# Intro

Es un lenguaje de programacion Dinamico (las variables no estan arraigadas a un tipo de dato particular), interpretado(usualmente barre linea por linea de manera secuencial) que se creo con el fin de prevenir envio de informacion innecesaria al servidor, es case sensitive, antes de la llegada de Node (que permite correr JavasCript fuera del navegador web) se ejecutaba solo del lado del cliente. No puede ser utilizado para aplicaciones de red por si solo(Para esto se puede usar por ejemplo Ajax). No tiene capacidad multihilo o multiples procesos simultaneos(No es asincrono), solo puede ejecutar un procedimiento secuencial a la vez.

No es buena practica mezclar el HTML con JS hay que tratar de evitarlo. Los tags de <scripts> se deben colocar ultimos, antes del cierre del <body> en el HTML para que cuando llegue al archivo js ya haya recorrido y renderizado todo el html y encuentre todo lo necesario para correr el JS.

El motor de Javascript lo primero que hace es barrer todo el documento, crear el objeto global Window

y luego ejecuta cada comando. Por esto puede haber objetos undefined hasta que se les asigna algun valor xq se los inicializa con undefined

Colocar ;(punto y coma) luego de cada sentencia no es obligaotior pero si es buena practica ya que si no lo hacemos pueden ocurrir errores en la interpretacion de nuestro codigo por parte del compilador

# Esctructura de carpetas

Se acostumbra colocar una carpeta llamada **assets** en la raiz y dentro de esta una carptea **css** y otra **js**

# **Alert, Prompt y Confirm**

Son cuadros de dialogo, estan dentro del objeto Window, alert() es solo para mostrar, prompt() para ingresar algun dato y confirm() para confirmar o no algo

# Tipos primitivos y objetos

Primitivos: Number, string, bool, undefined, null. Sus valores siempre se pasan por valor

Objetos definido - con nomenclatura de pares: sus valores siempre se pasan por referencia

var obj = {

numero:10,

nombre=”Pablo”

};

Objeto Anonimo: {

numero:10,

nombre=”Pablo”

};

Un objeto puede tener otro objeto dentro. (Cadena de objetos)

# Notacion de Corchete

es necesaria cuando queremos apuntar a una determinada parte de un objeto y cuyo valor ya tenemos por ejemplo:

var campo =”edad”;

console.log( persona[campo] );

# Funciones(no existen procedimientos o metodos)

Todas regresan algun valor si no tiene nada dentro regresa undefined. Si una variable u objeto no existe en el entorno(scope) de la funcion Javascript la busca en el contexto global

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

    this.nombre = nombre;

    this.apellido =  apellido;

    this.edad = edad;

    this.imprimirPersona = *function*(){

        return nombre +" "+ apellido +". Edad: "+ edad;

    }

}

*var* persona = new Persona("Pablo","Sanabria",32);

console.log(persona.imprimirPersona())

Funciones anonimas

(*function*(){

    console.log("anonima")

})();

Los parametros pueden ser cualquier cosa hasta incluso otra funcion (fn). Tambien pueden retornar otra funcion:

*function* crearFuncion(){

    return *function* (*persona*){

        console.log("Funcion 1 creada por " + persona.nombre);

        return *function*(){

            console.log("Funcion 2 creada");

        }

    }

}

*var* nuevaFuncion = crearFuncion();

*var* segFuncion = nuevaFuncion(persona);

segFuncion();

Podemos colocarles propiedades y metodos de cualquier tipo, ya que las funciones son objetos:

crearFuncion.estado = “Creada”;

## Funciones anonimas y callbacks

Las funciones anonimas o definidas nos permiten encapsular nuestro codigo y prevenir que accidentalmente sea manipulado por otras secciones del programa. El principal o uno de los usos mas importantes de las funciones anonimas son los callback por que nos permiten crear funciones y pasarlas a otras sin tener que definirlas con anterioridad, esto las hace mas flexibles. El acto de pasar una funcion a otra, se le denomina callback. Los callbacks pueden ser tanto anonimas como no anonimas. Sin embargo en casi la mayoria de ocaciones hacemos callbacks con funciones anonimas, esto resulta mas versatil.

