HIGHER-ORDER FUNCTIONS**

First class functions:

* Javascript trata a las funciones como first-class citizens.
* Esto significa que son simplemente valores.
* Las funciones son sólo otro tipo de objetos.
* Al ser valores las podemos almacenar en variables o en propiedades de objetos.
* También podemos pasar funciones como argumentos de otras funciones.
* Podemos devolver una función desde otra función.
* Podemos llamar métodos en funciones.bind()

Higher order functions:

* Son funciones que reciben otra función como argumento y retornan una nueva función, o ambas.
* Esto es posible debido a las First class functions.

**FUNCTIONS ACCEPTING CALLBACKS FUNCTIONS**

const oneWord = function (str) {

    return str.replace(/ /g, "").toLowerCase();

}

const upperFirstWord = function (str) {

    const [first, ...others] = str.split(" ");

    return [first.toUpperCase(), ...others].join(" ");}

// Higher-order functions

const transformer = function (str, fn) {

    console.log(`Original string: ${str}`);//Original string: JavaScript is the best!

    console.log(`Transformed string: ${fn(str)}`);//Transformed string: JAVASCRIPT is the best!

    console.log(`Transformed by: ${fn.name}`);//Transformed by: upperFirstWord

}

transformer("JavaScript is the best!", upperFirstWord);

// Higher-order functions

const transformer = function (str, fn) {

    console.log(`Original string: ${str}`);//Original string: JavaScript is the best!

    console.log(`Transformed string: ${fn(str)}`);//Transformed string: javascriptisthebest!

    console.log(`Transformed by: ${fn.name}`);//Transformed by: oneWord

}

transformer("JavaScript is the best!", oneWord);

**FUNCTIONS RETURNING FUNCTIONS**

const greet = function (greeting) {

    return function (name) {

        console.log(`${greeting} ${name}`); //Hey Joaquín

    }

}

const greeterHey = greet("Hey");

greeterHey("Joaquín")

greet("Hello")("Joaquín")//Hello Joaquín

// Arrow function

// como lo resolví yo

const greet = (greeting) => {

    const arrowGreet = (name) => {

        console.log(`${greeting} ${name}`);

    }

    return arrowGreet

}

greet("Hola")("Mauro")

// Como lo resolvió Jonas

const greetArr = greeting => name => console.log(`${greeting} ${name}`);

greetArr("Hi")("Jonas")

**THE CALL AND APPLY METHODS**

const lufthansa = {

    airline: "Lufthansa",

    iataCode: "LH",

    bookings: [],

    book(flightNum, name){

        console.log(`${name} booked a seat on ${this.airline} flight ${this.iataCode}${flightNum}`);

        this.bookings.push({flight: `${this.iataCode}${flightNum}`, name})

    }

}

lufthansa.book(239, "Joaquín Caggiano");//Joaquín Caggiano booked a seat on Lufthansa flight LH239

lufthansa.book(639, "Alumine Llado");//Alumine Llado booked a seat on Lufthansa flight LH639

console.log(lufthansa);

const eurowings = {

    airline: "Eurowings",

    iataCode: "EW",

    bookings: []

}

const book = lufthansa.book;

// Does not work

// book(23, "Sarah Williams") no funciona porque no encuentra la palabra clave this

**CALL METHOD:**

**.call(**arg1, arg2**):** el primer argumento que le pasamos es al que queremos que apunte la palabra clave **this**, el segundo argumento son los argumentos normales de la función.

Este método nos permite establecer la palabra clave **this** en cualquier función que queramos llamar.

book.call(eurowings, 23, "Sarah Williams");

book.call(lufthansa, 543, "Mauro Mazzieri");

**APPLY METHOD:** no se usa más en el moderno JavaScript.

**.apply():** hace exactamente lo mismo, pero la única diferencia es que **apply** no recibe una lista de argumentos después de la palabra clave **this**. En cambio, va a recibir un array de argumentos.

**Esto no se usa más…**

const flightData = [485, "Indiana Jones"];

book.apply(eurowings, flightData);

console.log(eurowings);

**Porque se puede hacer esto:**

book.call(eurowings, ...flightData);

**THE BIND METHOD**

Nos permite al igual que el método **call** establecer manualmente la palabra clave **this** en cualquier llamado a una función.

La diferencia es que **bind** no llama inmediatamente a la función. En cambio, devuelve una nueva función, donde la palabra clave **this** está vinculada. Por lo tanto, se establece en cualquier valor que le pasemos al **bind**.

const bookEW = book.bind(eurowings);

bookEW(666, "Dario Barassi");

console.log(eurowings);//{ flight: 'EW666', name: 'Dario Barassi' }

// Podemos establecer parámetros antes

const bookEW666 = book.bind(eurowings, 666);

bookEW666("Goku");

console.log(eurowings);//{ flight: 'EW666', name: 'Goku' }

**Objetos junto con detectores de eventos:**

lufthansa.planes = 300;

lufthansa.buyPlane = function () {

    console.log(this);

    this.planes++

    console.log(this.planes);

}

document.querySelector(".buy").addEventListener("click", lufthansa.buyPlane.bind(lufthansa))

**Partial applications:**

const addTax = (rate, value) => value + value \* rate;

const addVAT = addTax.bind(null, 0.23);

// addVat = value => value + value \* 0.23

Retornando una function dentro de otra function:

const addTaxRate = function(rate){

    return function (value) {

        return value + value \* rate

    }

};

const addVAT2 = addTaxRate(0.23);

console.log(addVAT2(100));//123

console.log(addVAT2(23));//28.29

**INMEDIATLY INVOKED FUNCTIONS EXPRESSIONS (IIFE)**

Hay veces que queremos que una function solo funcione una sola vez. Debemos convertir la function en una expresión poniendola entre paréntesis() y ejecutarla().

(function(){

    console.log("This function will never run again");

})();

// con arrow function también funciona

(() => console.log("This function will ALSO never run again"))()

Una de las razones por las cuales se usaban estas funciones inmediatamente invocadas era para proteger variables que no queríamos que se sobrescribieran, haciéndolas privadas por el scope.

Pero esto ya no se usa más ya que podemos crear un bloque de código usando las llaves:

{

    const privateData = "Joaquín"

}

console.log(privateData);//ReferenceError: privateData is not defined

**CLOSURES**

Un **closure** hace que una function recuerde todas las variables que existían en el lugar de nacimiento de la function.

Una function siempre tiene acceso al entorno de las variables del contexto de ejecución en el cual fue creado, inclusive después de que ese contexto de ejecución se haya ido.

