# INTRO

## Polyfills

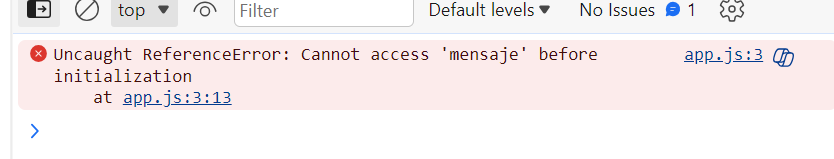
Es un codigo que provee una funcionalidad de una caractiristica de JavaScript (ES6) como por ejemplo una funcion de flecha, clases, etc… en versiones viejas como el ES5. Permite usar Javascript moderno.

## Let

En ES6 surge la posibilidad de declarar variables con la palabra reservada ***let*** que lo que hace a diferencia de ***var*** es lanzar un error cuando una variable no esta definida antes de ser usada por ejemplo:

console.log(mensaje);

*let* mensaje = "Hola Mundo";



Ademas no la muestra en el objeto global windows por lo que es mas seguro.

Tampoco puede declararse mas de una variable por bloque. Y al declarar 2 variables con el mismo nombre pero en diferentes bloques se toman como individuales:

A screenshot of a computer

Description automatically generated

Esto permite que para los ciclos podamos definir las variables de conteo con ***let*** y asi que solo sea valida para ese bloque por lo tanto si definimos un nuevo ciclo con el mismo nombre de variable no pasa nada

## Const

Como el Let tampoco sobreescribe la variable que se encuentre en objeto global window o donde esté corriendo Js

Tambien aparecieron las Constantes, que deben inicializarse al momento de definirlas y su valor luego no puede modificarse(a menos que sea un objeto, ahí si nos deja modificar sus propiedades pero no el objeto completo):

*const* IMPUESTO\_PAIS = 21;

Al igual que con ***let*** podremos declarar otra constante con el mismo nombre solo en un bloque diferente

# Strings

## startWith - endsWith - Includes (Case sensitive)

Devuelve true si el string empieza con el carácter o caracteres que le pasemos por parametro:

*var* saludo = "Hola Mundo";

console.log(saludo.startsWith("H"));

y el endsWith() igual pero al final del string

Includes() devuelve true o false depediendo si existe o no lo que le pasemos por parametro

Como segundo parametro les podemos pasar la cantidad de caracteres desde donde queremos que empiece a comparar(en base 1)

## repeat

regresa un string duplicado la cantidad de veces que le indiquemos en el parametro:

saludo.repeat(2);

## Literal Templates

Surgen para tratar de solucionar algnas de las limitaciones que JS tiene en el uso de Strings

*let* nombre = "Pablo";

*let* apellido = "Sanabria";

*let* nombreCompleto = `El nombre completo es ${nombre} ${apellido}`

console.log(nombreCompleto);

En lugar de las variables pueden ir funciones tambien. Lo que pongamos dentro de ${} es codigo JS puro

### Multilinea

*let* multilinea = `<h1 class="unaclase">Titulo</h1>

<p>Hola Mundo</p>`;

console.log(multilinea);

### Tags

Son funciones que se usan para validar o manejar/formatear los templates literales antes de asignarlos.

*function* etiqueta(*literales*, ...*substituciones*){

    //console.log(arguments);

    //console.log(literales);

    //console.log(substituciones);

*let* resultado = "";

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

        resultado += literales[i];

        resultado += substituciones[i];

    }

    return resultado;

}

*let* unidades = 5,

    costo = 10;

*let* mensaje = etiqueta`${unidades} lapices cuestan ${unidades \* costo} pesos`;

console.log(mensaje);

#### Raw

String.raw`Hola /n mundo //`; imprime en pantalla esto tal cual

# Parametros Opcionales o por defecto

pueden ser de cualquier tipo

//--colocar siempre primeros los parametros obligatorios y luego los opcionales

*function* saludar( fn = *function*(){console.log("Hola Mundo");}){

    fn();

}

saludar();

A tener en cuenta: Con parámetros por defecto, si no envías parámetros a la función, el objeto arguments estará vacío, aunque existan parámetros por defecto.