# ===

Compara valor y tipo de objeto

# variable = variable || “xxx”;

Si variable es undefined se le asigna el string

# This

Cada vez que se crea un contexto se genera un objeto this para ese contexto. Siempre es bueno ver que es lo que tiene el objeto this del contexto que querramos ver console.log(this)

var self = this;

Se utiliza cuando por algun motivo(aun no conocido) el objeto this apunta al contexto global y necesitamos que apunte al contexto desde donde lo estamos invocando por ejemplo una funcion dentro de de 2 funciones

# New

Sirve para crear un objeto vacio pero con un tipo especifico, similar a una clase en C#, El valor de this dentro de la funcion es un nuevo objeto y ese nuevo objeto es retornado:

*function* Jugador(*nombre*){

    this.nombre = nombre;

    this.pv = 100;

    this.sp = 100;

    this.estado = *function*(*jugadorObjetivo*){

        console.log(this);

        console.log(jugadorObjetivo);

    }

    this.tirarFlecha = *function*(*jugadorObjetivo*){

        if(jugadorObjetivo.pv>40){

            jugadorObjetivo.pv -= 40;

            this.estado(jugadorObjetivo);

        }else{

            jugadorObjetivo.pv = 0;

            console.info(jugadorObjetivo.nombre + " eliminado");

        }

    }

}

*var* gandalf = new Jugador("Gandalf");

*var* legolas = new Jugador("Legolas");

console.log(gandalf, legolas)

gandalf.curar(legolas);

# Prototipos

Son un conjunto de normas para intregrar la POO a Javascript. Con ellos podemos relizar Herencia,Encapsulamiento,Polimorfismo, Abstraccion. Dependiendo del tipo del objeto/variable se les asigna determinados prototipos. Podemos customizarlos, aunque no es recomendable, por ejemplo

Number.prototype.esPositivo = function(){

if(this > 0)

return true;

else

return false

};

Permiten expandir los objetos, sirve cuando queresmos agregar un metodo por fuera de la definicion del objeto/clase por ejemplo a miles de usuarios obtenidos de una BD

*Jugador*.prototype.curar = *function*( *jugadorObjetivo*){

    if(this.sp>=40){

        this.sp -= 40;

        jugadorObjetivo.pv +=  20;

    }else{

        console.info(this.nombre + " no tiene mas SPs")

    }

    this.estado(jugadorObjetivo);

}

# Operadores de comparacion

Los operadores === y !== son los operadores de comparación estricta. Esto significa que si los operandos tienen tipos diferentes, no son iguales. Por ejemplo,

1. 1 === "1" // false
2. 1 !== "1" // true
3. null === undefined // false

Los operadores == y != son los operadores de comparación relajada. Es decir, si los operandos tienen tipos diferentes, JavaScript trata de convertirlos para que fueran comparables. Por ejemplo,

1. 1 == "1" // true
2. 1 != "1" // false
3. null == undefined // true

Vale mencionar que el operador == no es transitivo, al contrario que ===.

1. "0" == 0 // true
2. 0 == "" // true
3. "0" == ""// false

# Funciones typeof e instanceof

if (typepof param == “function”) : nos indica el tipo, devuelve un string

param instanceof Persona: Compara 2 objetos y devuelve true or false

Validacion para saber si un objeto fue declarado con New:

if(this instanceof Persona){

    //inicializar o asignar valores al objeto

    this.nombre = nombre;

}

else

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

# Arreglos

Son una colección de objetos

pueden contener todo tipo de datos incluso mezclados:

var arr = [

true,

1,

“2”,

{},

function(){

console.log(“arreglos”)

}

[1,2,3,4]

]

No hay errores de desbordamiento si apuntamos a una parte del arreglo que no existe nos devuelve undefined.

Algunas de las funciones mas comunes son: reverse(), map(), join(), split() o toString() este ultimo es mas eficiente, push() a diferencia de otros lenguajes en Javascript se pueden agregar elementos unshift(), pop(), splice(), slice(),

podemos tener arreglos infinitos dentro de un arreglo: [4][4][1] al ver algo si tenemos que saber que cada posicion nos indica un nuevo arreglo: posicion 4 del primer arreglo, posicion 4 del segundo arreglo y posicion 1 del tercer arreglo.