El **closure** es básicamente este entorno de variables adjunto a la función, exactamente como estaba en el momento y lugar en que la función se creó.

const secureBooking = function () {

    let passengerCount = 0;

    return function () {

        passengerCount++;

        console.log(passengerCount);

    }

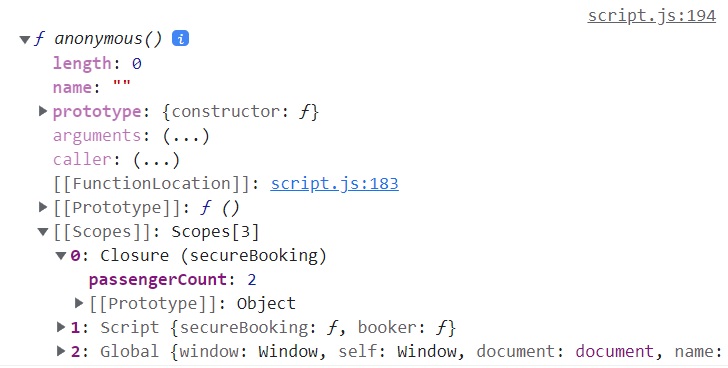
};

const booker = secureBooking();

booker();

booker();

console.dir(booker);



**Example 1:**

let f;

const g = function () {

    const a = 23;

    f = function () {

        console.log(a \* 2);

    }

}

const h = function () {

    const b = 4;

    f = function () {

        console.log(b \* 3);

    }

}

g();

f();

console.dir(f);//46 y en el scope, el closure es "a: 23"

h();

f();

console.dir(f);//12 y en el scope, el closure es "b: 4"

**Example 2:** El closure tiene prioridad sobre el scope global. Si tuviéramos una variable perGroup global de igual manera usaría la que se encuentra dentro de la function boardPassengers.

const boardPassengers = function (n, wait) {

    const perGroup = n / 3;

    setTimeout(function () {

        console.log(`We are now boarding all ${n} passengers`);

        console.log(`There are 3 groups, each with ${perGroup} passengers`);

    }, wait \* 1000);

    console.log(`Will start boarding in ${wait} seconds`);

}

boardPassengers(180, 3);

**SECTION 11: WORKING WITH ARRAYS**

**SIMPLE ARRAYS METHODS**

const arr = ["a", "b", "c", "d", "e"];

**.slice()**: podemos extraer una parte del array sin modificar el array original. Devuelve un nuevo array, sin modificar el original. Dentro del paréntesis debemos pasarle el inicio del index donde empieza a buscar, también podemos pasarle el parámetro donde queremos que termine, o pasar el index en negativo para empezar desde atrás para adelante.

console.log(arr.slice(2));// [ 'c', 'd', 'e' ]

console.log(arr.slice(2,4));// [ 'c', 'd' ]

console.log(arr.slice(-2));// [ 'd', 'e' ]

console.log(arr.slice(1, -2));// [ 'b', 'c' ]

console.log(arr.slice());// estamos copiando el array original

**.splice()**: funciona casi igual que el slice, pero la diferencia es que el método splice modifica el array original.

console.log(arr.splice(2));// [ 'c', 'd', 'e' ]

console.log(arr);// [ 'a', 'b' ]

**.reverse()**: devuelve el array de atrás para adelante. Este método modifica el array original.

console.log(arr.reverse());// [ 'e', 'd', 'c', 'b', 'a' ]

console.log(arr); // [ 'e', 'd', 'c', 'b', 'a' ]

**.concat()**: se utilizar para concatenar dos arrays. No modifica el array original.

const arr2 = ["f", "j", "h", "i", "j"];

const letters = arr.concat(arr2);

console.log(letters); // ['a', 'b', 'c', 'd','e', 'f', 'j', 'h','i', 'j']

console.log([...arr, ...arr2]);// ['a', 'b', 'c', 'd','e', 'f', 'j', 'h','i', 'j'] En este caso no mutamos ningun array

**.join()**: Lo utilizamos para separar el contenido de un array como queramos.

console.log(arr.join(" - ")); // a - b - c - d - e

**THE NEW AT METHOD**

const arr = [23, 13, 45];

console.log(arr[0]);// 23 - Como se hacía antes

console.log(arr.at(0)); // 23 -Como se hace ahora

// Conseguir el último elemento

console.log(arr[arr.length -1]); //45

console.log(arr.slice(-1)[0]);// 45

console.log(arr.at(-1)); //45

**LOOPING ARRAYS**

**.forEach()**: dentro de los paréntesis podemos pasarle en orden: el elemento actual, el índice actual, y por último el array completo que estamos lopeando.

Prestar atención que en el **for of** si queremos tener el índice actual debemos ponerlo primero entre los corchetes y utilizar el método **entries()**. En cambio, en el **forEach** el índice actual va como segundo argumento de la function.

const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

// Con for of

for(const [i, movement] of movements.entries()){

    if (movement > 0) {

        console.log(`Movement ${i + 1}: You deposited ${movement}`);

    } else {

        console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);// Math.abs devuelve el valor absoluto de un número

    }

}

// con forEach

movements.forEach((movement, i, arr) => {

    if (movement > 0) {

        console.log(`Movement ${i + 1}: You deposited ${movement}`);

    } else {

        console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);// Math.abs devuelve el valor absoluto de un número

    }

})

**FOREACH for MAPS and SETS**

**MAPS:** cuando hacemos un forEach de un Map, el primero parámetro es el valor actual, el segundo es la key, y el tercero es el Map completo.

const currencies = new Map([

    ['USD', 'United States dollar'],

    ['EUR', 'Euro'],

    ['GBP', 'Pound sterling'],

]);

currencies.forEach(function(value, key, map){

    console.log(`${key}: ${value}`);

})

**SETS:** un set no tiene keys ni índices, por lo que el segundo parámetro no tendría mucho sentido, por lo tanto, utilizamos un “\_” que significa que es una variable desechable.

const currenciesUnique = new Set(["USD", "GBP", "USD", "EUR", "EUR"]);

console.log(currenciesUnique);

currenciesUnique.forEach(function (value, \_, map) {

    console.log(`${value}: ${value}`);

})

**CREATING DOM ELEMENTS**

**.insertAdjacentHTML():** Este método acepta dos strings. El **primer string** es la posición en la queremos adjuntar el HTML, ejemplo: beforebegin, afterbegin, beforeend, afterend. El **segundo string** es el HTML que queremos insertar.

const account1 = {

  owner: 'Jonas Schmedtmann',

  movements: [200, 450, -400, 3000, -650, -130, 70, 1300],

  interestRate: 1.2, // %

  pin: 1111,

};

const containerMovements = document.querySelector('.movements');

const displayMovements = function (movements) {

    containerMovements.innerHTML = "";//para vaciar todo lo que había antes y empezar de 0.

    movements.forEach((mov, i) => {

        const type = mov > 0 ? "deposit" : "withdrawal"

        const html = `

        <div class="movements\_\_row">

          <div class="movements\_\_type movements\_\_type--${type}">${i + 1} ${type}</div>

          <div class="movements\_\_value">${mov}</div>

        </div>

        `

        containerMovements.insertAdjacentHTML("afterbegin", html)

    })

}

displayMovements(account1.movements)

**THE MAP METHOD**

El método map nos dará un nuevo array, el cual contendrá en cada posición los resultados de aplicar un callback function a los elementos originales del array.