## Parametro REST - Strings separados a Array

Es indicado con 3 puntos (…) seguido del nombre que le asignaremos. Se convierte en un arreglo que contiene el resto de los parametros pasados a una funcion. Solo puede existir uno solo y debe ir siempre ultimo

*function* saludar( *arr\_alumnos*, ...*masAlumnos*){

    //console.log(arguments);

    console.log(arr\_alumnos, masAlumnos);

}

*let* arr\_alumnos = ["Pablo"];

*let* arr\_alumnos2 = saludar(arr\_alumnos, "Pedro", "Vilma", "Sara");

## Operador Spread - Arr a Strings separados

Seria lo contrario a REST, se envia un array y cada valor se toma como un parametro separado.

*let* numeros = [1,2,84,3,4,55];

*let* max = Math.max(...numeros);

console.log(max);

Permite romper la relación de referencia de los objetos:

*let* persona = {

    nombre: "Pablo",

    apellido:"Sanabria",

    edad: 37,

}

*let* persona2 = {...persona};

persona2.nombre = "Leonardo";

persona2.edad = 41;

console.log(persona);

console.log(persona2);

Añadir propiedades a objetos a partir de otros objetos:

*let* persona = {

    nombre: "Pablo",

    apellido:"Sanabria",

    edad: 37,

}

*let* persona2 = {

    nombre: "Pablo",

    apellido:"Sanabria",

    edad: 37,

    direccion:"jvg 2314",

    ciudad: "San miguel"

}

persona2.nombre = "Leonardo";

persona2.edad = 41;

persona = {...persona2, ...persona};

console.log(persona);

console.log(persona2);

Quitar referencia de objeto a parametro de una funcion:

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

    persona.nombre = 'Tony';

    return persona;

}

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

*let* tony  = cambiaNombre( peter );

console.log({ peter, tony });

En arreglos:

// Arreglos

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

console.time('slice');//loggear tiempo para ver cuanto demora un proceso

*const* otrasFrutas = frutas.slice();

console.timeEnd('slice');

console.time('spread');

*const* otrasFrutas2 = [...frutas];

console.timeEnd('spread');

otrasFrutas.push('Mango');

console.table({ frutas, otrasFrutas });

# NEW.TARGET - META PROPIEDAD

Nos asegura que nuestros objetos siempre sean declarados como el objeto new cuando lo necesitamos

Es una propiedad de un no-objeto que provee informacion adicional relacionada con su procedencia.

Esto resuelve el problema de que cuando en ES5 llamábamos a la funcion call() del prototipo de una “FuncionObjeto” sin la palabra new no rompia pero deberia porque no se está pasando un Objeto si no una funcion: Persona.call(persona, “Hola”)

Cuando el constructor de la funcion es llamado new.target se llena con el operador new.

Si la funcion call() es ejecutada new.target no estará definida ya que no se ejecutó el constructor

if(typeof new.target !== “undefined”){

    //inicializar o asignar valores al objeto

    this.nombre = nombre;

}

else

    throw new Error("Esta funcion debe ser utilizada con el new");

# Funciones de Flecha o Arrows - Lambda function

Se comportan de una manera muy diferente a las funciones normales en ESC5. No hay creacion de this (No pueden cambiar el valor de this), super, arguments y new.target. El valor de estos objetos apuntan al objeto o funcion padre que no sea de tipo arrow.

No puede ser llamada con new. No tienen constructor por consecuencia no hay prototipo.

No pueden tener nombres duplicados de parametros

Comparacion:

*var* miFuncion1 = *function*(*valor*){

    return valor;

}

*let* miFuncion2 = *valor* *=>* valor;

//mas de un parametro

*let* sumar = (*num1*,*num2*) *=>* num1+num2;

//sin parametros

*let* saludar = () *=>* "Hola Mundo desde Fn Arrow";

//varias lineas

*let* saludarPersona = *nombre=>*{

*let* saluda = `Hola ${nombre}`;

    return saluda;

}

//retornar objeto literal

*let* obtenerLibro = *id* *=>* ({

    id: 3,

    nombre: "Harry Poter"

})

//Funcion anonima

*let* saludo3 = (*nombre* *=>* `Hola desde fn arrow anonima ${nombre}`)()

Podemos decir, que podemos desechar las funciones típicas del ES5 y solo utilizar funciones de flecha?

