**Básicos de JavaScript**

ISOMORFISMO: JS es capaz de ejecutarse en las 3 capas de una aplicación

1. Frontend (JS)

2. Backend (Node.js)

3. Persistencia de Datos (MongoDB, etc.)

CARACTERÍSTICAS:

- Lenguaje de alto nivel (recolección de basura)

- Interpretado. (no necesita fase de compilación)

- Dinámico

- Débilmente tipado

- multi paradigma

- sensible a mayúsculas y minúsculas

- no necesita “;” al final de cada línea

Primitivos: Se accede directamente al valor

- string

- number

- boolean

- null

- undefined

- NaN

Compuestos: Se accede a la referencia del valor

- object = {}

- array = []

- function () {}

- Class {}

- Etc.

**Variables**

-var ámbito global (no usar)

-objeto window (mapeo de toda la ventana del navegador, aquí van todas las (var) globales

let, solo existe en bloques)

-const para variables que no cambiaran y se usa para declarar funciones.

**Template string**

let nombre = "Jon";

let apellido = 'Mircha';

**Normal:**

let saludo = "Hola mi nombre es " + nombre + " " + apellido + ".";

**interpolación de variables con template string:**

backtip (´) acento grave o invertido

let saludo = ´Hola mi nombre es ${nombre} ${apellido}. ´;

**Métodos para strings**

Metodo indexOf: permite buscar un texto en otra cadena de texto.

Metodo length: devuelve un valor de entero que proporciona el número de elemento de source\_string.

Metodo slice: devuelve una copia de una parte del array dentro de un nuevo array empezando por inicio hasta fin (fin no incluido). El array original no se modificará.

Metodo substring: método devuelve un subconjunto de un objeto String

Metodo Split: divide un objeto de tipo String en un array (vector) de cadenas mediante la separación de la cadena en subcadenas.

Metodo repeat: construye y devuelve una nueva cadena que contiene el número especificado de copias de la cadena en la cual fue llamada, concatenados.

Metodo reverse: invierte el orden de los elementos de un array in place. El primer elemento pasa a ser el último y el último pasa a ser el primero.

Metodo splice:  permite cambiar el contenido del arreglo eliminando o sustituyendo los elementos existentes por otros nuevos.

Método join: une todos los elementos de una matriz (o un [objeto similar a una matriz](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Indexed_collections#working_with_array-like_objects)) en una cadena y devuelve esta cadena.

Metodo math random: devuelve un número de coma flotante pseudo-aleatorio, comprendido en el rango de 0 a menor que 1

Metodo toString: El método toString() devuelve una cadena de caracteres representando el array especificado y sus elementos.

Método concat: se usa para unir dos o más arrays, este método no cambia los arrays existentes, sino que devuelve un nuevo array.

Metodo factorial: El factorial de un número es el producto de dicho número por todos los anteriores menores que él hasta llegar al cero. Por definición el factorial del número 0 es 1.

El resultado se representa por una notación abreviada que es el símbolo de cierre de exclamación (!).

Dicho esto, el factorial del número 5 sería:

5! = 5 \* 4 \* 3 \* 2 \* 1

PARA CREAR LA FUNCION EN JS SE DEBE RETORNAR 1 AL FINAL DEL CODIGO PARA DETER EL CICLO YA QUE EL METODO FACTORIAL REGRESA UN 1 COMO MINIMO VALOR YA QUE LA MULTIPLICACION POR 0 DARIA 0.

**Expresiones regulares**

Las expresiones regulares son patrones que se utilizan para hacer coincidir combinaciones de caracteres en cadenas. En JavaScript, las expresiones regulares también son objetos. Estos patrones se utilizan con los métodos exec() y test() de RegExp, y con match(), matchAll(), replace(), replaceAll() (en-US), search() y split() métodos de String.

Para usar una expresión regular se crea una expresión regular literal con / / o con el objeto RegExp.

[a-z] => Eliminara todas las letras.

/\D/g; => Elimina las letras en general, la “g” indica que busque globalmente, si no se pone la g entonces solo borra la primer letra que se encuentre.

**Numbers**

let a = 2;

let c = 7.19

let d = "5.6";

c.toFixed (1); // 7.2

c.toFixed (2); // 7.19000

parseInt(c); // 7

parseFloat(c) //7.19

typeof c - number

typeof d - string

**Booleans**

let verdadero = true;

let false = false;

let v = Boolean(true);

let f = Boolean(false);

Truthy = (true, {}, [], 42 etc.)

falsy = (false, null, undefined, 0, NaN etc.)

**Null, undefined and NaN**

undefined: valor ausente, no se le asigna por nada a la variable por el usuario

null: valor ausente, se le asigna intencionalmente algo a la variable por el usuario

NaN: Not a Number

**Funciones**

Función declarada =

function estoEsUnaFuncion()

{

console.log("Uno");

}

invocación de función

estoEsUnaFuncion();

function unaFuncionQueDevuelveValor()

{

return "Hola";

}

let res = unaFuncionQueDevuelveValor();

console.log(res);

function saludar(nombre = "Desconocido", edad = 0){ //nueva caracteristica 2015, se pueden inicializar las variables

console.log(´hola mi nombre es ${nombre} y tengo ${edad}´);

}

saludar("perro", 12);

saludar(); //hola mi nombre es Desconocido y tengo 0

**Funciones declaradas vs funciones expresadas**

funcionDeclarada(); //Puede invocarse antes de ser declarada por el hosting, sube todo al scope global --solo funciones declaradas

function funcionDeclarada()

{

console.log("puede invocarse en cuqluier parte de nuestro codigo, incluso antes de que la funcion sea declarada");

}

funcionDeclarada();

//funcion expresada

const funcionExpresada = function (){

console.log("funcion que se le ha asignado como valor a una variable, si invocamos esta funcion antes de su definicial JS nos dira

cannot access funcionExpresada before initialization");

}

como JS es un lenguaje expresado, nos obliga a declarar las funciones antes de ser utilizadas, elimina errores de compilacion, con funciones expresadas

**Arrays**

const a = []; //arreglo vacio

const c = [1, true, "Hola", ["A", "B", "C"]]; //areglo inicializado

c.length; //tamaño real del arreglo

c[2]; //imprime Hola

c[3][2] //imprime C

se accede como si fueran cordenadas

const b = Array.of("x", "y","z"); // si imprimo c, en consola se muestra un arreglo con esos valores, otra forma de inicializar un arreglo.