A diferencia del forEach que mostrábamos el resultado en consola, en el método map necesitamos hacer un return.

Como en el forEach también podemos pasarle como parámetros el elemento actual, el índice actual y el array completo, en ese orden.

const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

const eurToUsd = 1.1;

const movementsUSD = movements.map((mov) => {

  return mov \* eurToUsd

});

console.log(movements);

console.log(movementsUSD);

**COMPUTING USERNAMES**

const account1 = {

  owner: 'Jonas Schmedtmann',

  movements: [200, 450, -400, 3000, -650, -130, 70, 1300],

  interestRate: 1.2, // %

  pin: 1111,

};//Hay más accounts

const accounts = [account1, account2, account3, account4];

const createUserNames = function (accs) {

    accs.forEach((acc) => {

        acc.userName = acc.owner

            .toLowerCase()

            .split(" ")

            .map((name) => name[0])

            .join("");

    });

};

createUserNames(accounts);

console.log(accounts);

A cada account ahora se le agrego la key userName con el value de las primeras iniciales de los nombres.

**THE FILTER METHOD**

Se utiliza para filtrar elementos que satisfacen una determinada condición a través de un callback.

const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

const deposits = movements.filter(function (mov) {

    return mov > 0

})

console.log(deposits);//[200, 450, 3000, 70, 1300]

Si lo hacemos con un **for of** necesitamos pushear el elemento en un array.

// con un for of

const depositsFor = []

for (const mov of movements) {

    if (mov > 0) {

        depositsFor.push(mov)

    }

}

console.log(depositsFor);//[200, 450, 3000, 70, 1300]

**THE REDUCE METHOD**

Utilizamos este método para reducir todos los elementos de un array a un solo valor.

A diferencia de los métodos map y forEach, el primer parámetro del callback en el método **reduce** es el **acumulador**, y luego el elemento actual, el índice, y si se necesita el array completo.

Aparte de pasar un callback, es necesario decir cual es el valor inicial del **acumulador**.

const balance = movements.reduce((acum, elemCurrent, i) => {

    // console.log(`Iteration ${i}: ${acum}`);

    return acum + elemCurrent

}, 0);

console.log(balance);//3840

Podemos utilizar el método reduce para ver el máximo valor de un array:

// //Maximum value

const max = movements.reduce((acc, mov) => {

    if (acc > mov){

        return acc

    } else {

        return mov

    }

}, movements[0]);

console.log(max);//3000

**THE MAGIC OF CHAINING METHODS**

const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

const eurToUsd = 1.1;

const totalDepositsUSD = movements

  .filter(mov => mov > 0)

  .map(mov => mov \* eurToUsd)

  .reduce((acum, mov) => acum + mov, 0);

console.log(totalDepositsUSD);//5522.000000000001

Solo podemos seguir encadenando métodos de array mientras devuelvan un nuevo array. Como el reduce devuelve un solo valor, no podríamos continuar la cadena.

**THE FIND METHOD**

El método **find** recupera un elemento del array. Devolverá el primer elemento que cumpla la condición que le asignamos.

La diferencia entre el método **filter** y **find**, es que filter devuelve todos los elementos que coinciden con la condición, mientras que el método find solo devuelve el primer elemento que cumpla la condición. Y la segunda diferencia es que el método filter devuelve un nuevo array, mientras que el find solo devuelve el elemento en sí.

const account = accounts.find(acc => acc.owner === "Jessica Davis");

console.log(account);//me devuelve el objeto que tiene ese nombre en la propiedad owner

**THE FINDINDEX METHOD**

Retorna el index del elemento encontrado.

btnClose.addEventListener('click', (e) => {

  e.preventDefault();

  if (

    inputCloseUsername.value === currentAccount.userName &&

    Number(inputClosePin.value) === currentAccount.pin

  ) {

    const index = accounts.findIndex((acc) => {

      return acc.userName === currentAccount.userName

    })

    //Delete account

    // console.log(index);

    accounts.splice(index, 1);

    //Hide UI

    containerApp.style.opacity = 0;

  }

  //Ponemos en blanco los campos otra vez

  inputCloseUsername.value = inputClosePin.value = '';

});

**THE SOME and EVERY METHOD**

El método **some** devuelve true o false dependiendo si se cumple la condición que apliquemos:

const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

const someMethod = movements.some((mov) => mov > 0);

console.log(someMethod);//true

Mientras que el método **every** solo devuelve true si todos los elementos cumplen la condición:

const everyMethod = movements.every((mov) => mov > 0);

console.log(everyMethod);//false

**THE FLAT AND FLATMAP METHOD**

Los utilizamos cuando tenemos que trabajar con un array que tiene anidado más arrays dentro. Lo que hacen estos métodos es desanidarlos.

En el método **flat** podemos pasarle como argumento cuantos niveles queremos que desanide, en cambio en el **flatMap** solamente desanida un nivel.

El método **flatMap** es la combinación de los métodos flat y map.

const arr = [[1, 2, 3], [4, 5, 6], 7, 8];

console.log(arr.flat()); //[1, 2, 3, 4, 5, 6, 7, 8]

const arrDeep = [[[1, 2], 3], [4, [5, 6]], 7, 8];

//Le pasamos como argumento 2 para decir que vamos 2 niveles más profundo en el array

console.log(arrDeep.flat(2)); //[1, 2, 3, 4, 5, 6, 7, 8]

const overalBalance = accounts

  .map(acc => acc.movements)

  .flatMap()

  .reduce((acc, mov) => acc + mov, 0);

console.log(overalBalance);

**SORTING ARRAY**

El método **sort** realiza el ordenamiento de los elementos de un array, y devuelve el array original pero modificado. Este método **solo** **funciona** con strings, a menos que le pasemos un **callback function** como parámetro al método **sort**.

Este callback function toma dos parámetros:

* El primero es el **valor actual**.
* El segundo es el **siguiente valor**.

// Strings

const owners = ["Jonas", "Zach", "Adam", "Martha"];

console.log(owners.sort());//["Adam", "Jonas", "Martha", "Zach"]

console.log(owners);//["Adam", "Jonas", "Martha", "Zach"]

//Numbers

console.log(movements);//[200, 450, -400, 3000, -650, -130, 70, 1300]

//return < 0: a, b (keep order)

// return > 0: b, a (switch order)

movements.sort((a, b) => {

  if(a > b){

    return 1;

  }

  if (b > a) {

    return -1;

  }

});

console.log(movements);//[-650, -400, -130, 70, 200, 450, 1300, 3000]

**MORE WAYS OF CREATING A FILLING ARRAYS**

**Fill Method:**

Si usamos el constructor **new Array()** y le pasamos un solo elemento, lo tomará como la cantidad de elementos que va a tener el array, pero estará vacío:

const x = new Array(7);

console.log(x);//[empty × 7]

A este array solamente le podemos aplicar el método **fill**:

x.fill(1);

console.log(x);//[1, 1, 1, 1, 1, 1, 1]