NO! Las funciones tradicionales no sólo trabajan como funciones, sino que permiten la creación de objetos y otros mecanismos totalmente útiles hoy en día. Las funciones de Flecha expanden el JavaScript y nos dan utilidades nuevas, pero no reemplazan a las funciones tradicionales.

# Objetos

En ES5 eran colecciones de pares. En ES6 si las propiedades y los valores tienen el mismo nombre lo podemos simplificar (si ponemos algun nombre distinto rompe):

*function* crearPersona(*nombre*, *apellido*, *edad*){

    return {

        nombre,

        apellido,

        edad

    }

}

*var* pers = crearPersona("Pablo","Sanabria",37);

## Metodos Simples o concisos

*var* obj =  {

    numero:10,

    nombre:"Pabl010",

    //Antes

    //imprimirNombre: function(){...

//Ahora Metodo conciso

    imprimirNombre(){

        console.log(this.nombre);

    }

}

## Nombres de propiedades computadas o procesadas

Los nombres computados, pueden tener ciertos caracteres especiales y espacios

*var* subfijo = " nombre";

*var* banda = {

//ojo que si ponemos algo con 2 espacios en el medio no funciona

    ["1banda" + subfijo]: "Qpersonaje",

    ["2banda" + subfijo]: "Lakonga"

};

console.log(banda["1banda nombre"]);

console.log(banda["2banda" + subfijo]);

## Object.is()

Ademas de los comparadores == y ===, se agrego el Object.is(parametro1,parametro2) que sirve para hacer comparaciones mas especificas o precisas.

## Object.assign()

Permite agarrar todas las propiedades de un objeto y heredarse las transmitírsela a otras sin usar la herencia directamente Object.assign(objetoReceptor,objetoDonador)

## Orden de enumeración de las propiedades

Todas las llaves van en orden ascendente. Todas las llaves de tipo string van ordenadas en manera en que fueron agregadas al objeto. Todos los símbolos aparecen en el orden en que fueron agregados al objeto

# Prototipos

## Cambiar el prototipo de un objeto

*let* gato = {

    sonido(){

        console.log("Miau");

    },

    chillido(){

        console.log("MIAU!!!");

    }

}

*let* perro = {

    sonido(){

        console.log("Guau");

    }

}

*let* angora  = Object.create(gato);

console.log(Object.getPrototypeOf(angora) === gato);

angora.sonido();

angora.chillido();

Object.setPrototypeOf(angora,perro);

console.log(Object.getPrototypeOf(angora) === gato);

angora.sonido();

angora.chillido();

## Acceder a un prototipo padre o heredado

*let* persona = {

    saludar(){

        return "HOla!, ";

    }

};

*let* amigo = {

    saludar(){

        //antes

        //return Object.getPrototypeOf(this).saludar(this) + "Saludos!!!";

        //ahora

        return super.saludar() + "Saludos!!!";

    }

};

Object.setPrototypeOf(amigo, persona);

console.log(amigo.saludar());

# Desestructuración

## De objetos

Sirve para cuando tenemos objetos o variables que se utilizan mucho. Entonces podemos desglosar un objeto para que cree diferentes objetos de manera automatica

*let* ajustes = {

    nombre: "Pablo Sanabria",

    email: "pablosanab@hotmail.com",

    facebook: "p.s.Face.com.77",

    premium: true

}

//se desestructura con el nombre que se le define en el objeto, no importa el orden

//let {nombre,email,facebook,premium} = ajustes;

//se puede modificar el nombre de la nueva variable/objeto al momento de la asignacion

//let {premium:dePago,nombre,email,facebook} = ajustes;

//tambien se pueden declarar nuevas variable

//let {premium:dePago,nombre,email,facebook, twitter = "Pablo.x.86"} = ajustes;

//tambien se pueden declarar nuevas variable con parametros opcionales si twitter existiera

//y tuviera valor en el objeto ajustes se mostraria ese valor

*let* {premium:dePago,nombre,email,facebook, twitter:cuentaTwi = "Pablo.x.86"} = ajustes;

console.log(dePago, nombre, cuentaTwi);

## De objetos anidados

*let* autoguardado = {

    archivo: "index.js",

    cursor: {

        linea:7,

        columna:16

    },

    ultimoArchivo: {

        archivo: "index.html",

        cursor: {

            linea: 8,

            columna: 20

        }

    },

    otroNodo:{

        subNodo:{

            cursor:{

                linea:11,

                columna:11

            }

        }

    }

};