const d = Array(100).fill(false); // hacemos uso del constructor Array para usar sus funciones, en este caso creara un arreglo con 100 valores false

const e = new Array(); //forma vieja de declarar un arreglo (ya no se usa)

const e = new Array(1,2,3,false); //forma vieja de inicializar un arreglo

b.push("a"); //inserta elemento

b.pop(); //saca ultimo elemento en entrar

const colores = ["Rojo", "Verde", "Azul"];

colores.forEach(function (el, index){ //function aninima que regresara el valor y el indice de colores

console.log(´<li id="${index}">${el}</li>´);

});

---------------------12. Objetos ---------------------

en JS todo es un objeto

let a = new String("Hola"); //mando a llamar el objeto String q existe en JS atra vez del constructor, inicializo dentro de la variable a con Hola

const b = {} //objetos se declaran con "{}"

const c = new Object();

//Los objetos dentro de un objeto a las variables se les llama propiedades/atributos y a las funciones se les llama metodos

const jon ={

nombre:"Yorce",

apellido:32,

pasatiempo: ["Correr", "Hacer ejercicio","Dar Clases"],

contacto:{

email:"eded@gmail.com",

twitter:"@ded",

movil:"33333"

},

saludar: function(){

console.log('Hola');

}

decirMiNombre: function(){

console.log(´Hola me llamo ${this.nombre} ${this.apellido} y tengo ${this.edad} años´ );

}

}

//accediendo al objeto

jon["nombre"]; //una forma no tan comun

jon.nombre; //forma mas usada

jon.pasatiempo[1];

jon.contacto.twitter;

jon.saludar();

jon.decirMinombre();

//Object es el objeto principal de JS

console.log(Object.keys(jon)); // en consola nos da las llaves (nombre, epellido, pasatiempos,....)

console.log(Object.values(jon)); // ["Yorce", 32, Array(2),{...},f,f]

jon.hasOwnProperty("nombre"); //true, si existe esa propiedad

-------------------------13. Tipos de operadores-------------------

operadores aritmeticos

+, -, \*, / , %, (),

parentesis, raices potencias, multip div, suma resta . De izquierda a derecha

operadores relacionales

<,>,>=(compuesto),<=,==,===,!=,!==

= : asignacion

== : comparacion de valores 7 == "7" -> true

=== : comparacion de valor y tipo de dato 7 === "7" -> false, 7 === 7 -> true, 0 === false -> false

incremento y decremento

let i = 1;

i=i+2; //3

i += 3 //4

//Operador unario

//tener cuidado con estos , en tiempo de ejecucion hay diferencias entre poner antes o despues el incremento o decremento

let j = 1;

j++ //= 1, aqui no hace el incremento si no hasta la proximo uso de j, pensariamos q es 2

j-- //= 2, aqui no hace el decremento si no hasta la proximo uso de j, pensariamos q es 1

++j //= 2, aqui si hace el incremento luego luego del res anterior

--j //= 1, aqui si hace el decremento luego luego del res anterior

//Logicos

! Not

|| OR

&& AND

----------------14. Condicionales------------------------------------

if

else if

else if

else

operador ternario

condicion ? verdadero : falso

let edad = 19;

let eresMayor = (edad >= 18)

? "Eres mayor de edad"

: "Eres menor de edad";

//switch - case

let dia = 2;

switch(dia){

case 0:

console.log("Domingo");

break;

case 1:

console.log("Domingo");

break;

case 2:

console.log("Domingo");

break;

case 3:

console.log("Domingo");

break;

case 4:

console.log("Domingo");

break;

default:

console.log("mmmmm");

break;

}

-----------------------15. ciclos----------------

ya no se utilizar tanto

let contador = 0;

while(contador < 10){

console...

contador++;

}

//al menos se ejecuta una vez la instruccion en el do while

do{

console...

contador++;

}while(contador < 10)

//mas utilizado forEach

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

conosle... i

}

let num = [10,20,30,40,50];

for(let i = 0; num.length < 10; i++){

conosle... num[i]; //10,20,30,40,50

}

//nuevos desde el ecmaSC (for in, for of)

const jon ={

nombre:"Yorce",

apellido:32,

pasatiempo: ["Correr", "Hacer ejercicio","Dar Clases"]

}

//for especial para recorrer un objeto

for(const propiedad in jon){

console.log(´key: ${propiedad}, value: ${jon[propiedad]}´); //aqui no se uza el punto para acceder a el value, se uzan [].

}

//for especial para recorrer cualquier coleccion de datos q sea iterable (arreglo, cadena texto)

for(const elemento of num){

console.log(elemento); //imprime 10,20,30,40,50

}

let cadena = "Hola Mundo";

for(const elemento of cadena){

console.log(elemento); //imprime Hola Mundo

}

//-------------16. exceptions---------------------------------------------------

try{

//codigo a evaluar

// si en la segunda linea se encontro un error, el catch cacha ese error pero ejecuta la primera linea donde todo iba bien, se detiene justo donde encuntra el error

}catch(){

//captura cualquier error surgido o lanzado en el try

}finally{

//se ejecutara siempre al final de un bloque try-catch (Se usa en apertura y cierre de archivos, BD: abre en try y cierra siempre en finally)

}

//errores personalizados

if isNaN(numero) //true si no es un numero

{

throw new Error("El caracter introducido no es un numero");

}

catch(error){

console.log(´se produjo el siguiente error: ${error}´)

}

// si no inserto un numero mostrata

// se produjo el siguiente error: El caracter introducido no es un numero

//---------------17.Break and continue--------------

//Break and continue: no se puede usar en foreach, solo con for

const numeros = [1,2,3,4,5,6,7,8,9,0];

for(let i=0; numeros.length, i++){

if(i === 5){

break; // ya no imprime a partir del indice 6

}

console.log(numeros[i]);

}

for(let i=0; numeros.length, i++){

if(i === 5){

continue; // se salta solo la iteracion del indice 5 y continua, imprime 1,2,3,4,5,7,8,9

}

console.log(numeros[i]);

}

//-----------------------------------Video 18: "Destructuración"

let {nombre, edad, apellido} = props; // asignacion de elementos de props en variables separadas

//sin destructuring

const numeros = [1,2,3];

let uno = numeros[0],

dos = numeros[1],

tres = numeros[2];

console.log(uno, dos, tres);

//con destructuring

const [uno, dos, tres] = numeros;

console.log(uno, dos, tres);

const persona ={

nombre: 'Dan',

apellido: 'Lima',

edad: 35

};

let {nombre, edad, apellido} = persona;

console.log(nombre, edad, apellido);

//variable creada se debe llamar igual que la propiedad del objeto

//no importa si se ponen las varibles en diferente orden

//-----------------------------------Video 19: Objetos literales

let nombre = "Caty",

edad = 20;

const dog ={

nombre: nombre, //normal sin el uso de obj literales

edad: edad, //normal sin el uso de obj literales

raza: "Callejero", // no tengo esa variable la pongo normal

ladrar(){ //anteriormente ladrar: function(){ conosle... }

console......