También podemos aplicar el método fill en arrays con datos, en este caso le decimos llenar el 23 desde el índice 2 al 6 (el 6 no incluido).

const arr = [1,2,3,4,5,6,7];

arr.fill(23, 2, 6);

console.log(arr);//[1, 2, 23, 23, 23, 23, 7]

**Array.from():**

El **primer argumento** es la longitud del array, y el **segundo** es una function map.

const y = Array.from({length: 7}, () => 1);

console.log(y);//[1, 1, 1, 1, 1, 1, 1]

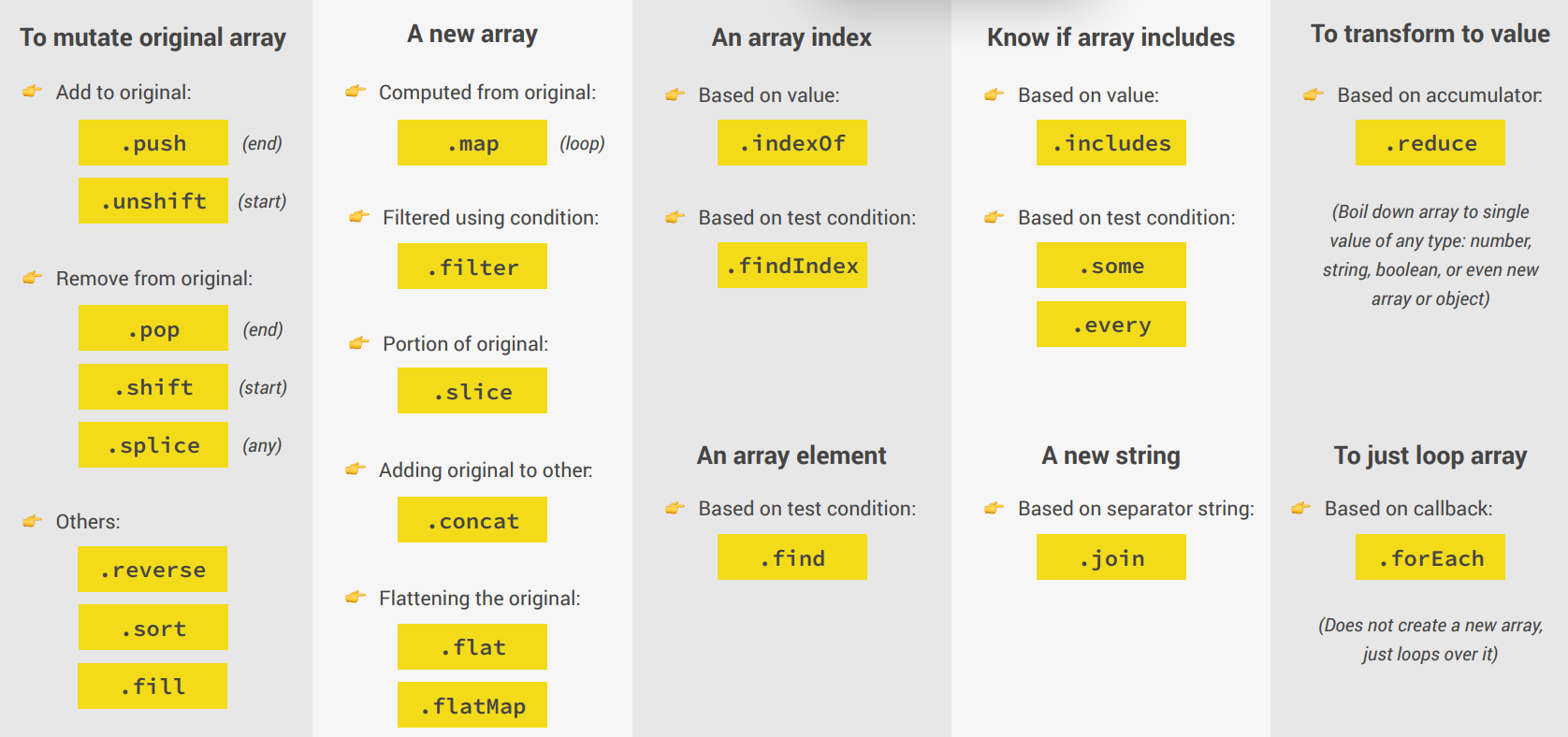
const z = Array.from({length: 7}, (\_, i) => i + 1);

console.log(z);//[1, 2, 3, 4, 5, 6, 7]

const random = Array.from({length: 100}, (\_, i) => Math.trunc(Math.random() \* 100) + i);

console.log(random);//100 números random

**WHICH ARRAY METHOD TO USE?**



**SEC-12: NUMBERS, DATES, INTL and TIMERS**

**CONVERTING and CHECKING NUMBERS**

**Formas de convertir un string en un número:**

Converting:

// Converting

console.log(Number("23"));

console.log(+"23");

Parsing:

Podemos utilizar el método **parseInt()** del objeto **Number**, el cual recibe como primer parámetro un string, y como segundo parámetro el tipo de sistema numeral que utilicemos (decimal 10, o binario 2):

// Parsing

console.log(Number.parseInt("30px", 10));//30

console.log(Number.parseInt("30px", 2));//NaN

console.log(Number.parseInt("e23", 10));//NaN - debe empezar con un numero

También tenemos el **parseFloat()** el cual nos dará los decimales:

console.log(Number.parseFloat("2.5rem"));//2.5

Otro método que tiene es **isNaN()**, el cual sirve para chequear si algún valor es un número:

console.log(Number.isNaN(20)); //false

console.log(Number.isNaN("20")); //false

console.log(Number.isNaN(+"20X")); //true

console.log(Number.isNaN(23/0)); //false

**isFinite()** se utiliza más que isNaN para chequear si un valor es un número:

console.log(Number.isFinite(20));//true

console.log(Number.isFinite("20"));//false

**MATH and ROUNDING**

**Math.sqrt:** raíz cuadrada.

console.log(Math.sqrt(25)); //5

console.log(25 \*\* (1 / 2)); //5

**Max y min:** Máximo y Mínimo valor.

console.log(Math.max(5, 23, 5, 8, 14));//23

console.log(Math.min(5, 23, 5, 8, 14));//5

**Math.PI:** Calcular el área de un círculo.

console.log(Math.PI \* Number.parseFloat("10px") \*\* 2);//314.1592653589793

**Math.random():** nos da un número en 0 y 1, y con el Math.trunc le quitamos los decimales.

console.log(Math.trunc(Math.random() \* 6) + 1);//nos da un numero entre 1 y 6

Función para obtener números random:

const randomInt = (min, max) => {

  return Math.floor(Math.random() \* (max - min) + 1) + min;