//primer objeto

//let {cursor}= autoguardado;

//3er objeto

// let {ultimoArchivo:{cursor:ultimoArchivo2}} = autoguardado;

// console.log(ultimoArchivo2);

//4to objeto

*let* {otroNodo:{subNodo:{cursor:ultimoArchivo2}}} = autoguardado;

console.log(ultimoArchivo2);

//otra manera

*let* superNodo = autoguardado.otroNodo.subNodo.cursor;

console.log(superNodo);

## De arreglos

*let* frutas = ["banana", "uva", "frutilla"];

//es en orden secuencial

*let* [fruta1,fruta2] = frutas;

console.log(fruta1, fruta2);

//si quiero solo la 3era fruta

*let* [,,fruta3] = frutas;

console.log(fruta3);

//se puede sobreescibir valores

*let* otraFruta = "manzana";

//aqui otraFruta toma el valor de la primera posicion del arr frutas

[otraFruta] = frutas;

console.log(otraFruta);

//-------------

//permite cambiar valores en variables

*let* a = 1;

*let* b = 2;

*let* temp;

[a,b]=[b,a];

console.log(a);

console.log(b);

## De arreglos anidados

*let* colores1 = ["rojo", ["verde","amarillo"],"morado", "naranja"];

*let* [color1, [color2] ] = colores1;

 console.log(color1,color2);

*let* colores2 = ["rojo", "verde","amarillo","morado", "naranja"];

 //hacer que los colores luego del primero se guarden en otro arreglo

 //el valor rest(...) para desdestructuracion solo sirve para arreglos no para objetos

*let* [colorPrincipal, ...demasColores] = colores2;

 console.log(colorPrincipal);

 console.log(demasColores);

## De parámetros(con parametros por defecto)

*function* crearjugador(*nickname*,

    { *hp*, *sp*, *clase* }= {hp:100,sp:50,clase:"Mago"}

){

    //antes mandando el parametro opciones si no conociamos la funcion no sabiamos que podia

//venir adentro

//opciones = opciones || {};

    // let hp = opciones.hp,

    // sp = opciones.sp,

    // clase = opciones.clase;

    console.log(nickname,hp,sp,clase);

    //codigo para crear el jugador...

 }

 crearjugador("Pipino",{

        hp: 100, sp: 50, clase: "Mago"

});

*const* imprimeArgumentos2 = ( ...*args* ) *=>* {

    return args;

}

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

console.log({ casado, vivo, nombre, saludo });

*const* crearPersona = ( *nombre*, *apellido* ) *=>* ({ nombre,apellido });

*const* { apellido: nuevoApellido } = crearPersona( 'Fernando', 'Herrera' );

console.log({ nuevoApellido });

*const* tony = {

    nombre: 'Tony Stark',

    codeName: 'Ironman',

    vivo: false,

    edad: 40,

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

};

// const imprimePropiedades = ( personaje ) => {

//     console.log( 'nombre',personaje.nombre );

//     console.log( 'codeName',personaje.codeName );

//     console.log( 'vivo',personaje.vivo );

//     console.log( 'edad',personaje.edad );

//     console.log( 'trajes',personaje.trajes );

// }

*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 );

# Simbolos

Es un nuevo tipo de dato, son perfectos para poner nombres de propiedades, ya que siempre son diferentes(cada uno es unico), y asegurarnos de que no colisionen o entren en conflicot con nombres de otras propiedades/elementos de otras librerias

*let* primerNombre = Symbol();

*let* segundoNombre = Symbol('Segundo Nombre');//le podemos poner una alias

*let* persona = {

    [segundoNombre]: "Ezequiel"

};

persona[primerNombre] = 'Fernando';

console.log(persona[primerNombre], persona[segundoNombre]);

console.log(segundoNombre);

## Compartir Simbolos

Para esto en ES6 existe un pool de todos los simbolos creados

//verificar que el simbolo no hay sido creado

*let* userID = Symbol.for("userID");

*let* objeto = {};

objeto[userID] = "12345";

console.log(objeto[userID], userID)

//hacer referencia al simbolo creado

*let* userID2 = Symbol.for("userID");//este nuevo simbolo apunta a la referencia del anterior