}

}

//con objeto literal

const dog ={

nombre, //debe ser el mismo nombre q la variable de afuera, si uso obj literal

edad,

raza: "Callejero", // no tengo esa variable sa pongo normal

ladrar(){ //anteriormente ladrar: function(){ conosle... }

console......

}

}

//-----------------------------------Video 20: parametros rest & operator spread

Ejemplo: parametros rest (capacidad de tener parametros infinitos)

function Suma(a,b, ...c){ // el c puede tener valor o no

let resultado = a+b;

c.forEach(funcion (n){

resultado += n;

});

return resultado;

}

console.log(Suma(1,2)); 3

console.log(Suma(1,2,3)); 6

console.log(Suma(1,2,3,4)); 10 // 3 y 4 se reciben de tipo array [3,4]

console.log(Suma(1,2,3,4,5)); 15

console.log(Suma(1,2,3,4,5,6)); 21

Spread operator u operador de propagacion: multiples parametros

const arr1 = [1,2,3,4,5],

arr2 = [6,7,8,9,0];

const arr3 = [...arr1, ...arr2]; //concatenados, se transforma en un arreglo nuevo arr3

//-----------------------------------Video 21: Arrow functions

//para funciones expresadas

//sin funcion flecha

const saludar = function(){

console.log("Hola");

}

const saludar = () => console.log('Hola'); // si solo tiene una linea el cuerpo se pueden quitar las llaves

const saludar = nombre => console.log(`Hols ${nombre}`); //cuando una arrow func resive solo un parametro (nombre) podemos eliminar los parentesis, si recibe mas de 1 se deben de poner

const Semar = (a,b) => a+b; //return implicito

console.log(Sumar(5,4));

//Funcion de varias lineas

const functionnn = () => {

console.....

.....

.....

}

const numeros = [1,2,3,4,5];

numeros.forEach(el, index) => console.log(`${el} esta en la posicion ${index}`);

numeros.forEach(function(el, index){

console.log(`${el} esta en la posicion ${index}`);

})

IMPORTANTE

function Perro(){

console.log(this); //Aqui imprimira el contexto del papa donde fue declarada en este caso window

}

//////////////////////////ver

//dentro de un objeto literal, no es una buena practica crear un objeto a partir de una arrow function,

const perro = {

nombre: "Kenai",

ladras: () => { /

console.log(this); //Aqui imprimira el contexto del objeto donde se encuentra, en este caso window (solo si uso arrow function), ----forma incorrecta

}

}

const perro = {

nombre: "Kenai",

ladras: function(){

console.log(this); // aqui this es el objeto perro, por que estoy usando funcion normal y respeta el contexto donde se crea----forma correcta

}

}

///////////////////////////////// forma nueva de crear un metodo dentro de un objeto literal valida, no es arrow function

const perro = {

nombre: "Kenai",

ladras(){

console.log(this); //Aqui imprimira el contexto en done se encuentra imprimero objeto perro, tambien podria ser de la forma anterior de ECS5 ladrar: funcion(){}....

// imprimira: {nombre: "Kenai, ladras: f"} ----forma correcta

}

}

//-------------22. Prototipos--------------

POO

2015, ECMASC6 -> clases, azucar sintatica, las clases el complador lo combierte a funciones prototipas

Clases: modelo a seguir, machote, molde

objeto: instancia de una clase, ej: obj perro de la clase animal, instancia es la copia de ese modelo a seguir.

atributo: caracteristica o propiedad del objeto (variables de un objeto)

metodos: son las acciones que un objeto puede realizas (son funciones dentro de un objeto)

const animal={

nombre:"",

sonar(){

}

}

//funcion constructora

function Animal(nombre, genero){

this.nombre = nombre;

this.genero = genero;

this.sonar = function(){

console.log("Hago sonidos por que estoy vivo"); //lo ideal es sacar este metodo y colgarlo en el prototipo por que si no cada instancia lo repetira, lo ideal

//q las func constructoras solo tengan losatributos no los metodos

}

}

//aqui los sacamos de la func constructoras y los pegamos al prototipo, mejor rendimiento en memoria, estan en las clases si no en el prototipo

Animal.prototype.sonar = function(){

console.log("Hago sonidos por que estoy vivo");

}

const snoppy = new Animal("snoopy", "macho");

const snoppy = new Animal("lola", "hembra");

//---------------------23. Herencia Prototipica-----------------

function Perro(nombre, genero,tamanio){

this.super = Animal; //es como super en otros lenguages

this.super(nombre, genero);

this.tamanio = tamanio;

}

//perro esta heredando de animal

Perro.prototype = new Animal();

Perro.prototype.constructor = Perro;

//sobreescritura de metodos del prototipo padre en el hijo

Perro.prototype.sonar = function(){

console.log("soy un perro y mi sonido es un ladrido");

}

//nuevo metodo

Perro.prototype.ladrar = function(){

console.log("guau guau");

}

const snoppy = new Animal("snoopy", "macho"); //anteriormente sin herencia

const snoppy = new Perro("Snoopy", "Macho", "Mediano"); //ahora con herencia

//---------------24. Clases y herencia-------------

class Animal{

//el constructor es un metodo especial que se ejecuta en el momento de instanciar la clase

constructor(nombre, genero){

this.nombre = nombre;

this.genero = genero;

}

//MEtodos

sonar(){

console.log("Hago sonidos por que estoy vivo");

}

saludar(){

console.log(´Hola me llamo ${this.nombre}´);

}

}

class Perro extends Animal{

constructor(nombre, genero, tamanio){

//con el metodo super se manda a llamar el constructor de la clase padre

super(nombre, genero);

this.tamanio = tamanio;

this.raza = raza; // este metodo no lo recibo desde el constructor, lo obtengo y retorno con get y set methods

}

sonar(){

console.log("soy un perro y mi sonido es un ladrido");

}

ladrar(){

console.log("Guauu Guau");

}

get getRaza(){

return this.raza;

}

set setRaza(raza|){

this.raza = raza;

}

}

const mimi = new Animal("Mimi, Hembra"),

scooby = new Perro("Scooby", "Macho", "Gigante");