}

**Rounding integers**

**Math.trunc():** redondea un número quitándole el decimal.

console.log(Math.trunc(23.3));//23

**Math.round():** redondea un número a su entero más cercano.

console.log(Math.round(23.3));//23

console.log(Math.round(23.9));//24

**Math.ceil():** redondea para arriba.

console.log(Math.ceil(23.3));//24 - redondea para arriba

console.log(Math.ceil(23.9));//24

**Math.floor():** redondea para abajo.

console.log(Math.floor(23.3));//23 - redondea para abajo

console.log(Math.floor(23.9));//23

**Rounding decimals**

**toFixed():** retorna siempre un string y no un número. Y le pasamos como argumento cuantos decimales queremos.

console.log((2.7).toFixed(0)); //3

console.log((2.7).toFixed(3)); //2.700

console.log((2.345).toFixed(0)); //2.35

**THE REMAINDER OPERATOR**

Devuelve el resto de una división. Se utiliza para saber si el resultado de un número es par o impar:

console.log(5 % 2);// 1

console.log(5 / 2);// % = 2 \* 2 + 1

console.log(8 % 3);// 2

console.log(8 / 3);// 8 = 3 \* 2 + 2

console.log(6 % 2);// 0

console.log(6 / 2);// 3

console.log(7 % 2);// 1

console.log(7 / 2);//3.5

**Función para saber si es par o impar:**

const isEven = (n) => n % 2 === 0;

**NUMERIC SEPARATOR**

Utilizamos el guion bajo (**\_**) para separar un número largo, en vez de comas (,) como haríamos normalmente:

const diameterSolar = 287\_460\_000\_000;

Hay que tener en cuenta que, si queremos convertir un string en números, y este número tiene guion bajo, no nos resultará:

console.log(Number("230\_000"));//NaN

**WORKING with BIGINT**

Este es el número máximo que podemos utilizar:

console.log(Number.MAX\_SAFE\_INTEGER);//9007199254740991

Con los **BigInt** podemos almacenar un número tan largo como queramos, para ello al final del número escribimos la letra “**n**”, o también con la function **BigInt()**:

console.log(92373278567832563465843675346n);

console.log(BigInt(92373278567832563465843675346));

No podemos mezclar en operaciones matemáticas números normales con bigints, por lo tanto, hay que convertirlo primero:

const huge = 785236577892345897n;

const num = 14;

console.log(huge \* BigInt(num));

En las divisiones devuelve un número entero más cercano:

console.log(10n / 3n);// 3n

**CREATING DATES**

Hay 4 formas de crear fechas en Javascript:

**Primera forma:**

const now = new Date();

console.log(now);//Mon Jul 04 2022 11:11:01

**Segunda forma:** parseando desde un string.

console.log(new Date("december 24, 2015"));//Thu Dec 24 2015 00:00:00

**Tercera forma:** pasando en números el año, mes, dia, hora, minutos y segundos.

console.log(new Date(2037, 10, 19, 15, 23, 5));//Thu Nov 19 2037 15:23:05

**Cuarta forma:** le pasamos cuantos días queremos, multiplicando por 24 horas, por 60 minutos, por 60 segundos, por 1000 milisegundos.

console.log(new Date(0));//Wed Dec 31 1969 21:00:00

console.log(new Date(3 \* 24 \* 60 \* 60 \* 1000));//Sat Jan 03 1970 21:00:00

**Working with Dates**

const future = new Date(2037, 10, 19, 15, 23);

console.log(future);//Thu Nov 19 2037 15:23:00

**getFullYear:**

console.log(future.getFullYear());//2037

**getMonth:**

console.log(future.getMonth());//10

**getDate:**

console.log(future.getDate());//19

**getDay:** es el día de la semana.

console.log(future.getDay());//4-Jueves

**getHours, getMinutes, getSeconds:**

console.log(future.getHours()); //15

console.log(future.getMinutes()); //23

console.log(future.getSeconds()); //0

**toISOString:**

console.log(future.toISOString());// 2037-11-19T18:23:00.000Z

**getTime:** muestra los milisegundos desde 1969 hasta ahora. El timestamp.

console.log(future.getTime());// 2142267780000

console.log(new Date(2142267780000));// Thu Nov 19 2037 15:23:00

**Date.now():** nos da el timestamp actual.

console.log(Date.now());// 1656945869947

console.log(new Date(1656945869947));// Mon Jul 04 2022 11:44:29

**setFullYear:** método para setear el año que queramos. También podemos setear mes, día, etc.

future.setFullYear(2040);

console.log(future);// Mon Nov 19 2040 15:23:00

**OPERATIONS with DATES**

const future = new Date(2037, 10, 19, 15, 23);

console.log(+future);//2142267780000

const calcDaysPassed = (date1, date2) => {

  return Math.abs(date2 - date1) / (1000 \* 60 \* 60 \* 24);

};

const day1 = calcDaysPassed(new Date(2037, 3, 14), new Date(2037, 3, 4));

console.log(day1);//10

**INTERNATIONALIZING DATES (Intl)**

const now = new Date();

const options = {

  hour: "numeric",

  minute: "numeric",

  day: "numeric",

  month: "long",//2-digit

  year: "numeric",//o tambien 2-digit

  weekday: "long"//short - narrow

}

const locale = navigator.language;

// console.log(locale);

labelDate.textContent = new Intl.DateTimeFormat(locale, options).format(now);

* **now:** es la fecha actual.
* **options:** formateamos que datos de la fecha queremos.
* **locale:** nos da el formato de lenguaje que tenemos en nuestra PC.

**INTERNATIONALIZING NUMBERS (Intl)**

const num = 3884764.23;

const options = {

  style: "currency",//percent / unit / currency

  // unit: "mile-per-hour",//celsius / mile-per-hour

  currency: "EUR",

  // useGrouping: false

}

console.log('US:', new Intl.NumberFormat('en-US', options).format(num)); //US: €3,884,764.23

console.log('Germany:', new Intl.NumberFormat('de-DE', options).format(num)); //Germany: 3.884.764,23 €

console.log('Browser:', new Intl.NumberFormat(navigator.language, options).format(num)); //Browser: 3.884.764,23 €

**TIMERS: setTimeout and setInterval**

**setTimeout**: corre solamente una vez después de haber definido el tiempo.

const ingredients = ['olives', 'spinach'];

const pizzaTimer = setTimeout(

  (ing1, ing2) => {

    console.log(`Here is your pizza with ${ing1} and ${ing2}`);

  },

  2000,

  ...ingredients

);

console.log('Waiting for the pizza...');

if(ingredients.includes("spinach")) {

  clearTimeout(pizzaTimer);

}//No se va a mostrar el console.log de pizzaTimer porque contiene spinach

**setInterval:** se mantiene corriendo básicamente por siempre, hasta que nosotros lo paremos.

setInterval(() => {

  const now = new Date();

  const hour = `${now.getHours()}`.padStart(2, 0);

  const min = `${now.getMinutes()}`.padStart(2, 0);

  const seconds = `${now.getSeconds()}`.padStart(2, 0);