//obtener el id de un simbolo

*let* idSimbolo = Symbol.for("id unico");

console.log(Symbol.keyFor(idSimbolo));

*let* id3 = Symbol("id unico"); //si creamos unpo nuevo con el mismo id

console.log(id3 === idSimbolo);//son distintos

console.log(id3,idSimbolo);

## para leer lossimbolos de un objeto :

let simbolos = Object.(getOwnPropertySymbols(persona));

# Sets

Son una lista ordenada de valores sin duplicados que permiten rapido acceso a la data que contienen ya que tienen funciones nativas en sus prototipos. Pueden tener cualquier valor inclusive otros sets. Tambien se los conoces como StrongSet, permite realizar indices, busquedas directas.

*let* items = new Set();

//ignora valores repetidos

items.add(10);

items.add(10);

items.add(11);

items.add(7);

items.add("7");//esto si lo permite xq internamente es un object

console.log(items, items.size);

*let* items2 = new Set([1,2,3,4,5,7]);

console.log(items2.has(7));//devuelve true o false

//borrar elementos

console.log(items2, items2.size);

items2.delete(3);//compara con el contenido, no con la posicion, si no existe lo ignora

//console.log(items2, items2.size);

items2.clear();//elimina todos los elementos

//console.log(items2);

//recorrer sets

*let* personas = new Set(["Fernand", "PAblo", "Susana"]);

personas.forEach( *function*(*valor*, *llave*, *setOriginal*){

    //console.log(valor, llave, setOriginal);

}

);

//Convertir Set en array

*let* numeros = [1,2,3,2,4,7,5,6,7];

*let* setNumeros = new Set(numeros);

//console.log(setNumeros);

*let* arrNumeros = [...setNumeros];

//console.log(arrNumeros);

//Ej elimnacion de duplicados en array

*function* eliminarDuplicados(*items*){

    return [... new Set(items)];

}

console.log(eliminarDuplicados(numeros));

## WeakSet

Trabaja similar a los Set pero con una manera de manejo de informacion diferente:

En Add(), Has, Remove(), si enviamos algo que no sea un objeto como parametro va a dar error.

No hace repeticiones o ciclos For In y tampoco ForEach. No tienen Keys, Values por lo que no hay manera de saber cuantos elementos hay dentro. Tampoco tienen Size.

Solo sirven para almacenar referencias a los objetos

# Maps o mapas

Nuevo tipo de colecciones de datos que tiene una llave y un valor Key-Value pair, es similar a los Set con respecto a su manejo de datos y sus propiedades

*let* mapa = new Map();

mapa.set("nombre","Fernando");

mapa.set("edad",31);

mapa.set("apellido");// si no colocamos valor por defecto queda undefined

//console.log(mapa);

console.log(mapa.get("nombre"));//obtener valor pasando key

console.log(mapa.has("nombre"));//verifica existencia devuelve true o false

mapa.delete("nombre");//borra todo el par

mapa.clear("nombre");//elimina todos los valores del mapa

console.log(mapa);

//--inicializar por defecto

*let* mapa2 = new Map([ ["nombre","Fernando"],["edad",31] ]);

console.log(mapa2);

//--Ciclo For Of

for( *let* i of mapa2){

    console.log(i);

}

# Clases

Todo el codigo dentro de una clase funciona en modo Estricto. Tiene metodos estaticos y privados. Por defecto, los métodos son públicos.

*class* Persona {

*constructor*(*nombre*){

        this.nombre =  nombre;

    }

    decirNombre(){

        console.log(this.nombre);

    }

}

*let* fernado = new Persona("Fernando");

fernado.decirNombre();

console.log(typeof Persona);//devuelve function en vez de class por alguna razon desconocida

## Clases como expresiones

*class* Persona = {

*constructor*(*nombre*){

        this.nombre =  nombre;

    }

    decirNombre(){

        console.log(this.nombre);

    }

}

## Clases como parámetros

*function* creadorClases(*definicionClase*){

    return new definicionClase();

}

*let* objeto =  creadorClases( *class* {

*constructor*(){

        this.nombre = undefined;

        this.edad = 30;

    }

    saludar(){

        console.log("Holis!");

    }

});

objeto.saludar();