# Arguments

Son propios de cada funcion, siempre hacen referencia a los valores de los parametros que se indican en el momento de la invocacion de la funcion (aunque no estén definidos explícitamente) y no de la declaracion.

*miFuncion(10,20);*

function miFuncion(){

console.log(arguments);

}

Podemos mandar de todo:

*miFuncion(10, {}, function(){…});*

# Sobrecarga de operadores

No existe en javascript pero podemos hacer lo siguiente para simularla, llamando a la funcion original dentro de la customizada:

*function* crearProducto(*nombre*, *precio*){

    nombre = nombre || "sin nombre";

    precio = precio || 0;

    console.log( nombre + " " + precio) ;

}

*function* crearProducto100(*nombre*){

    crearProducto(nombre,100);

}

*function* crearProductoCamisa(*precio*){

    crearProducto("Camisa",precio);

}

crearProducto();

crearProducto100("Lapiz");

crearProductoCamisa("150");

Polimorfismo

No es mas que una funcion que puede recibir diferentes tipos de parametros o datos ,es buena practica saber que tipo de dato estamos recibiendo:

*function* determinarDato(*a*){

    if (a === undefined)

        console.log("parametro undefined");

    if ( typeof a === "number")

        console.log("parametro numerico");

    if ( typeof a === "string")

        console.log("parametro texto");

    if ( typeof a === "object"){

        //console.log("parametro objeto, puede ser cualquier tipo");

        //se puede consultar por el tipo de objeto cusomizado por ejemplo Persona

        if (a instanceof Persona)

            console.log("parametro objeto del tipo Persona");

        if (a instanceof Number)

            console.log("parametro objeto del tipo numerico");

        if (a instanceof String)

            console.log("parametro objeto del tipo texto");

        if (a instanceof Boolean)

            console.log("parametro objeto del tipo bool");

    }

}

*var* b = new Number(3);

determinarDato(persona);

# Contexto o Scope

El ambito de una variable es la zona del programa en la que se define la misma. Javascript define 2:

Contexto Global y de Funcion

# Contexto en Funciones

En JavaScript las funciones son muy especiales porque pueden ser asignados a variables, pasarse como argumentos, devolverse desde funciones, etc. Debido a esto se dice entonces que las funciones son de primera clase, porque pueden ser tratadas como cualquier otro valor.

## **Funciones declaradas.**

Cuando esta se define en el programa, de manera individual, con la  estructura "estandar".

1. function miFuncion() {
2. //Bloque de código
3. }

## **Expresión de función**

¡Una función también se puede asignar! Eso significa que podemos usar una función como una expresión y almacenarla en una variable. Con esto se puede concluir que una función puede ser tratada como un valor más.

1. var miFuncion = function () {
2. //Bloque de codigo
3. };

Lo que se le asignó a la variable es una **funcion anónima**. Sin nombre.

Se suele usar funciones anónimas cuando las funciones se pasan a variables o a otras funciones. Sin embargo, hay que replantearse quitarles el nombre por razones de otro mundo: el debugging. Pero bueno, ese es harina de otro costal.

En programación una expresión es una unidad de código que al evaluarse produce un valor. Una variable, por ejemplo, es una expresión, por que al evaluarse nos entrega el valor que almacena; una operación aritmética es un expresión: una combinación de operandos y operadores que al ser evaluados producen un valor.

Una **expresión de función** es eso, una expresión que evalúa a una función. Eso significa que una vez asignada, ¡es un valor más! Por eso al final de la asignación de la función en una variable se pone punto y coma, a diferencia de la declaración de función que no produce o evalua a ningún valor. Solo le dice al interprete, ¡Hey, aquí defino esta función para más tarde!

## IFFE: Expresión de función ejecutada inmediatamente

Cuando una función se usa como expresión, al evaluarse produce un valor: la función. Sin embargo, podemos meternos en el camino justo cuando sucede eso y ¡ejecutarla inmediatamente!. A esto se le conoce como función inmediatamente o automáticamente invocada.