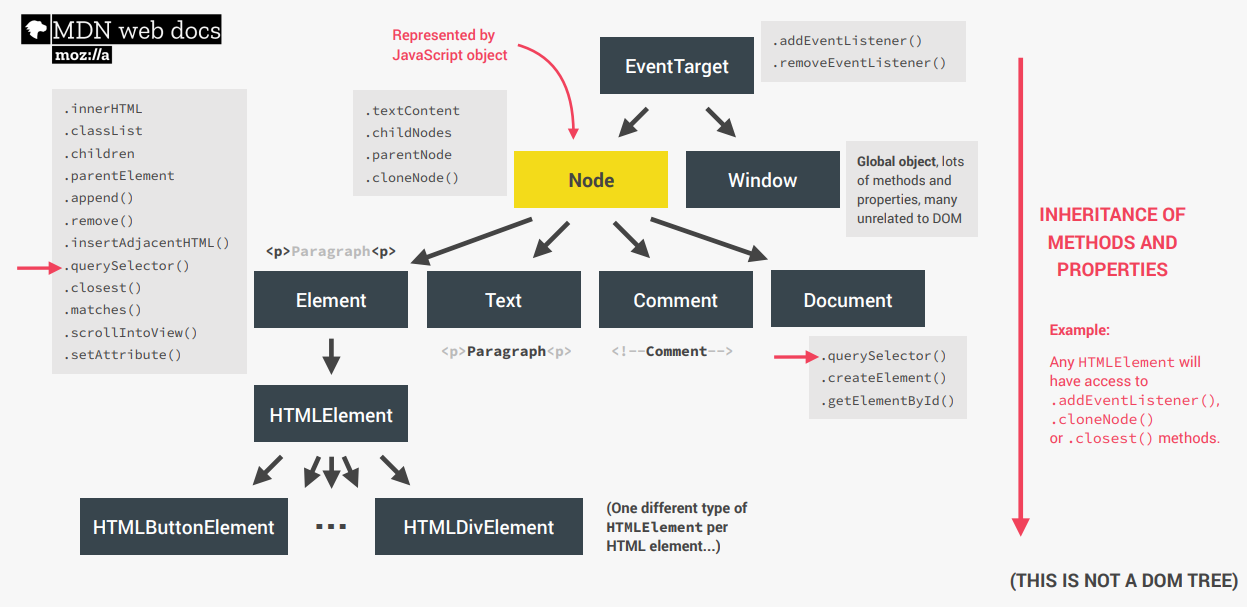
  console.log(`${hour}:${min}:${seconds}`);

}, 1000)

**SEC-13: ADVANCED DOM and EVENTS**

**HOW THE DOM REALLY WORKS**

* Nos permite hacer interacciones de Javascript con el browser.
* Podemos escribir JS para crear, modificar y eliminar elementos HTML; setar estilos, clases y atributos; y escuchar y responder eventos.
* El árbol DOM es generado desde un documento HTML, el cual podemos interactuar con él.
* El DOM es una API compleja que contiene muchos métodos y propiedades para interactuar con el árbol DOM.



**SELECTING, CREATING and DELETING ELEMENTS**

**Selecting Elements:**

Al seleccionar elementos con **getElementsByTagName** obtenemos un HTMLCollection de todos los elementos que le especifiquemos (en este caso buttons), el cual se modifica si borramos algún de estos elementos. Esto no pasa con **querySelectorAll**, si borramos un elemento, de igual manera nos seguirá apareciendo en el **nodeList**.

const allSections = document.querySelectorAll(".section");

// console.log(allSections);

const allButtons = document.getElementsByTagName("button");

// console.log(allButtons);

Podemos seleccionar elementos por sus clases también, y nos devolverá un HTMLColecction:

console.log(document.getElementsByClassName("btn"));

**Creating Element:**

Utilizamos **createElement()**, el cual recibe como parámetro el elemento HTML que queremos crear:

const message = document.createElement("div");

**Inserting Elements:**

Luego con el método **prepend()** podemos insertar este mensaje como el primer elemento del header por ejemplo:

message.classList.add('cookie-message');

message.innerHTML =

  "We use cookied for improved funcionality and analytics. <button class='btn btn--close-cookie'>Got it!</button>";

header.prepend(message);

En cambio, si lo queremos insertar en el último elemento utilizamos **append()**.

header.append(message);

Ahora, si queremos que aparezca como primer y último elemento a la vez, debemos hacer una copia de este elemento:

header.append(message.cloneNode(true));

También Podemos insertarlo **antes** o **después** de un elemento:

**Before ():**

header.before(message);

**After ():**

header.after(message);

**Deleting Elements:**

Utilizamos el método **remove()**.

document.querySelector(".btn--close-cookie").addEventListener("click", () => {

  message.remove();

  // Antes se hacía así para remover un elemento:

  // message.parentElement.removeChild(message);

})

**STYLES, ATTRIBUTES and CLASSES**

**Styles:**

Si queremos buscar el valor de un estilo en particular de un elemento podemos conseguirlo a través de **getComputedStyle()**. Le debemos pasar como parámetro el elemento al que le queremos ver sus estilos, y también le podemos pedir un estilo en específico:

console.log(getComputedStyle(message));//todos los estilos que tiene

console.log(getComputedStyle(message).color);//rgb(187, 187, 187)

message.style.height = Number.parseFloat(getComputedStyle(message).height, 10) + 40 + "px";

Podemos setear un style de una propiedad de css con **setProperty**:

document.documentElement.style.setProperty("--color-primary", "orangered");

**Attributes:**

Elemento HTML:

<img

   src="img/logo.png"

   alt="Bankist logo"

   class="nav\_\_logo"

   id="logo"

   designer = "Joaquin"

data-version-number = "3.0"

/>

Para leer atributos de un elemento HTML que es reconocido por JS, utilizamos el nombre de este atributo después del elemento seleccionado:

const logo = document.querySelector(".nav\_\_logo");

console.log(logo.alt);//Bankist logo

console.log(logo.src);//ubicación de la imagen

console.log(logo.className);//nav\_\_logo

Si el atributo no es reconocido por JS porque lo diseñamos nosotros mismo debemos usar el método **getAttribute()** y le pasamos como parámetro el atributo que escribimos:

console.log(logo.getAttribute("designer"));//Joaquin

console.log(logo.getAttribute("src"));//img/logo.png

Podemos setear estos atributos:

logo.alt = "Beautiful minimalist logo"

console.log(logo.alt);//Beautiful minimalist logo

ó

logo.setAttribute("company", "Bankist");

Podemos utilizar **data attributes** los cuales se almacenan en **dataset**:

console.log(logo.dataset.versionNumber);//3.0

**Classes:**

Tenemos 4 métodos: **add()** para agregar una clase, **remove()** para eliminar una clase, **toggle()** para activar y desactivar la clase, y **contains()** para verificar si un elemento tiene cierta clase.

logo.classList.add();

logo.classList.remove();

logo.classList.toggle();

logo.classList.contains();

**IMPLEMENTIN SMOOTH SCROLLING**

Con el método **getBoundingClientRect()** obtenemos datos de un elemento, como las coordenadas, Width, heght, etc.