*class* Cuadrado{

*constructor*(*lado*){

        this.lado  = lado;

    }

    getArea(){

        return this.lado \* this.lado;

    }

};

*function* imprimirCuadrado(*cuadrado*){

    if(!(cuadrado instanceof Cuadrado)){

        console.log("El parametro enviado no es un cuadrado");

        return

    }

    else

        console.log(cuadrado.getArea());

}

*let* mesa = new Cuadrado(10);

imprimirCuadrado(mesa);

## Metodo Estatico

Sin instanciar la clase, es importante que no requiera niguna propiedad de la misma clase, es decir que sea autonomo

*class* Persona {

*constructor*(*nombre*){

        this.nombre =  nombre;

    }

    decirNombre(){

        //console.log(this.nombre);

    }

    static crear(*nombre*){

        return new Persona(nombre);

    }

}

*let* yo = Persona.crear("Pablito");

## Propiedades Computadas o metodos computados

Son metodos cuyos nombres se pueden definir a traves de una variable

*let* nombreMetodo = "gritarNombre";

*class* Persona {

*constructor*(*nombre*){

        this.nombre =  nombre;

    }

    decirNombre(){

        //console.log(this.nombre);

    }

    [nombreMetodo](){

        console.log(this.nombre.toUpperCase());

    }

    static crear(*nombre*){

        return new Persona(nombre);

    }

}

## Herencia

*class* Rectangulo {

*constructor*(*alto*,*largo*){

        this.alto = alto;

        this.largo = largo;

    }

    getArea(){

        return this.alto \* this.largo;

    }

}

*let* rectangulo = new Rectangulo(3,2);

console.log(rectangulo.getArea());

*class* Cuadrado extends *Rectangulo*{

*constructor*(*alto*){

        super(alto,alto);

    }

}

*let* cuadrado = new Cuadrado(3);

console.log(cuadrado.getArea());

## Sobreescribir funciones del padre

*class* Cuadrado extends *Rectangulo*{

*constructor*(*alto*){

        super(alto,alto);

    }

    getArea(){

        return "Cuadrado " + (this.alto \* this.alto);//ejecuta esto

        //return super.getArea();//ejecuta el metodo del padre

    }

}

# Promesas

*function* tareaAsincrona(){

*let* promesa = new *Promise*((*resolve*,*reject*)*=>*{

        setTimeout(*function*(){

            console.log("Proceso Asincrono Terminado");

            reject();

        },1300);

    });

    return promesa;

}

//es opcional el reject

tareaAsincrona().then( *function*(){

    console.log("resolve: Todo Ok");

},

*function*(){

    console.error("reject: Todo Mal");

});;

console.log("Codigo Secuencial");

## Callbacks

Son la forma tradicional de hacer tareas a destiempo, no son asincronos a menos que tengan una tarea asincrona dentro, demoran la accion de una funcion hasta que se realice una tarea antes

*const* getUsuarioById = (*id*, *callback*) *=>* {

*const* usuario = {

        nombre: "Fernando",

        id

    };

    //simulamos error

    if(id === 20){

        callback(`El usuario con el id ${id} no existe`);

    }

    else{//callback( err, usuario);

        callback( null, usuario);

    }

};

//getUsuarioById(id, (callback)

getUsuarioById(5, (*err*, *usuario*)*=>*{

    if(err){

        return console.log(err);

    }

    console.log('Usuario de BD ', usuario);

});

Puede haber un callback dentro de otro y esto hace que sea dificil de leer el codigo:

para ejecutar un callback por ejemplo con node, en la terminal escribir el comando: node nombredelcallback

*const* empleados = [

    {

        id:1,

        nombre: "Pablo"

    },

    {

        id:2,

        nombre: "Ezequiel"

    },

    {

        id:3,

        nombre: "Juan"

    }

];

*const* salarios = [

    {

        id:1,

        salario: 1000

    },

    {

        id:2,

        salario: 2000

    }

];

*const* getEmpleado= (*id*, *callback*)*=>*{

*const* empleadoDB = empleados.find( *empleado* *=>* empleado.id === id);

    if(!empleadoDB){

        callback(`No existe el empleado con id ${id}`);

    }

    else{

        callback(null, empleadoDB);

    }

};

//crear una fn getSalario, que reciba el id de la persona y que si la persona no existe el mensaje será