Antes:

1. var miFuncion = function () {
2. console.log("naci!");
3. };
5. miFuncion(); // "naci!"

Ahora:

1. function () {
2. console.log("naci!");
3. }()

Pero como la llamo si no tiene nombre, ¿Alguna ayuda?

Esto se acerca, pero no es valido. Aquí algo muy importante es que al no estar siendo asignada a una función y tampoco estar siendo definida, el interprete no puede entender lo que dices(no quiere decir que no sea una expresión). Entonces, al estar alejado de lo que se hacía antes, podemos decir al interprete que tome esa función como una expresión cuando no este siendo ni asignada ni definida:

1. (function () {
2. console.log("naci!");
3. })

Lo que hicimos ahora es decirle al interprete que esa función es una expresión, para que entienda que al no estar en ninguno de los dos casos de antes, siga entendiendo que es una expresión, ya que si no lo hiciéremos lo tomaría como error sintáctico. ¡ Es un tipo raro, como una expresión y a la vez como una sentencia/instrucción!

Ahora tenemos una función anónima flotando por ahí, a la deriva, sin rumbo. Lo que debemos hacer es activarla inmediatamente. Porque si no lo hacemos ahora, más tarde mucho menos, pues no habría forma de llamar a algo sin nombre.

1. (function () {
2. console.log("naci!");
3. })();

!Listo! Lo que dice arriba es "trata esta función como expresión— porque no esta ni asignada ni declarada--, y una vez hecho esto llámala(recuerda, una vez evaluada la expresión, producirá un valor, en este caso, la función). De no hacerlo, el valor (la función) quedara a la deriva.

A esto precisamente se le conoce como función auto invocada o automáticamente invocada. Es una función que se crea y llama inmediatamente, en un solo paso.

## **Parámetros**

Indican qué información del exterior necesita la función para ser ejecutada.

Los argumentos, por el contrario, son los valores actuales o expresiones que se le pasan a la función en el momento de ser llamada o invocada.

Nuestro iffe será de nuevo valido para el interprete también posee estas características de cualquier función, pero hay que adaptarla a su nueva forma.

Si antes le definíamos los parámetros en la cabecera de la función, sigue siendo de la misma manera.

1. (function (creador) {
2. console.log("naci!");
3. })();
4. //COMPARACION
6. var funcion = function(creador){
7. console.log("naci!");
8. }

11. function funcion(creador){
12. console.log("naci!")
13. }

¿Los argumentos se pasaban al momento de la llamada cierto? Ahora, como nuestra llamada esta justo debajo de la función, ¡entonces ahí irán!

1. (function (creador) {
2. console.log("naci y me creo " + creador);
3. })("orlando");
4. //COMPARACION
6. var funcion = function(creador){
7. console.log("naci y me creo " + creador);
8. }
10. funcion("orlando");
12. function funcion(creador){
13. console.log("naci y me creo " + creador);
14. }
16. function("orlando");

¡Mira, ahí esta! Funcionado como una función cualquiera.

1. (function (creador) {
2. console.log("naci y me creo " + creador);
3. })("orlando"); // "naci y me creo orlando"

## **El contexto**

El contexto es el entorno de una función en ejecución. Es decir, el contexto gestiona toda la información relacionada a una función en ejecución, ya sean sus variables locales, parametros, scope, etc..

Es muy importante porque es quien orquesta y verifica que la función tenga lo necesario para funcionar correctamente.

Cuando ejecutas una función esta empieza su ejecución: nada raro. Sin embargo, los lenguajes de programación se apoyan sobre algunas estructuras en memoria, como siempre.  En javascript a esta se le conoce como pila de contextos de ejecución o execution context stack.

¿Que es una stack?

Una estructura de datos. ¿Que es una estructura de datos? Es un método o manera de organizar la información en memoria, en una computadora.

¿Que hace un stack? Un stack puede verse como un librero. Los libros siempre se ponen uno sobre otro, no uno abajo de otro. También los libros se sacan desde arriba—a menos que quieras tirarlos todos, no lo sé—.