OLD WAY:

Para scrollear desde un botón al elemento elegido utilizamos el método **window.scrollTo()**, el cual recibe como parámetro un **objeto**, el cual el primer key es el **left** value, la segunda key es el **top** ( A estos dos valores para que siempre funcione le podemos sumar el eje **x** e **y** del viewport con **window.pageXOffset** y **window.pageYOffset** ) , y la tercera key es **behavior**, la cual puede recibir **smooth**.

NEW WAY:

Elegimos el elemento al que nos queremos desplazar y utilizamos el método **scrollIntoView()**, al cual le pasamos un objeto con la key **behavior** y el valor **smooth**.

const btnScrollTo = document.querySelector('.btn--scroll-to');

const section1 = document.querySelector('#section--1');

btnScrollTo.addEventListener('click', e => {

  const s1coords = section1.getBoundingClientRect();

  // console.log(s1coords);

  // console.log(e.target.getBoundingClientRect());

  // console.log("Current scroll (X/Y)", window.pageXOffset, window.pageYOffset);

  // console.log(

  //   'height/width viewport',

  //   document.documentElement.clientHeight,

  //   document.documentElement.clientWidth

  // );

  // Scrolling - OLD WAY

  // window.scrollTo({

  //   lefth: s1coords.left + window.pageXOffset,

  //   top: s1coords.top + window.pageYOffset,

  //   behavior: 'smooth',

  // });

  // Scrolling - NEW WAY

  section1.scrollIntoView({behavior: "smooth"})

});

**TYPES of EVENTS and EVENT HANDLERS**

Para que un evento ocurra una sola vez podemos utilizar el **removeEventListener()**:

const h1 = document.querySelector("h1");

const alertH1 = (e) => {

  alert("Pasaste por el título");

  h1.removeEventListener("mouseenter", alertH1);

}

h1.addEventListener("mouseenter", alertH1);

**EVENT DELEGATION:**

El primer paso es agregar un addEventListener al elemento padre en común.

El segundo paso es determinar cual elemento originó el evento.

document.querySelector(".nav\_\_links").addEventListener("click", (e) => {

  e.preventDefault()

  // Matching stategy

  if(e.target.classList.contains("nav\_\_link")) {

    const id = e.target.getAttribute("href")

    // console.log(id);

    document.querySelector(id).scrollIntoView({behavior: "smooth"})

  }

})

**DOM TRAVERSING**

Lo utilizamos para recorrer elementos del árbol del DOM.

// DOM TRAVERSING

const h1 = document.querySelector('h1');

// Going downwards: child

console.log(h1.querySelectorAll('.highlight')); //NodeList(2) [span.highlight, span.highlight]

console.log(h1.childNodes); //NodeList(9) [text, comment, text, span.highlight, text, br, text, span.highlight, text]

console.log(h1.children); //HTMLCollection(3) [span.highlight, br, span.highlight]

h1.firstElementChild.style.color = "white";// le aplica ele stilo al primer child del h1

h1.lastElementChild.style.color = "white";

// Going upwards: parents

console.log(h1.parentNode);//el elemento padre

console.log(h1.parentElement);

// Para elegir un elemento padre más allá del primero que tenga utilizamos closest()

h1.closest(".header").style.background = "orange"

// Going sideways: siblings

console.log(h1.previousElementSibling); //en este caso fue null

console.log(h1.nextElementSibling); // un h4

// Para leer todos los siblings hacemnos este truco:

console.log(h1.parentElement.children); // HTMLCollection(4) [h1, h4, button.btn--text.btn--scroll-to, img.header\_\_img]

**BUILDING A TABBED COMPONENT**

// Tabbed component

const tabs = document.querySelectorAll('.operations\_\_tab');

const tabsContainer = document.querySelector('.operations\_\_tab-container');

const tabsContent = document.querySelectorAll('.operations\_\_content');

tabsContainer.addEventListener('click', function (e) {

  const clicked = e.target.closest('.operations\_\_tab');

  // console.log(clicked);

  // Guard clause

  if (!clicked) return;

  // Remove classes

  tabs.forEach(tab => tab.classList.remove('operations\_\_tab--active'));

  tabsContent.forEach(tabC => tabC.classList.remove('operations\_\_content--active'));

  // Active tab

  clicked.classList.add('operations\_\_tab--active');

  // Activate content area

  document

    .querySelector(`.operations\_\_content--${clicked.dataset.tab}`)

    .classList.add('operations\_\_content--active');

});

**PASSING ARGUMENTS TO EVENT HANDLERS**

// MENU FADE ANIMATION

const nav = document.querySelector('.nav');

const handlerHover = function(e) {

  if (e.target.classList.contains('nav\_\_link')) {

    const link = e.target;

    const siblings = link.closest('.nav').querySelectorAll('.nav\_\_link');

    const logo = link.closest('.nav').querySelector('img');

    siblings.forEach(el => {

      if (el !== link) el.style.opacity = this;

    });

    logo.style.opacity = this;

  }

}

// Passing "argument" into handler

nav.addEventListener('mouseover', handlerHover.bind(0.5));

nav.addEventListener('mouseout', handlerHover.bind(1));

**IMPLEMENTING A STICKY NAVIGATION: THE SCROLL EVENT**

No es la mejor forma de hacerlo.

// Sticky navigation

const initialCoords = section1.getBoundingClientRect();

window.addEventListener('scroll', function () {

  if (window.scrollY > initialCoords.top) {

    nav.classList.add('sticky');

  } else {

    nav.classList.remove('sticky');

  }

});

**A BETTER WAY: THE INTERSECTION OBSERVER API**

Este API básicamente le permite a nuestro código observar cambios en la forma en que un determinado elemento intercepta otro elemento, o que intercepta al viewport.

Para utilizarlo debemos crear un **new IntersectionObserver()**, al cual le pasamos como parámetro un **callback function** y un **objeto con opciones**. Este objeto necesita una propiedad **root**, el cual es el elemento que cruza al objetivo. También necesita un **threshold**, el cual es el porcentaje de intersección donde el callback va a ser llamado, y también puede tener un **rootMargin**, el cual es una caja de porcentaje en pixeles que aplicaremos fuera de nuestro elemento de destino

Por otro lado, el callback va a tener dos argumentos: **entries** y **observer**.

const header = document.querySelector('.header');

const navHeight = nav.getBoundingClientRect().height;

const stickyNav = function (entries) {

  const [entry] = entries;

  console.log(entry);

  if (!entry.isIntersecting) {

    nav.classList.add('sticky');

  } else {

    nav.classList.remove('sticky');

  }

};

const headerObserver = new IntersectionObserver(stickyNav, {

  root: null,

  threshold: 0,

  rootMargin: `-${navHeight}px`,

});

headerObserver.observe(header);

**REVELING ELEMENT ON SCROLL**

// Reveal sections

const allSections = document.querySelectorAll('.section');

const revealSection = function (entries, observer) {

  const [entry] = entries;