//-----------25. Metodos estaticos, getters y setters------------

//en JS no existen las clases provadas, protected, solo son publicas. tampoco las clases abstractas

//si existen los metodos estaticos, se pueden ejecutar sin necesidad de instanciar la clase

static queEres(){

console.log("Los perros somos....");

}

Perro.queEres(); // se llama directo desde la clase, sin instanciar

//Los setters and getters son metodos especiales que nos permiten establecer y obtener los valores de atributos de nuestra clase

get getRaza(){

return this.raza;

}

set setRaza(raza|){

this.raza = raza;

}

//metodo obtenedor JS lo combierte a propiedad

scooby.setRaza = "Gran danes";

scooby.getRaza // llamada como si fuera una propiedad

//--------------------26. objeto console------------

//tiene muchos metodos y atributo

console.log

console.error

console.warn

console.info

console.log(a,b,c,d);

console.log(´mi nombre es %s´, nombre);

console.clear();

console.dir(document); //desplega sus propiedades y metods como un objeto

console.log(document); //desplega como el html normal en la consola

console.group("cfrfr");

console.log("fff");

console.groupEnd();

console.log(console);

console.log(Object.entries(console).sort()); //muestra todos los metodos q tiene la consola en formato tablita

const numeros = [1,2,3,4,5];

const vocales = ["a","e","i","o","u"];

console.table(numeros); //muestra su index y valor

console.table(vocales);

console.time("cuanto tiempo tarda mi codigo"); // debe ser la misma etiqueta, nos dice cuanto tarfo el for en ejecutarse

const arreglo = Array(100000);

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

arreglo[i] = i;

}

console.timeEnd("cuanto tiempo tarda mi codigo"); // debe ser la misma etiqueta

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

console.count("codigo for"); //cuantas veces se ejecuto mi codigo

console.log(i);

}

//metodo para testing

let x = 3,

y = 2;

pruebaXY = "Se esperaba q x siempre fuera menos que y";

console.assert(x < y, {x,y,pruebaXY}); //condicion, en forma de objeto la variables en juego, variable del mensaje

//-----------27. Object Date---------------

Date()

let fecha = new Date();

fecha.getDate(); //dia del mes

fecha.getDay(); //D L M....0,1,2...

fecha.getMonth(); //Ene Feb Mar....0,1,2...

fecha.getYear(); //120, apartir del año 1900 cuantos años han pasado, no usado

fecha.getFullYear(); //2022, mas usado

fecha.getHours();

fecha.getMinutes();

fecha.getSeconds();

fecha.toString(); // nos da la hora completa en string, dia mes año hora y GMT -0600 hora estandar central, son -6 horas a la hora 0 donde esta el meridiano de grenwich,

// mismo formato que nos da el constructor new date

fecha.getUTCHours() //hora 0, del meridiano de wrenwich, no la hora de donde estoy

fecha.Date.now() //cuantos segundos han pasado desde 1 de enero 1970, time stamp, usando cuando guardo en BD q aceptan en time spam, moment JS, libreria muy usada para fechas.

let cumpleJon = new Date(1983,5,23); // me dice cuando naci exactamente

//..................28: Objeto Math..................

Objeto Math

metodos(abs(), cas()) y propiedades(PI, E) valores q no cambiaran

Math.PI;

Math.abs(-7.8); //7.8

Math.ceil(7.2); // 8 redondea al cielo

Math.floor(7.8) // 7 al piso

Math.round(7.5) // 8, (7.49) -> 7

Math.sqrt(81)

Math.pow(2,5)

Math.sign(-0.3) // -1 te dice si es negativo -1, cero 0 o positivo 1

Math.random() //numero entre 0 y 1

Math.random() \* 1000 // numero entre 0 y 1000

Math.round(Math.random() \* 1000) ////quita la parte fraccionaria con el round

//............29. operadores de corto circuito.................

cortocircuito or: cuando el valor de la izquierda en la expresion siempre pueda validar a true, es el valor que se cargara por defecto

cortocircuito and: cuando el valor de la izq en la expresion siempre pueda validar a false, es el valor q cargara por defecto

console.log(19 || "valor "); // 19 es true entonces se imprimira 19, valor truly el 19

console.log(null || "valor"); //null es false, entonces imprimira el valor de la derecha, null es falsy

//-------------30. alert, confirm y prompt

metodos q cuelgan del objeto window, no necesario ponerlo

window.alert("Hola esto es una alerta");

let confirm = window.confirm("Hola esto es una confirmacion"); //aceptar y cancelar , confirm: true or false

let aviso = window.prompt("Hola esto es una aviso"); aviso= variable con lo q escribimos

//-------------31. Expresiones regulares------------

//validar correo, tarjetas de credito

//secuencia de caracteres que forma un patron de busqueda, principalmente en la busqueda de un patron de una cadena de caracteres

let cadena = "Lorem ipsum dedededed ededede dededed eded";

//utilizando funcion contructora

let expReg = new RegExp(patron, banderas);

let expReg = new RegExp("lorem", "ig"); //banderas g:buscar todas las coincidencias, i: ignora mayusculas y minusculas

console.log(expReg.test(cadena)); //el metodo test solo dice true or false

console.log(expReg.exec(cadena)); //el metodo exec debuelve un objeto con mas info

//otra forma de crear una FR

let expReg2 = /lorem/ig;

console.log(expReg.test(cadena));

console.log(expReg2.exec(cadena));

let expReg2 = /[0-9]/ig; // \d para numeros de 0 a 9

let expReg2 = /lorem{1,2}/ig; // \d lorem hasta 2 veces, revisar mas sobre llaves con numeros

//-----------------32. Funciones anonimas autoejecutables-----------------

ejemplo jquery tiene una funcion autoejecutable

(function( global, factory ){

}

//funcion anonima autoejecutable

alert("Hola");

function (){} //funcion sin nombre

function(){} //esto vendria siendo el nombre de la funcion como alert

function(){}() //esto vendria siendo el nombre de la funcion como alert y el parentesis q la ejecuta

(function(){}) //la estructura de la funcion la meto en un parentesis para q no quede desprotegida

(function(){

//codigo aqui

})(); //asi quedaria la funcion anonima aut, aqui si se requiere ; al final

(function(d,w,c){

//codigo aqui

console.log(d);

console.log(w);

c.log("Este en un console.log");

})(document, window, console); //esto era muy usando cuando no habia modulos

//formatos de FAE

(function(){

//Crockford(JS the good parts)

})();

((function(){

//clasica

})());

