**Intro Js**

A inicios de los 98 – Módems 28KBps 🡪 Permitió aplicaciones web más complejas

Lenguaje de programación del lado del cliente.

Brendan Eich -> Netscape 🡪 Primera versión, llamado LiveScript

Por marketing le pusieron JavaScript debido a la popularidad de Java.

1997 se envió la especificación de JS 1.1 a la ECMA -> European Computer Manufacturers Association 🡪 TC39 creo el primer estándar llamado ECMA-262

JavaScript no es más que la implementación que realizó la empresa Netscape del estándar ECMAScript.

**Usos de JavaScript**

Hoy día pueden correr en el cliente y en el servidor.

Podemos crear webservers con la ayuda de node js.

Se pueden crear apps móviles con ayuda de ionic, react native, native script….

Tecnologías como Angular, View, React.

OJOO

Cada navegador tiene su propia implementación del estándar de ECMAScript

**Node nos permite correr JavaScript por fuera del navegador web**

**Ej:** Abro git bash y escribo node + enter 🡪 Escribo console.log(‘Hola Mundo’)

**Ej:** En visual estudio code creo archivo app.js y lo corro con la terminal 🡪 **node app.js**

**Recomendación:** Los archivos de JavaScript se recomienda ponerlos al final, para que cuando cargue la página web sea eso lo último que carga y no haga más lenta la carga de la página.

Por rendimiento poner los scripts al final del HTML

**Versiones de JavaScript:** 2009 -> ECMAScript 5 -> Soportada en todos los navegadores web  
2015 -> ECMAScript 6 🡪 Constantes, let, clases, etc -> Mayoría de navegadores web

De ahí decidieron hacer lanzamientos anuales.

**Polyfill**

Código que provee el funcionamiento de una nueva característica de JavaScript (ES6), en versiones viejas como ES5.

**Consola JavaScript**

JavaScript es un lenguaje interpretado -> Usualmente se lee de manera secuencial

**Depurar en el navegador** 🡪 En sources pongo un punto de debug en la parte del js que quiero visualizar

**Depurar en VSCode** 🡪 Run debug y seleccionar node

**USO DE VAR**

Al usar var se está colocando la variable dentro de un objeto global llamado window.

Usar var no es buena práctica.

El primer problema es que puedo nombrar una variable igual a alguna propiedad del objeto window y puede ser algo difícil de lidiar 🡪 Por eso es mejor let y const, porque no sobreescriben el objeto global window o donde sea que está corriendo JavaScript.

Con let somos capaces de ejecutar un código más limpio.

**Undefined 🡪** Valor de cualquier variable que no esté inicializada.

**Alert, Prompt y Confirm** son funciones que están dentro del objeto window. Si ejecutamos código en un lugar donde no esté el objeto window, no va a funcionar. Por ejemplo en node no funcionaría.

*En node ---------- > Global es similar a Window en un navegador web.*

**FUNDAMENTOS DE JAVASCRIPT**

JavaScript es un lenguaje débilmente tipado. Js intuye si es String o número, etc.  
Tipo de dato 🡪 Describe el contenido del valor que tiene la variable

**Primitivos** -> Es una información que no es un objeto y son inmutables

**Tipos De Datos Primitivos 🡪** Hay 6 tipos de datos primitivos en JS

Boolean -> true/false  
Null -> Sin valor en lo absoluto  
Undefined -> Una variable declarada a la que áun no se le asigna valor  
Number -> Integers, floats, etc.  
String-> Una cadena de caracteres  
Symbol -> Es un valor único que no es igual a ningún otro valor

**PALABRAS RESERVADAS**

let, for, switch, enum, const, etc …

**ARREGLOS**

Lista de información que contiene un grupo de elementos, usualmente del mismo tipo, pero en JS no es una regla que sean del mismo dato… [] .. Empiezan con la posición 0

**OBJETOS LITERALES**

JS tiene dos tipos: Objetos y primitivos

Objetos literales, objeto que tiene pares de valores

Al hacer const a = {} me bloquea que pueda hacer luego a = 123;

**FUNCIONES**

Su objetivo es centralizar cierta lógica y permitir la reutilización.

Es buena práctica que la definición de las funciones esté al inicio.

No se recomienda del todo definir la función con function saludar() porque luego con un var saludar = 123 la puedo dañar.

Es buena práctica asignarla como función anónima a const.

Las funciones tradicionales (definidas con function) tienen internamente un objeto llamado arguments, con las funciones de flecha eso no ocurre.