  // console.log(entry);

  if (!entry.isIntersecting) return;

  entry.target.classList.remove('section--hidden');

  observer.unobserve(entry.target);

};

const sectionObserver = new IntersectionObserver(revealSection, {

  root: null, //null = viewport

  threshold: 0.15,

});

allSections.forEach(function (section) {

  sectionObserver.observe(section);

  section.classList.add('section--hidden');

});

**LAZY LOADING IMAGES**

// LAZY LOADING IMAGES

const imgTargets = document.querySelectorAll('img[data-src]');

const loadImg = function (entries, observer) {

  const [entry] = entries;

  // console.log(entry);

  if (!entry.isIntersecting) return;

  // Replace src with data-src

  entry.target.src = entry.target.dataset.src;

  entry.target.addEventListener('load', () => {

    entry.target.classList.remove('lazy-img');

  });

  observer.unobserve(entry.target);

};

const imgObserver = new IntersectionObserver(loadImg, {

  root: null,

  threshold: 0,

  rootMargin: "200px"

});

imgTargets.forEach(img => imgObserver.observe(img));

**BUILDING A SLIDER COMPONENT**

// BUILDING A SLIDER COMPONENT

const slider = function () {

  const slides = document.querySelectorAll('.slide');

  const btnLeft = document.querySelector('.slider\_\_btn--left');

  const btnRight = document.querySelector('.slider\_\_btn--right');

  const dotContainer = document.querySelector('.dots');

  let curSlide = 0;

  const maxSlide = slides.length;

  // Functions

  const createDots = function () {

    slides.forEach((\_, i) => {

      dotContainer.insertAdjacentHTML(

        'beforeend',

        `<button class="dots\_\_dot" data-slide="${i}"></button>`

      );

    });

  };

  const activateDot = function (slide) {

    document.querySelectorAll(".dots\_\_dot").forEach(dot => {

      dot.classList.remove("dots\_\_dot--active")

    })

    document.querySelector(`.dots\_\_dot[data-slide="${slide}"]`).classList.add("dots\_\_dot--active");

  }

  const goToSlide = function (slideNumber) {

    slides.forEach((slide, i) => {

      slide.style.transform = `translateX(${100 \* (i - slideNumber)}%)`;

    });

  };

  // Next slide

  const nextSlide = function () {

    if (curSlide === maxSlide - 1) {

      curSlide = 0;

    } else {

      curSlide++;

    }

    goToSlide(curSlide);

    activateDot(curSlide);

  };

  // Prev slide

  const prevSlide = function () {

    if (curSlide === 0) {

      curSlide = maxSlide - 1;

    } else {

      curSlide--;

    }

    goToSlide(curSlide);

    activateDot(curSlide);

  };

  const init = function(){

    goToSlide(0);

    createDots();

    activateDot(0);

  }

  init()

  // Event handlers

  btnRight.addEventListener('click', nextSlide);

  btnLeft.addEventListener('click', prevSlide);

  document.addEventListener('keydown', function (e) {

    if (e.key === 'ArrowLeft') prevSlide();

    if (e.key === 'ArrowRight') nextSlide();

  });

  dotContainer.addEventListener("click", function (e) {

    if(e.target.classList.contains("dots\_\_dot")) {

      const {slide} = e.target.dataset;

      goToSlide(slide);

      activateDot(slide);

    }

  })

}

slider();

**LYFECICLE DOM EVENTS**

DOM Content Loaded: este evento es disparado por el **document** tan pronto el HTML este completamente parseado. Este evento no espera por imágenes u otros recursos externos a que carguen.

Load event: este evento es disparado por el **window** tan pronto como el HTML este parseado y como todos los recursos externos también (imágenes, CSS).

Before Unload event: es disparado por el **window**. Este evento es creado inmediatamente antes de que un usuario este por abandonar la página.

document.addEventListener('DOMContentLoaded', function(e){

console.log('HTML parsed and DOM tree built!', e);

});

window.addEventListener('load', function(e){

console.log('Page fully loaded', e);

});

window.addEventListener('beforeunload', function(e){

e.preventDefault();

console.log(e);

e.returnValue = '';

});

**SEC-14: OBJECT ORIENTED PROGRAMMING**

Es un paradigma de la programación basado en el concepto de objetos.

Hay 4 principios fundamentales de OOP:

1. Abstraction: ignorar detalles que no importan, permitiéndonos conseguir una perspectiva de la visión general de lo que estamos implementando.
2. Encapsulation: mantiene propiedades y métodos en privado dentro de la clase. No son accesibles desde fuera de la clase. Algunos métodos pueden ser expuestos como públicos (API).
3. Inheritance: Cuando tenemos dos clases parecidas podemos hacer una sola clase, donde la child class se extiende de la parent class. Conteniendo así todas las propiedades y métodos de la clase padre, pero agregando las nuevas propiedades y métodos que queramos a esta clase hija.
4. Polymorphism: una clase hija puede sobrescribir un método que heredó de una clase padre.

**CONSTRUCTOR FUNCTIONS and the “NEW” OPERATOR**

const Person = function (firstName, birthYear) {

  // console.log(this);//Person {} devuelve siempre el objeto

  this.firstName = firstName;

  this.birthYear = birthYear;

  //Nunca crear métodos dentro un constructor, porque realizaría miles de copias de la misma función

  // this.calcAge =  function () {

  //   console.log(2022 - this.birthYear);

  // }

};

const joaquin = new Person('Joaquín', 1997);

console.log(joaquin); // Person { firstName: 'Joaquín', birthYear: 1997 }

// 1. New {} is created.

// 2. function is called, this = {}.

// 3. linked to prototype.

// 4. function automatically return {}

console.log(joaquin instanceof Person);//true

**PROTOTYPES**

// PROTOTYPES

Person.prototype.calcAge = function () {

  console.log(2022 - this.birthYear);

};

joaquin.calcAge(); //25

console.log(joaquin.\_\_proto\_\_ === Person.prototype); // true

console.log(Person.prototype.isPrototypeOf(joaquin)); //true

// Estas no son propiedades directas de constructor

Person.prototype.species = 'Homo Sapiens';

console.log(joaquin.species); //Homo Sapiens

console.log(joaquin.hasOwnProperty('firstName')); //true

console.log(joaquin.hasOwnProperty('species')); //false

**PROTOTYPAL INHERITANCE ON BUILT-IN OBJECTS**

console.log(joaquin.\_\_proto\_\_.\_\_proto\_\_); //Este es Object.prototype

const arr = [3, 6, 6, 5, 6, 9, 3]; // new Array === []

console.log(arr.\_\_proto\_\_); //todos los métodos de los array

console.log(arr.\_\_proto\_\_ === Array.prototype); // true

// Podemos agregar un nuevo método con Array.prototype

Array.prototype.unique = function () {

  return [...new Set(this)];

};

console.log(arr.unique()); // [3, 6, 5, 9]

**ES6 CLASSES**