+function(){

//unaria

}();

!function(){

//facebook

}();

//------------33. Modulos--------------

// Ver ejercicios

//-------------------44. Temporizadores------------

setTimeout(() => {

Console.log("Ejecutando un SetTimeout, esto se ejecuta una sola vez");

}, 3000)

setInterval(() => {

Console.log("Ejecutando un SetInterval, esto se ejecuta indefinidamente cada cierto intervalo de tiempo");

}, 1000)

setInterval(() => {

console.log(new Date().toLocalTimeString());

}1000);

let temporizador = setTimeout(() => {

Console.log("Ejecutando un SetTimeout, esto se ejecuta una sola vez");

}, 1000)

clearTimeout(temporizador); //lo detiene clearTimeout

let temporizador = setInterval(() => {

console.log(new Date().toLocalTimeString());

}, 1000)

clearInterval(temporizador); //lo detiene clearInterval

//---------------45. Asincronia y el Event loop-------------------

// JS es un lengiaje single thread, un solo hilo

Event loop in JS -> lenjuage de un solo hilo,

un thread es por cada proceso de mi maquina, al abrir el navegador, puede ser q un proceso pueda correr varios hilos, dependiendo del lenjuage, single o multi threads

JS, peticiones - single thread - tipo peticion(DB, API, Respuesta aritmetica). call stack, LIFO, la ultima en trar ultima en salir

procesamiento single thread y multi thread

Operaciones de CPU (loop infinito for, operacion de cpu, JS consume mas rendimiento de la maquina) y Operaciones de I/O (operaciones q esperan peticiones de un recurso solicitado, formulario)

Operaciones concurrentes (dos o mas tareas se ejecutan simultaneamente, progresivamente una antes y una despues) y paralelas (dos o mas tareas se ejecutan al mismo tiempo)

Operaciones bloqueantes (fase de espera, no devuelve control a la aaplicacion hasta q haya terminado su tarea) y no bloqueantes (se ejecuta, y devuelve el control al hilo principal, no importa si haya terminado la tarea o no)

operaciones sincronas (respues en el presente) y asincronas (respues en un futuro (devuelve el control al hilo despues de ejecutarse), podria definirse como no bloqueante)

JS usa un modelo asincrono y no bloqueandte.

con un loop de eventos implemetado en un solo hilo (single thread) para operaciones de entrada y salida(input/output)

//codigo sincrono bloqueante (cada operacion se hace de una vez bloqueando el flujo, hasta q devuelve respuesta, el hilo se desbloquea y se sigue con la sig tarea)

(() =>{

console.log("Codigo Sincrono");

console.log("Inicio");

function dos(){

console.log("Dos");

}

function uno(){

console.log("uno");

dos();

console.log("tres");

}

uno();

console.log("fin");

})() //imprimira: codigo sincrono, inicio, uno,dos,tres,fin

//codigo asincrono no bloqueante - uso de setTimeout para simular asincronia, el setTimeout con 0 minutos tambien se toma un tiempo para comenzar a ejecutarse, funciona

// con lifo en el stack de operaciones.

(() => {

console.log("Codigo Asincrono");

console.log("Inicio");

function dos(){

setTimeout(function(){

console.log("Dos");

},1000);

}

function uno(){

setTimeout(function(){

console.log("Uno");

},0);

dos();

console.log("tres");

}

uno();

console.log("Fin");

})() //imprimira: codigo asincrono, inicio, tres,fin,uno,dos

//------------46. Callbacks----------------

//parte de asincronia, opeaciones de netrada y salida en un solo hilo, asincronas y no bloqueantes (no detienen ejecucion, regresan control al programa en cuando el proc este listo)

// las callbacks (llamada de vueltas), es una funcion que se va a ejecutar despues de q otra lo haga, mecanismo q tiene JS para ejecutar sus funciones, (funciones anonimas, etc)

// las calbacks primer mecanismo q tenemos para manejar la asincronia

// promesas, asyncronia y generadores, son una mejor implemtacion para simular asincronia y no las callbacks

function cuadradoCallback(value, callback){

setTimeout(( => {

callback(value, value \* value)

}, 0 | Math.random() \* 100))

}

cuadradoCallback(0, (value, result) => {

console.log("inicia callback");

console.log(´callback: ${value}, ${result}´);

cuadradoCallback(1, (value, result) => {

onsole.log(´callback: ${value}, ${result}´);

cuadradoCallback(1, (value, result) => {

onsole.log(´callback: ${value}, ${result}´);

cuadradoCallback(1, (value, result) => {

onsole.log(´callback: ${value}, ${result}´);

cuadradoCallback(1, (value, result) => {

onsole.log(´callback: ${value}, ${result}´);

cuadradoCallback(1, (value, result) => {

onsole.log(´callback: ${value}, ${result}´);

onsole.log(´Fin callback´);

onsole.log(´callback hell´);

onsole.log("http://callbackhell.com");

});

});

});

});

});

});

//---------------47. Promesas---------------

verlo con un if else, si se cumple cierta operacion la promesa se cumple, else reject

una mejora con las promesas es que ya manejan el error, en las callbacks necesitabamos hacerlo manual

Fetch parte moderna de hacer ajax, utiliza mucho este patron (then, catch)

function cuadradoPromise(value){

if(typeof value !== "number"){

return Promise.reject(´Error, el valor "${value}" ingresado no es un numero ´);

}

return new Promise((resolver, reject) = {

setTimeout(() => {

resolver({

value,

result: value \* value

});

},0 | Math.random() \* 1000);

});

}

cuadradoPromise(0)

.then((obj) => {

console.log("inicio promesa");

console.log(´Promise: ${obj,value}, ${obj.result}´);

return cuadradoPromise(1);

})

.then((obj) => {

console.log(´Promise: ${obj,value}, ${obj.result}´);

return cuadradoPromise(2);

})

.then((obj) => {

console.log(´Promise: ${obj,value}, ${obj.result}´);

return cuadradoPromise(3);

})

.then((obj) => {

console.log(´Promise: ${obj,value}, ${obj.result}´);

return cuadradoPromise(4);

})

.then((obj) => {

console.log(´Promise: ${obj,value}, ${obj.result}´);

return cuadradoPromise(5);

})

.then((obj) => {

console.log(´Promise: ${obj,value}, ${obj.result}´);

console.log("fin promesa");

})