De esa manera funciona un stack frame, por eso se le conoce como una estructura de datos lifo(last in, first out), último en entrar, primero en salir Así maneja a las funciones:

—  Hey, stack. Aquí Juanito llama a esta función.

—  Okey, pasámelo— dijo el stack frame.

¿Que es lo importante del contexto? Que cada función, sin exclusión, tiene el suyo.

Se relaciona mucho o está más acentuado su comportamiento en las IFEE, porque como son funciones que se auto invocan, tambien crean sus contextos de manera automática y aislada de otras funciones comunes.

Frecuentemente— o más de lo que se puede esperar— usamos ifee's por su capacidad de aislar su contenido ante otro codigo y por tener la sintaxis simple como para inferir que se trata de un cojunto de codigo en si mismo, una unidad.

**Regresando a ejemplo de la clase.**

Creo que con lo que he explicado ahora ya entenderás el 60% del código del que tenias dudas, pero hay algo más.

Te saco el extracto de mi respuesta anterior:

Cuando tú retornas una función de otra función lo que estas haciendo se le conoce como **closure**. Este closure puede manejar las variables locales de la función desde donde fue creado(recuerda el su ámbito). Pero como el array fue llamado al final, entonces la ultima configuración de la variable fue 3(recuerda que todo debajo de un return no se ejecutará por lo que los closures tendrá ese mismo valor(numero).

La razón por la que no toma lo valores que vamos reasignado es por que solo se conoce el valor hasta que el array de función ha sido devuelto y ha tomado el valor de numero.

¿Si usamos IIFE's por que funciona?

Funciona por que en lugar de almacenar solo la función y que esta busque el último valor de numero, lo que hará es que creara una nueva referencia (como variable local) de numero y entonces cada función manejara una variable independiente.

En resumen. Lo que hace la función creafunciónes, es esto. Dice --hey crea este arreglo vacío; establece numero a 1.Luego agrega esta función al array pero que esta función imprima el valor de numero(1). Así hasta el final.

La razón por que falla es que al haber almacenado la funciones al arreglo, estas tienen que saber el valor de numero. Cuando la función esta a punto de acabar se re enlaza el valor de numero de cada función al ultimo valor que este tuvo. Por eso todas las funciones imprimen el mismo numero.

La razón por la que función en ifees es por que cada vez que llega a una, hace una configuración del contexto, de manera individual. Por lo que cada función auto invocada posee su propia referencia a **numero**.  
  
Una cosa que anotar, antes de que lo olvide, es que el return función indica que se devuelve una función nueva y al iffe tener su propia referencia, dentro de su contexto de numero, esta se la pasara a la función y la función tendrá, por defecto, una referencia nueva numero.

# Number

.toFixed(2) -> Coloca o corta la cantidad de decimales que le enviemos por parametro , en este caso 2, Maximo 20.

.toPrecision(2) - > puede devoler notacion exponencial dependiendo del numero. reduce el numero a la cantidad de caracteres enviada

NaN= Non a Number

.valueOf() -> devuelve el valor primitivo

# Boolean

var a = new Boolean(); -> por defecto devuelve false, si ponemos un string no vacio por parametro devuelve true, colocando 1 devuelve true y cero devuelve false, null:false, NaN:false, infinity:true, undefined:false.

con a.valueOf() obtenemos el valor de este objeto, en los if debemos colocar esto si no se evalua que el objeto esté definido. Si es una variable comun lo hace normalmente

# String

Es un objeto en donde cada propiedad es un carácter

.indexOf() -> devuelve la primera posicion del carácter o palabra pasado por parametro

.lastIndexOf() -> devuelve la ultima posicion del carácter o palabra pasado por parametro

# Date

en javascript los meses estan en base cero, enero=0, febrero =1 …

fecha.setDate(); setea el dia que le enviemos por parametro a menos que sea mayor al ultimo dia del mes en ese caso suma los dias restantes al mes siguiente

Para facilitar las operaciones con fechas podemos modificar el prototipo

*var* fecha = new Date(2024,3,6);

console.log(fecha);