Funciones de flecha comenzaron con el estándar ES6

En las funciones de flecha es buena práctica dejar el paréntesis () => {}

**Retorno:** Si la función no tiene la palabra return entonces retorna undefined

**Nota**: Las funciones de flecha no tienen internamente el objeto arguments como si lo tienen las funciones tradicionales -🡪 Si quiero obtener los argumentos todos en un arreglo con una función de flecha entonces uso (…args) eso se llama parámetro Rest, no pueden tener ningún otro parámetro después de ese. Si necesito extraer algo aparte se pone adelante.

Cuando la función retorna un arreglo y yo quiero obtener los valores individualmente entonces hago:

*const* [casado, vivo, nombre, saludo] = imprimeArgumentos2(10, true, false, 'Daniel', 'Hola');

Con esto extraigo cada valor de forma independiente y me evito acceder posición por posición al arreglo.

También sirve con objetos, solo que haría const {nombre, apellido} = crearPersona(‘Daniel’, ‘Campu’);

**Desestructuración de objetos**

*const* tony = {

    nombre: 'Tony Stark',

    codeName: 'Ironman',

    vivo: false,

    edad: 40,

    trajes: ['Mark I', 'Mark V', 'Hulkbuster']

};

*const* imprimePropiedades = ({ *nombre*, *codeName*, *vivo*, *edad* = 15, *trajes* }) *=>* {

    console.log({ nombre });

    console.log({ codeName });

    console.log({ vivo });

    console.log({ edad });

    console.log({ trajes });

}

imprimePropiedades(tony);

**ESTRUCTURAS DE CONTROL DE FLUJO**

**Valor Y Referencia 🡪**

**Cuando trabajamos con primitivos y los mandamos a una función como argumento lo estamos mandando por valor.**

**En JS todos los objetos son pasados por referencia**, en js todo es un objeto exceptuando los primitivos.

Rompiendo la referencia: El … en este caso se llama operador spread y saca todos los valores del objeto. Ayuda a pasar por valor y no por ref, no confundir con el parámetro Rest.

*let* juan = { nombre: 'Juan' };

*let* ana = {...juan }; //Forma de romper la referencia y pasar el objeto por valor, es como crear una copia

En el caso de pasar un objeto como valor a una función se haría así:

*const* cambiaNombre = ({...*persona* }) *=>* {

    persona.nombre = 'Tony';

    return persona;

}

*let* peter = { nombre: 'Peter' };

*let* tony = cambiaNombre(peter);

**En arreglos:**

*const* frutas = ['Manzana', 'Pera', 'Pina'];

*const* otrasFrutas = [...frutas];

**Otra forma con arreglos:**

*const* otrasFrutas = frutas.slice();

**DOM – Document Object Model**

Conjunto de objetos anidados para crear un documento HTML 🡪 Tener acceso dinámico a todos los elementos a través de código.

document.querySelector('body')

document.querySelectorAll('img')

document.querySelector('small').innerText = '10'

document.getElementById('computadora-cartas');

document.querySelector('#computadora-cartas')

document.querySelector('.carta');

document.querySelectorAll('.carta')

document.getElementsByClassName('carta');

**Recomendado usar el query selector**

**Creando elementos en el DOM por consola**

const divBotones = document.querySelector('#divBotones')

const botonNuevo = document.createElement('button');

Agregar botón al div

divBotones.append(botonNuevo)

botonNuevo.innerText = 'Destruir el mundo';

botonNuevo.classList.add('btn')

botonNuevo.classList.add('btn-success')

**CALLBACK: Función que se ejecuta o dispara con la ocurrencia de una acción.**

**SECCION 6**

Patrón módulo -> Es el patrón mas común 🡪 Es compatible con casi cualquier versión ecmascript, también permite encapsular el código.

(() *=>* {

})();

Esta sentencia crea un nuevo scope el cual no puede ser llamado por nombre. Es una función anónima autoinvocada.

'use strict'

Se recomienda usarlo siempre cuando se implementa el patrón módulo.

**HACER PUBLICAS COSAS DE MI MODULO**

*const* miModulo = (() *=>* {

return {

        nuevoJuego: inicializarJuego

    };

})();

**Luego desde el index.html**

<script>

miModulo.nuevoJuego();

</script>

**Minificación de código**

Código comprimido a su máxima expresión.

Como cuando un baja una librería por ejemplo underscore-min.js

Es bueno hacerlo cuando el código va para la web o cuando lo van a tocar más personas

**CLASES EN JAVASCRIPT**

Prototype: \_\_proto\_\_ ……Es el ADN del objeto, quien es su padre, madre, de donde fueron originados. El objeto sabe de quien viene.