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

//----48. Async - await ---------------------------------------------

esto no viene a remplazar a las promesas, si no trabajan en conjunto

//es muy util cuando en un sitio tenemos un loader de todos los productos, tenemos q esperar a q tengamos todo el objeto para poder pintar en el dom

function cuadradoPromise(value){

if(typeof value !== "number"){

return Promise.reject(´Error, el valor "${value}" ingresado no es un numero ´);

}

return new Promise((resolver, reject) = {

setTimeout(() => {

resolver({

value,

result: value \* value

});

},0 | Math.random() \* 1000);

});

}

async function funcionAsincronaDeclarada(){

try{

console.log("Inicio Async Function");

let obj = await cuadradoPromise(0); //await le dice q se espere hasta obtener la respuesta para continuar a la sigueinte linea

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(1);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(2);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(3);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(4);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(5);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

console.log("Fin de Async Function");

}catch(err){

console.error(err);

}

}

funcionAsincronaDeclarada();

//funcion asincrona expresadas

const funcionAsincExpresada = async () => {

try{

console.log("Inicio Async Function");

let obj = await cuadradoPromise(0); //await le dice q se espere hasta obtener la respuesta para continuar a la sigueinte linea

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(1);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(2);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(3);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(4);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

obj = await cuadradoPromise(5);

console.log(´Async Function: ${obj,value}, ${obj.result}´);

console.log("Fin de Async Function");

}catch(err){

console.error(err);

}

}

//----------------49. Symbols-----------------------------------------

nuevos tipos de datos que existen en JS

symbol, 2015, primitivo, se mantiene valor privado y de uso interno, no usamos new, creamos indetificadores unicos

//para identiificar las propiedades de objetos, evitar coaliciones entre propiedades, como sobreescribir

let id = "hola";

let id2 = "hola";

id === id2 - true

//------

let id = Symbol("id"); //id es etiqueta

let id2 = Symbol("id"); //id es etiqueta

id === id2 - false

const NOMBRE = Symbol(); //por q no cambian

const SALUDAR = Symbol();

const persona = {

[NOMBRE]: "Jon" //la notacion de los corchetes nos dice q es un symbol, propiedad privada

}

console.log(persona); //imprimira el objeto Symbol con su valor Jon, pero no imprimira el nombre de la propiedad, se mantiene privada, esa es la caracteris de symbols

persona.NOMBRE = "Jonathan Mircha"; // imprimira el objeto con la propiedad NOMBRE y el symbol jon, no se sobreescribe, se crea una prop normal nueva

persona[NOMBRE] // valor de la referencia de este symbol (Jon)

persona[SALUDAR] = function(){

console.log("Hola");

}

console.log(persona); //imprimira NOMBRE: "Jon...", Symbol(): Jon, Symbol(): f()

persona[SALUDAR](); //ejecucion de la funcion symbol: Hola

//si queremos q el programador tenga una referencia de esos simbolos:

const NOMBRE = Symbol("nombre");

const SALUDAR = Symbol("saludar"); //imprimira Symbol(nombre): "Jon", ....Symbol(saludar): f

//imprime propiedad q esta dentro del obj persona

for(let propiedad in persona){

console.log(propiedad);

console.log(persona[propiedad]);

} //imprimira: NOMBRE Jonathan Mircha, solo las propiedad, por q los symbolos son privados

//como listar los symbolos, los aributos provados del objeto?

Object.getOwnPropertySymbols(persona); // imorimira un objeto de dos symbolos: symbol(nombre), symbol(saludar)

//elementos privados dentro de un objeto y q sus referencias seran unicas -> uso

//-------------------50. Sets------------------ se parecen mas a los arreglos

//areglo q solo accepta valores unicos, valores primiticos no objetos, tipo de dato complejo

const set = new Set([1,2,3,3,4,5,true, true {}, "Hola" , {}]) // imrpime 1,2,3,4,5,{},"Hola",{}

set.size // en lugar de lenght

const set2 = new Set();

set2.add(1);

set2.add(2);

set2.add({});

set2.add(2);

set2.add(3);

//como recorrerlos

for(item of set){

console.log(item);

}

set2.forEach(item => console.log(item))

set[0] - no funciona por q no es un arreglo, es un tipo de dato iterable no un arreglo

//convertimos set a un arreglo para poder sacar su valor por indice

let arr = Array.from(set);

arr[1] //2

//metodo especial para borrar valores

set.delete("Hola");

//revisa si existe

set.has(3); //true

//limpiar set

set2.clear();

//---------------51. Maps---------------- se parecen mas a los objetos

//map son objetos q sirven para almacenar conjunto e3 valores asociados a manera de objeto, tipo de dato complejo,objeto primitivo, parecido a un objeto primitico, coleccion de datos relacionados entre si.

const mapa = new Map();

//agregar valores do formas

mapa.set("nombre","Jon"); //pone o agrega

mapa.set("apellido","Mircha"); // es como si fuera un diccionario, una coleccion, asi se despliega en consola

mapa.size

mapa.has("nombre") //true or false

mapa.get("nombre"); //obtiene

mapa.set("nombre","Jonathan") //sobreescribe

mapa.delete("apellido");

for(let [key, value] of mapa) //uso de destructuracion

{

console.log(´Llave: ${key}, Valor:${value}´);

}

//aqui se pueden general llaves de otros tipos a string (undefined, null...) no recomendable

mapa.set(19,"19")

mapa.set(false,"false")

mapa.set({},{})

//uso, guardar coleccion de datos q estan relacionados, elemento iterable, convertir a array

//otra forma de agregar elementos

const mapa2 = new Map({

["nombre","Kenai"],

["edad", 7],

[null, "nulo"]

});

const llavesMapa2 = [...mapa2.keys()];

const valoresMApa2 = [...mapa2.values()];

//--------------52. WeakSets & WeakMaps

//mapas deviles, y conjuntos deviles

//solo pueden almacenar referencias deviles, llaves de tipo objeto, recolector de basura limpia las referencias deviles, mejor rendimiento

//carencias, no son elementos iterables, no se puden borrar elementos de un jalon, solo de uno en uno, no tiene propiedad size

//weak sets

const ws = new WeakSets([1,2,3,3,4,5,true, true {}, "Hola" , {}]); //error, esto no esta permitico con weaksets, solo en sets

const ws = new WeakSets();

let valor1 = {"valor1":1};

let valor2 = {"valor2":1};

let valor3 = {"valor3":1};

ws.add(valor1);

ws.add(valor2); //se agregan solo referencias, valores de tipo objeto

ws.has(valor1);

ws.delete(valor2);