*Date*.prototype.sumarDias = *function*(*dias*){

    this.setDate(this.getDate() + dias);

    return this;

}

console.log(fecha.sumarDias(10));

y tambien podemos hacerlo para años, meses, etc…

Librería recomendada Moment.js

# Math

contiene calculos matematicos

# Expresiones Regulares

Son utilizadas para buscar patrones en strings

//i = insensible

//g = todas las apariciones

//m = multilinea para "Enters"

*var* reg1 = RegExp("a");

*var* reg2 = /m/i;

*var* texto = "Hola Mundo";

*var* arr = texto.match(reg2);

console.log(arr)

Documentacion: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_expressions>

# Manejo de Errores

try { //el try necesita un catch o un finally

    throw new Error("error manejado");//forzamos al catch

} catch (e) {//es opcional

    console.log(e);

    registroError(e.message);

}

finally{//si no se utiliza catch debe ir finally

    console.log("finally");

}

*function* registroError(*e*){

*var* diaError = new Date();

    console.log("Se registró un error: ",e ," a las ", diaError.getTime() );

}

# Cookies

Es informacion que podemos guardar en la maquina del cliente. Solo se crean si corremos en un servidor. Idioma, mail. usuario, cosas generales que no sean sensibles. Sobreviven los reinicios del navegador podemos ponerle fecha de expiracion.

### Para instalar nuestro server Local:

instalar node-js y npm con el cmd:

<https://www.npmjs.com/package/http-server>

http-server C:\Users\pablo.sanabria\Documents\Javascript -p 8081

Nota: no colocar una ruta con espacios

### Creacion

Escape y unescape estan deprecados ver [decodeURIComponent()](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/decodeURIComponent) o [decodeURI()](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/decodeURI).

*function* crearCookie(*nombre*, *valor*){

    //Ojo con los caracteres por ejemplo el punto y coma

    //para eso escapamos los valores para cambiar los valores que puedan llegar a dar problemas por su version HTML

*var* hoy =  new Date();

    hoy.setMonth(hoy.getMonth()+1);//vencimiento en un mes

    valor = escape(valor);

    //var valorOriginal = unescape(escape(valor)); Inversa

    document.cookie=nombre +"="+ valor+";expires"+hoy.toUTCString()+";";

}

*function* borrarCookie(*nombre*){

*var* hoy =  new Date();

    hoy.setMonth(hoy.getMonth()-1);//vencimiento en un mes

    document.cookie=nombre +"="+"x;expires"+hoy.toUTCString()+";";

}

*function* getCookie(*nombre*){

*var* cookies = document.cookie;

*var* cookiesArr = cookies.split("; ");

    console.log(cookiesArr);

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

*const* parArr = cookiesArr[i].split("=");

        cookiesArr[i] = parArr;

        if(parArr[0] === nombre ){

            return unescape(parArr[1]);

        }

    }

    return undefined;

}

console.log(getCookie("nombre"));

//crearCookie("nombre","Pablo");

//borrarCookie("nombre","Pablo");

# Funciones especiales: Call(), Apply() y Bind()

Sirven para lo que se llaman **funciones prestadas, por ejemplo con un objeto utilizar las funciones de otro objeto.**

Bind() solo se le indica a la funcion a que objeto va a estar apuntando el this pero no se ejecuta como tal la funcion, y ya con el aplly() y call() directamente se puede ya ejecutar la funcion indicando a que objeto estará apuntando el this

**Call**

1. var person1 = {firstName: 'Jon', lastName: 'Kuperman'};
2. var person2 = {firstName: 'Kelly', lastName: 'King'};
4. function say(greeting) {
5. console.log(greeting + ' ' + this.firstName + ' ' + this.lastName);
6. }
8. say.call(person1, 'Hello'); // Hello Jon Kuperman
9. say.call(person2, 'Hello'); // Hello Kelly King

**Apply**

1. var person1 = {firstName: 'Jon', lastName: 'Kuperman'};
2. var person2 = {firstName: 'Kelly', lastName: 'King'};
4. function say(greeting) {
5. console.log(greeting + ' ' + this.firstName + ' ' + this.lastName);
6. }
8. say.apply(person1, ['Hello']); // Hello Jon Kuperman
9. say.apply(person2, ['Hello']); // Hello Kelly King