//err: no se encontró un salario para el empleado {nombre del empleado}

//Si existe: la salida será un objeto

// {

//     nombre: "Fernando",

//     salario: 1000,

//     id: 1

// }

*const* getSalario = (*empleado*, *callback*) *=>* {

*const* salarioDB = salarios.find( *salario* *=>* salario.id === empleado.id);

    if(!salarioDB){

        callback(`No se encontró un salario para el empleado ${empleado.nombre}`);

    }

    else{

        callback(null, {

           nombre: empleado.nombre,

            salario : salarioDB.salario,

            id: empleado.id

        });

    }

}

//uso

getEmpleado(3, (*err*, *empleado*)*=>*{

    if(err){

        return console.error(err);

    }

    getSalario(empleado,  (*err*, *resp*)*=>*{

        if(err){

            return console.error(err);

        }

        console.log(`El salario del empleado ${resp.nombre} es de ${resp.salario}`);

    });

    //console.log("Empleado ", empleado);

});

para resolver este problema se crearon las promesas, para que las funciones en vez de retornar callbacks retornen promesas

//definicion

*const* getEmpleado= (*id*) *=>* {

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

*const* empleadoDB = empleados.find( *empleado* *=>* empleado.id === id);

        if(!empleadoDB){

            reject(`No existe el empleado con id ${id}`);

        }

        else{

            resolve(empleadoDB);

        }

    });

};

*const* getSalario = (*empleado*) *=>* {

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

*const* salarioDB = salarios.find( *salario* *=>* salario.id === empleado.id);

        if(!salarioDB){

            reject(`No se encontró un salario para el empleado ${empleado.nombre}`);

        }

        else{

            resolve({

            nombre: empleado.nombre,

                salario : salarioDB.salario,

                id: empleado.id

            });

        }

    });

}

//uso

// getEmpleado(1).then( empleado => {

//     // getSalario(empleado).then( resp => {

//     //     console.log(resp);

//     // });

//     //para no estar haciendo lo mismo que con los callbacks y anidar promesas se lo haceen el then

//     return getSalario(empleado);

// })

// .then( resp => console.log(resp))

// .catch( err => console.log(err));

//mejorando el codigo

getEmpleado(5)

    .then( getSalario)

    .then( console.log)

.catch(console.log);

## Async Await ES7

*const* getNombre = async()*=>*{ //al poner async ya sabemos que devuelve una promesa

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

        setTimeout(() *=>* {

            resolve("Pablito");

        }, 3000);

    });

}

//una funcion aync no necesariamente debe tener un await

*const* saludo = async ()*=>*{

    //para que funcione el await tiene q estar en una funcion async

*const* nombre = await getNombre();//se espera el resultado de esta prmesa antes de seguir

    return `Hola ${nombre}`;

}

saludo().then( *mensaje* *=>* {

    console.log(mensaje);

});

// getNombre().then(nombre => {

// });

Ejemplo final

*const* empleados = [

    {

        id:1,

        nombre: "Pablo"

    },

    {

        id:2,

        nombre: "Ezequiel"

    },

    {

        id:3,

        nombre: "Juan"

    }

];

*const* salarios = [

    {

        id:1,

        salario: 1000

    },

    {

        id:2,

        salario: 2000

    }

];

//definicion

*const* getEmpleado= async (*id*) *=>* {

*const* empleadoDB = empleados.find( *empleado* *=>* empleado.id === id);

    if(!empleadoDB){

       throw new Error(`No existe el empleado con id ${id}`); //reject();

    }

    else{

        return empleadoDB;

    }

};

*const* getSalario = async (*empleado*) *=>* {

*const* salarioDB = salarios.find( *salario* *=>* salario.id === empleado.id);

    if(!salarioDB){

        throw new Error(`No se encontró un salario para el empleado ${empleado.nombre}`);

    }

    else{

        return {

            nombre: empleado.nombre,

            salario : salarioDB.salario,

            id: empleado.id

        };

    }

}

*const* getInformacion = async (*id*)*=>*{

*const* empleado =  await getEmpleado(id);

*const* resp =  await getSalario(empleado);

    return `El empleado ${empleado.nombre} tiene un salario de ${resp.salario} pesos`;

};

getInformacion(2)

.then(console.log)

.catch(console.log);