//Aqui lo q hara es escribe los ws y despues limpia las referencias con null, esta es una forma de eliminar los datos, a travez de las referencias con null

setInterval(() => cosole.log(ws), 1000);

setTimeout(() => {

valor1 = null;

valor2 = null;

valor3 = null;

}, 5000);

//weak maps

const wm = new WeakMap({ //esto es incorrecto

["nombre","Kenai"],

["edad", 7],

[null, "nulo"]

}); //tambien solo se permiten llaves de tipo objeto

const wm = new WeakMap();

let llave1 = {};

let llave2 = {};

let llave3 = {}; //objetos

wm.set(llave1, 1)

wm.set(llave2, 3); //no iterables, no tamaño

wm.has(llave1)

wm.has(llave3)

wm.get(llave1)

wm.delete(llave2)

//Aqui lo q hara es escribe los wm y despues limpia las referencias con null, esta es una forma de eliminar los datos, a travez de las referencias con null, recolector de basura

setInterval(() => cosole.log(wm), 1000);

setTimeout(() => {

llave1 = null;

llave2 = null;

llave3 = null;

}, 5000);

//---------------53. Iterables & Iterators---------------------------

en el dom, nos devuelve nodos del dom, q es como un arreglo, elemento iterable

Iterable: elemento el cual su contenido se recorre

Iterador: ese apuntador q recorre los elementos, mecanismo

array.from for in, for of...

podemos acceder a la interface q trae JS de iterable para recorrer elementos en colecciones de datos

const iterable = [1,2,3,4,5];

//Accedemos al iterador del iterable

const iterador = iterable[Symbol.iterator]();

console.log(iterable); //imprime arreglo de 5 elementos

console.log(iterador); // nos dice que es un Array Iterator{}

//como recorrer los elementos

iterador.next() // value: 1, done: false

iterador.next() // value: 2, done: false

iterador.next() // value: 3, done: false

iterador.next() // value: 4, done: false

iterador.next() // value: 5, done: false

iterador.next() // value: undefined, done: true

//hacerlo en un ciclo para no hacer uno por uno

let next = iterador.next();

while(!next.done){

console.log(next.value);

next = iterador.next();

} //imprimira al Array iterator con : 1,2,3,4,5

//-----

const iterable = new Set[1,2,2,3,3,4,5];

while(!next.done){

console.log(next.value);

next = iterador.next();

} //imprimira al Set iterator con : 1,2,3,4,5, no elementos repetidos por el set

//..................54. Generators--------------

//sintaxis mas sencilla para generar iterables

function\* iterable(){ //le \* le indica a JS q es un generador, sirve para volver iterable una funcion, convertir el codigo de una funcion en iterable

yield "hola";

console.log("Hola consola");

yield "hola2";

console.log("seguimos con mas insrucciones de nuestro codigo");

yield "hola3";

yield "hola4";

}

let iterador = iterable(); //por ser una funcion generador ya conoce el Symbol q se usa para usar el iterador, la funcion actua como asyncrona, ejecuta tarea por tarea

console.log(iterador.next());// value: "hola", done: false - Hola consola, los console los imprime normal, los yield se van esperando a ejecutar

console.log(iterador.next());// value: "hola2", done: false - seguimos con mas insrucciones de nuestro codigo

console.log(iterador.next());// value: "hola3", done: false

console.log(iterador.next());// value: "hola4", done: false

console.log(iterador.next());// value: undefined, done: true

// tambien con un ciclo

let iterador = iterable();

for(let y of iterador){

console.log(y)

}

//imprimee:

// value: "hola", done: false - Hola consola, los console los imprime normal, los yield se van esperando a ejecutar

// value: "hola2", done: false - seguimos con mas insrucciones de nuestro codigo

// value: "hola3", done: false

// value: "hola4", done: false

// value: undefined, done: true

//uso si estamos esperando valores de una base de datos, pero necesitamos tener todos los datos antes de procesar la informacion, usamos generadores ya que los yield aseguraran

//esperar hasta tener todos los datos y continuar, lo guardamos en un arreglo todos los resultados q nos dan los yiel

const arr = [...iterable()]; // ["hola","hola2","hola3","hola4"] // obtendra todos los resultados de los yiel y asi ya ocuparlos

//ejemplo con ejercicio cuadrado

function cuadrado(valor){

setTimeout(() => {

return console.log({valor, resultado: valor \* valor});

}, Math.random() \* 1000);

}

function\* generador(){

console.log("Inicia Generator");

yield cuadrado(0);

yield cuadrado(1);

yield cuadrado(2);

yield cuadrado(3);

yield cuadrado(4);

yield cuadrado(5);

console.log("Inicia Generator");

} //funcion q lo hace manual

//imprime:

Inicia Generator

valor: 0, resultado: 0

valor: 1, resultado: 1

valor: 2, resultado: 4

valor: 3, resultado: 9

valor: 4, resultado: 16

valor: 5, resultado: 25

Inicia Generator

let gen = generador(); //con un ciclo rapidamente se hace

for(let y of gen){

console.log(y);

}

imprime: //tener cuidado por q imprime no como las promesas o funciones asincronas, paso a paso, era bloqueante,

//pero aqui es asincronia no bloquenate, uno acaba antes que otro, y no espera

valor: 2, resultado: 4

valor: 3, resultado: 9

valor: 4, resultado: 16

valor: 5, resultado: 25

valor: 1, resultado: 1

valor: 3, resultado: 9

valor: 4, resultado: 16

//----------55: Proxies-----------------

//tiene q ver con la POO, el proxi es un nuevo mecanismo q permite crear un objeto basado en un objeto literal inicial. similar a las instamcias de las clases.

// proxies, va a crear una copia de un objeto literal, asi como un objeto se genera a partir la una clase

//el proxi hace una vinculacion entre el objeto original y el objeto copia, y atravez de su manejador se hacen validaciones antes de hacer su implementacion

//handler - manejador

const persona = { //objeto original

nombre: "",

apellido: "",

edad: 0

}

const manejador = { //parametro propio del manejador llamado set

set(obj, prop, valor){ //aqui podemos poner todo tipo de validaciones, podemos restringir para q no sufra cambios el objeto persona

obj[prop] = valor; //Aqui le decimos q la copia osea el proxi tendra los valores asignados al objeto en el proxi jon, el proxi estara vinculado con persona tambien, osea tambien

//tendra los valores asignados al objeto

//valida si existe esa propiedad en el objeto original persona

if(Object.keys(obj).indexOf(prop) === -1){

return console.error(´La propiedad "${prop}" no existe en el objeto persona´);