**Bind**

1. var person1 = {firstName: 'Jon', lastName: 'Kuperman'};
2. var person2 = {firstName: 'Kelly', lastName: 'King'};
4. function say() {
5. console.log('Hello ' + this.firstName + ' ' + this.lastName);
6. }
8. var sayHelloJon = say.bind(person1);
9. var sayHelloKelly = say.bind(person2);
11. sayHelloJon(); // Hello Jon Kuperman
12. sayHelloKelly(); // Hello Kelly King

Call y apply son bastante intercambiables. Simplemente decida si es más fácil enviar una matriz o una lista de argumentos separados por comas.

Siempre recuerdo cuál es cuál recordando que Call es para coma (lista separada) y Apply es para Array.

Bind es un poco diferente. Devuelve una nueva función. Call y Apply ejecutan la función inmediatamente.

Bind es genial para muchas cosas. Podemos usarlo para curry functions como en el ejemplo anterior. Podemos tomar una función hello simple y convertirla en helloJon o helloKelly. También podemos usarlo para eventos como onClick, en los que no sabemos cuándo se activarán, pero sabemos qué contexto queremos que tengan

# JSON

*var* objeto = {

    nombre : "Pablo",

    apellido : "Sanabria",

    edad : 37

};

console.log("Objeto Literal: ", objeto);

*var* objJson = JSON.stringify(objeto);

console.log( objJson);

*var* objDesdeJson = JSON.parse(objJson);

console.log( objDesdeJson);

# Ciclo For In

Se usa para iterar en cada una de las propiedades y los prototipos de un objeto, similar al foreach()

for( prop in persona){

    console.log( prop , persona[prop]);

}

Si solo quiero iterar las prop y no los prototipos, esto se conoce como reflection: es la habilidad que tienen los objetos de conocerse asi mismos

for( prop in persona){

    console.log("Props: ")

    if(persona.hasOwnProperty(prop))

        console.log( prop , persona[prop]);

}

# Foreach

para arreglos

*var* arrIte = [1,2,3,4,true,6,7,"Ocho",9];

arrIte.forEach( *function*(*val*){

    console.log(val);

});

# Rotulacion de ciclos

Se usa para saltos entre ciclos:

for\_principal:

for (*let* index = 0; index < 3; index++) {

    console.log("index ",index);

    for\_secundario:

    for (*let* x = 0; x < 3; x++) {

        console.log("x ",x);

        for (*let* i = 0; i < 3; i++) {

            console.log("i ",i);

            break for\_secundario;

        }

    }

}

# Funciones de tiempo

setTimeout() y setInterval() esta ultima necesita que le definamos un corte con clearInterval()

# Eventos

*function* evento(*arg*){

    console.log("Evento disparado");

    console.log(arg);//codigo de la tecla

    //muestra:

    //KeyboardEvent {isTrusted: true, key: 'e', code: 'KeyE', location: 0, ctrlKey: false, …}

    //isTrusted: true quiere decir que el evento lo disparó el usuario

    //console.log(arg.x, arg.y);//posicion en pantalla de donde se hace click en un boton por ejemplo

}

*var* elemento = document.getElementById("txtDemo");

elemento.addEventListener("keypress",evento);

onsubmit=”return validar()” ya hay librerias hechas para la validacion del form

Obtener los parametros de la url:

*function* getParamURL(*name*) {

  return decodeURIComponent((new RegExp('[?|&]' + name + '=' + '([^&;]+?)(&|#|;|$)').exec(location.search)||[,""])[1].replace(/\+/g, '%20'))||null

}

# Pup ups

<https://sweetalert2.github.io/>

buscar por que hay muchas

# use strict" - Modo estricto

Existe un modo estrcito para que no se permitan hacer ciertas cosas pero debemos confiar en ella al 100% por que no todos los navegadores lo interpretan igual para que funcione en un archivo de manera comleto debemos encapsuar el codigo del mismo con una funcion anonima