Actualmente se recomienda trabajar con clases en vez de prototipe 🡪 Class 🡪 ES6

WebPack transformar código moderno de JS a una versión globalmente aceptada.

**Problemática y Necesidad de Las Clases**

Antes de ES6 se tuvo una **alternativa para la herencia**

**Problema, herencia con prototype y tener que definir con new**

*function Persona(nombre, edad) {this.nombre = nombre; this.edad=edad;}*

*const maría = new Persona(‘Maria’, 18);*

**Clases básicas en JS**

El constructor se ejecuta en el momento que se crea una nueva instancia de la clase

El constructor es el único método que retorna una instancia de un objeto en vez de undefined

**Gets y Sets en JS**

Son un poco distintos a los de Java, tienen la palabra clave get y set al inicio del método y se invocan sin paréntesis, como si fuera una igualación.

**Propiedades o métodos estáticos**

No se está trabajando con una clase instanciada.

Permiten usar dichas propiedades o métodos sin necesidad de instanciar la clase

Palabra reservada **static**

**HERENCIA**

Se usa la palabra **extends**

**WEBPACK**

Angular utiliza webpack, Angular 🡪 compañías como slack, discord, Airbnb, etc dan dinero a webpack.

**Tareas automáticas** -> Live reload de los cambios, minimizar el código, ofuscar código, incrementar compatibilidad con otros navegadores etc.

**Node** 🡪 Correr código de JavaScript en el servidor.

**Pero también nuestra computadora, puede ser un servidor de desarrollo.**

**BABEL 🡪 Librería que convierte código de JS moderno a JS antiguo para tener compatibilidad**

**NODE 🡪 350 mil paquetes, me permite desarrollar localmente. Sólo instalo la librería o dependencia y ya.**

**WEBPACK -🡪 Es un empaquetador de módulos. Ayuda a gestionar las dependencias. El webpack junto con npm me permite instalar un paquete y usarlo de una. Me permite montar servidores de desarrollo y pruebas. Me permite cargar módulos. Convertir a diferentes formatos. Minimizar código. Compilar de SASS a CSS. Complilar de TS a JS. Nos permite trabajar con JS moderno.**

* **Config inicial puede ser un dolor de cabeza.**
* **Puede ser complicado detectar problemas con algún paquete.**

**PASO A PASO**

**Npm init 🡪 Va a crear el package.json 🡪** Todas las aplicaciones en node tienen un package.json el cual indica las dependencias de la aplicación en dllo, pdn, etc. Me dice que paquetes necesita mi aplicación, cuales paquetes no van en pdn, etc.

**JavaScript tiene una forma para importar:**

import { saludar } from './js/componentes'

**Los frameworks más importantes usan webpack 🡪** webpack son dependencias de desarrollo.

**Carpeta node\_modules 🡪** No es algo que va a ser necesario en producción, o al menos no generalmente.

En el package.json en scripts agregó uno nuevo para ejecutar el comando webpack en el build.

**npm run build 🡪** Hace algunas cosas y me genera una carpeta llamada ***dist***. -> Adentro se creó un archivo main.js

OJOOOOOOOOOOOOOO, para importar en otro lado -🡪 **EXPORT**

export *const* saludar = (*nombre*) *=>* {

    console.log('Creando etiqueta h1');

*const* h1 = document.createElement('h1');

    h1.innerText = `Hola, ${nombre}`;

    document.body.append(h1);

}

**Archivo de config del webpack**: webpack.config.js

Instalar html loader y html webpack plugin

---Se puede crear un webpack y un webpackprod ………. Se configurar su ejecución en los script del package.json

**BABEL**

Ayuda a la transformación de un estándar más alto de ecmascript a un estándar mas bajo o más aceptado, por ejemplo pasar del 6 al 5.

Babel is a JavaScript compiler

**GITHUB** –

**GITHUBPAGES – Hacer npm run build 🡪 Luego renombrar la carpeta dist por docs y ya con eso hacer push. GITHUBPAGES toma lo que haya dentro de docs y lo despliega en un sitio de hosting**

Hosting gratuito para proyectos de HTML, JS y CSS. 🡪 React, Angular, etc usted podría hacerlo.

**MANTENER INFORMACIÓN**

**----Solo se puede tener un localStorage por dominio**

Sólo funcionan cuando va orientado a la web, pertenece al navegador.