}

//tambien se pueden hacer validaciones normales

if( (prop === "nombre" || prop === "apellido") && !(/^[A-Za-zÑ\s]+$/g.test(valor)) ){

return console.error("La propiedad ${prop} solo acepta letras y espacio en blanco");

}

obj[prop] = valor

}

}

const jon = new Proxy(persona, manejador) //objeto copia

jon.nombre = "Jon";

jon.epellido = "Mircha";

jon.edad = 35;

//imprimira: Proxy {nombre: "", apellido: "", edad: 0}, tiene el handler y un tarjet, el tarjet es la vinvulacion q se hizo con persona

//--------------56. Propiedades Dinamicas de los Objetos (compute properties)

//crear prop al vuelo, general el nombre de las prop de manera dinamica

let aleatorio = Math.round(MAth.random() \* 100 +5);

const objUsuarios ={

propiedad: "Valor",

[´id\_${aleatorio}´]: "valor aleatorio" //tambien se pueden generar prop dinamicas al objeto original

}

const usuarios = ["Jon", "Irma", "Miguel", "Kala", "Kenai"];

//por cada usuario

usuarios.forEach((usuario, index) => objUsuarios[´id\_${index}´] = usuario); //propiedad dinamicas generadas al objUsuarios, su uso podria ser general variables en tiempo real

//---------57. this--------------------------------

//this hace referencia al objeto en cuestion

this - > hace referencia al objeto global, obj window

this === window .ambito global

this.nombre = "Contexto Global"; // encontraremos en el objeto window la prop nombre

function imprimir(){

console.log(this.nombre ); //contexto global

}

imprimir(); // contexto global

const obj = {

nombre: "Contexto Objeto",

imprimir: function(){

console.log(this.nombre); //imprimira Contexto Objeto, por q esta en el contecto de este obj

}

}

obj.imprimir(); //imprimira Contexto Objeto

const obj2 = {

nombre: "Contexto Objeto 2",

imprimir // ? aqui imprimira Contexto Objeto 2 y no Contexto Global, pues aun q llama a la funcion imprimir, el this es de este obj2

}

}

const obj3 = {

nombre: "Contexto Objeto3",

imprimir: () => {

console.log(this.nombre); //imprime "Contexto Global", cuando se usan las arrow function tener cuidado, por q no manejan su propio scope, entonces traera el this de la funcion imprimir original

}

}

function Persona(nombre){

this.nombre = nombre;

return function(){ //clousures, funcion dentro de funcion, //aqui imprimira Contexto Global y no Jon, por que esta funcion aninima crea un nuevo scope y no conoce al nombre y por lo tanto traera al nombre global

console.log(this.nombre);

}

//solucion, hacer una arrow function, q no tiene scope y tomara el nombre de jon

return () => console.log(this.nombre)

//antes se solucionaba con that

that.nombre = nombre

return function(){ //y aqui ya tiene el that de este contexto : Jon

console.log(that.nombre);

}

}

let jon = new Persona("Jon");

//-----------58. call, apply, bind---------------

//JS ES5 ofrece call, apply y bind para conservar la referencia de un scope en particular y utilizarlo en una invocacion de algun codigo

console.log(this); //window

this.lugar = "Contexto Global"

function saludar(){

console.log(´${saludo} ${aQuien} desde el ${this.global}´); //contexto global

}

saludar("Hola", "Kenai"); //Hola Kenai desde el contexto global

const obj = {

lugar: "contexto objeto"

}

saludar.call(obj, "Hola", "Jon"); //Hola jon desde el contexto objeto

saludar.call(null, "Hola", "Jon"); //Hola Jon desde el contexto global

saludar.call(this, "Hola", "Jon"); //Hola Jon desde el contexto global

//hace lo mismo la diferencia el parametros a modo de array

saludar.apply(obj, ["Adios", "Mircha"]); //Adios Mircha desde el contexto objeto

saludar.apply(null, ["Adios", "Mircha"]); //Adios Mircha desde el contexto global

saludar.apply(this, ["Adios", "Mircha"]); //Adios Mircha desde el contexto global

//bind - enlace

const persona = {

nombre: "Jon",

saludar: function(){

console.log(´Hola ${this.nombre}´)

}

}

}

this.nombre = "Window";

persona.saludar();

const otraPersona ={

saludar: persona.saludar.bind(persona); //enlaza el contexto de persona, llamar a contextos diferentes a los q estoy trabajando

saludar: persona.saludar.bind(this); //enlaza this.nombre = "Window"

}

//--------------59. Json---------------

//Json: JS object notation, formalo lijero de intercambio de datos, maquinas facil de interpretarlo, entre lenguajes se pueden comunicar con json, antes xml ->sistema tributario de mex

// json no es exclusivo de JS, por douglas crockfor, todos los lenguajes tiene soporte a json

//estandar de intercambio de informacion

//el archivo .js el nombre de las propiedades y los valores deben ir entre comillas

//JS es un estandar del lenguaje, front y back

//json es una interfaz del lenguaje

//parse, analiza una cadena de texto y la convierte a un tipo de dato q JS valide (string, array, etc)

"{}" si no le pongo parse, imprimera la cadena {}, si le pongo JSON.parse("{}") ahora si es un objeto {}, siempre y cuando es valido para Json

//stringify: convierte un objeto valido de JS y lo convierte a string, contrario de parse

JSON.stringify({}); // imprime la cadena "{}"

JSON.stringify({x:2, y:3}); //imprime: {"x": 2, "y": 3}

//--------------------60. WEB APIs-----------------

Web APIs Application Programming Interface: serie de objetos y mecanismos que tiene el lengiaje implementado (JS de los navegadores) para poder interactuar con

el navegador (DOM) barra de direcciones, ventana, posicionamiento, SO donde esta el usuario,

CSSOM (CSS Object Model),

BOM (Brownser Object Model) (Non standard)

WEB APIs-----------------

\* Eventos

\* Forms

\* AJAX - Fetch

\* History

\* Web Storage

\* Geolocation

\* Drag and Drop

\* Indexed DB

\* Canvas

\* MAtchMEdia (controla las media queries de CSS)

\* Etc n cantidad de web apis

Window

\* navigator

platfrom

language

userAgent Mozilla

History

indexdb

Location

pathname

localstorage - almacenamiento localstorage

document, q es el arbol del dom que mapea la estructura del html

alert

interfaz de spech, comando de voz se envia a la compuesto

let texto = "Hola";

const hablar = (texto) => speechSynthesis.speak(new SpeechSynthesisUtterance(texto));

hablar(texto);

//------------61. DOM-----------------------------