Se puede ver en el navegador -> Application -> Storage -> Local Storage o Session Storage.

No almacenar información sensible porque está información es visible al usuario final.

**Local Storage ->** La información almacenada acá no posee tiempo de expiración, aguanta hasta un reinicio de la máquina.

**Session Storage ->** Se borra todo cuando se cierra el navegador web

**PROMESAS Y CALLBACKS**

**CALLBACKS -> Es una función que se pasa como argumento y cuyo objetivo es que se dispare en el momento en que se termine el procedimiento.**

Los callbacks se ejecutan de manera secuencial.

El estándar es callback(error, success);

**CALLBACK HELL 🡪** Se vuelve muy difícil de mantener cuando se deben ejecutar acciones o más callbacks con el resultado de otro callback -🡪 En el ES6 fue que nacieron las promesas para lidiar con este inconveniente.

**PROMESAS**

**No son bloqueantes ni síncronas a diferencia de los callbacks.**

Una de las dos partes se compromete a hacer algo y luego notifica si logró cumplir lo que iba a hacer o no.

Sintaxis promesas 🡪 Parámetros 🡪 1. Un callback cuyos params son resolve y reject

return new *Promise*( (*resolve*, *reject*) *=>* {

} );

Para invocarlas existen varias formas

buscarHeroe(heroeId1).then(*heroe* *=>* {

    console.log(`Enviando a ${ heroe.nombre } a la misión`)

})

**JS permite ejecutar varias promesas en paralelo 🡪 Promise.all 🡪 Solo se ejecuta el código si las promesas se ejecutan exitosamente todas.**

*Promise*.all([ buscarHeroe(heroeId1), buscarHeroe(heroeId2) ])

    .then( ([*heroe1*, *heroe2*]) *=>* {

    console.log(`Enviando a ${ heroe1.nombre } y ${ heroe2.nombre } a la misión `)

}).catch(*err* *=>* {

    alert(err);

}).finally(() *=>* {

    console.log('Se termino el promise.all');

});

**ASYNC**

Al poner la palabra

async

antes de una función lo que hace es que la vuelve asíncrona y que me retorne una promesa, facilita la sintaxis para retornar promesas.

export *const* buscarHeroeAsync = async(*id*) *=>* {

*const* heroe = heroes[id];

    if(heroe) {

        return heroe;

    }else{

        throw Error(`No existe un héroe con el id ${ id }`)

    }

}

buscarHeroeAsync('iron')

    .then(*heroe* *=>* console.log(heroe))

    .catch(console.warn);

**AWAIT**

Tiene una condición para usarse y es que debe estar dentro de una función async. Ayuda a esperar a que la promesa se resuelva. Hay que usarlo sabiamente porque en pocas palabras le estoy diciendo a la función en donde se usa que se debe esperar a que la promesa se resuelva lo cual puede hacer que la aplicación se note lenta.

export *const* obtenerHeroesArr = async() *=>* {

*const* heroesArr = [];

    for( *const* id of heroesIds ){

*const* heroe = await buscarHeroeAsync(id);

        heroesArr.push(heroe);

    }

    return heroesArr;

};

Problema común y como resolverlo, lo anterior no es lo mejor 🡪 **NO USAR AWAIT DENTRO DE CICLO FOR PORQUE MULTIPLICARIA EL TIEMPO QUE TARDA CADA PROMESA**

export *const* obtenerHeroesArr = async() *=>* {

*const* heroesArr = [];

    for( *const* id of heroesIds ){

        heroesArr.push(buscarHeroe(id));

    }

    return await *Promise*.all(heroesArr);

};

Lo mismo pero mas elegante

return await *Promise*.all( heroesIds.map(*id* *=>* buscarHeroe(id)) );

MANEJO DE ERRORES CON EL AWAIT

Se puede usar Try – Catch

export *const* obtenerHeroeAwait = async(*id*) *=>* {

    try{

*const* heroe = await buscarHeroeAsync(id);

        return heroe;

    }catch( err ){

        console.log('CATCH!!!')

        return {

            nombre: 'Sin nombre',

            poder: 'Sin poder'

        }

    }

}

For await

    for await(*const* heroe of heroesPromesas) {

        console.log(heroe);

    }

**PETICIONES HTTP**

JSON -> Estándar para intercambio de información

//PROMESAS EN CADENA

fetch( jokeUrl )

    .then( *resp* *=>* resp.json() )

    .then( *jsonResponse* *=>* console.log(jsonResponse) );

PROVIDERS ( Services) -> Permite centralizar la lógica para realizar peticiones