**Funciones de contacto con el usuario:**

Veremos la funciónes nativas de JS como prompt() que hace esta función exactamente? Solicita o envía un mensaje al usuario que responda a determinados eventos al igual alert() o confirm() además confirm() devuelve true si cinfirmas o false si canselas. En cualquiera de los casos se espera una respuesta →prompt("Introduce tu nombre de usuario") Lo sacamos por consola →console.log(prompt("Introduce tu nombre de usuario:?")); Si quisiéramos un retorno ante ponemos la funcion→alert( prompt("Introduce tu nombre de usuario:?")); alert retorna el resultado de la operación . Algo más limpio seria guardar en una variable el resultado de esa pregunta. Como ves puedo capturar ese valor introducido y vamos a chequear con un condicional

var usuario = prompt("Introduce tu nombre de usuario:?"); Otra función interesante es trim(); que elimina espacios al principio o al final de una cadena de caracteres.

console.log(“ Esto es un Sting ”).trim();resultado se muestra sin espacios esto nos sirve para verificar en el if que usuario sigue siendo true una vez que paso por trim como muestra el ejemplo

if(usuario usuario && usuario.trim())

{

console.log("Usuario introducido.!!");

} else

{

console.log("NO se introdujo Usuario... ");

}

console.log(Boolean());

console.log(usuario);

La lógica seria que si solamente mostramos el prompt() y no introducimos nada retorna una cadena vacía “ ” que equivaldría a false. Lo que hacemos con el if() es preguntar si la equivalencia guleana es true o false de manera tal que podemos convertir una cadena de texto en un valor guleano. Empleando el constructot→ boolean() verificamos que es el tipo de valor o dato ha introducido. Cadena vacía→false, cadena con algún tipo de dato→true. Siguiendo con este ejemplo vamos a utilizar un confim()

var usuario = prompt("Introduce tu nombre de usuario:?");

if(usuario && usuario.trim())

{

var confirmar = confirm("Es " + usuario +" correcto?");

if (confirmar)

{

alert("Usuario correcto");

}

else

{

alert("Usuario Incorrecto");

}

}

else {

console.log("NO se introdujo Usuario... ");

}

/\*La asignación de las variable dependerá del uso que se

le darán en el desarrollo del proyecto. Se usara la convención (camellCase),

se inicia la variable con la primera letra en minuscula y la palabra siguiente sin

espacio con la primera letra en Mayúscula.

Cabe destaca que hay tres alternativas para inicializar variables

var

let

const

OPERADOR DE ASIGNACIONES

var a; var b = 2 La variable (a) esta basia en console.log dara

error undefind

la variable (b) mostrara la asignacion, La variable (a) ya esta declarada

en el proyecto, para agregarle un valor usamos el signo de asignacion =

También podemos asignar valores de una variable a otra variable.

Ej. a = 5, b = a veremos que el valor de las variables son iguale.

Otra alternativa de cómo podemos asignar el valor de una variable a otra. Declaramos

nuevamente las variables a y le asignamos un valor en este ej. vamos a cambiar

la segunda línea y declaramos la variable (b) bacía y la asignación la podemos hacer

en otra línea distinta del proyecto, es decir cuando la necesitemos.

Para incrementar una variable solamente en 1 la sintaxis más concisa seria

Variable→ a = 25; a = ++; lo mismo sucede cuando restamos 1, a --; pero si

tenemos que realizar operaciones con valores mayores que uno, en ese caso sería,

variable→ a = 23 y aplicamos la siguiente regla (variable operador asignación valor)

a += 5; a pasa a valer 28. Esto se aplica a cualquier valor, entero o decimales EJ.

var ventasDiarias = 13567.34; ventasDiarias += 345.67; ventasDiarias pasa

se incremento y pasa a valer 13913.01. Asi como podemos incrementar el valor

de una variable tambien podemos reducir ese valor, mutiplicar o dividir sigiendo

la regla nombrada → (variable operador asignación valor) Ej. de \* y /

Var salario = 45000 lo incrementamos salario \*=5 y lo reducimos a la mitad

salario /= 2;

Variables con cadenas de caracteres. Las cadenas de caracteres se definen

rodeandolas con comillas var miNombre = "Manuel"; ¿Como podemos incluir parte de

la cadena entre comillas? Eso se denomina secuencias de Escape: hay dos formas para

lograr eso, 1) se antepone una barra invertida antes de cada comilla dentro

de la cadena Ej1→ var = "Manuel \"Manu\""; Ej2→ Se inicia la cadena con comillas

simles''y el texto a destacar con comillas dobles""Ej2→ var = ´Manuel "Manu"´; Otras

secuencias de escapes muy usadas son: Mostrar la barra invertida en la cadena→ \\

o lograr un salro de linea en la cadena→ \n. Concatenar cadenas de caracteres,

para eso se usa el signo + Ej→ miNombre = "Manuel" + " Orozco"

Concatenar variables con cadena de caracteres:Ej→ var verbo = "programar";

var mensaje = "Estoy aprendiendo a " + verbo; Tambien se puede segir

concatenando la cadena despues de la variable. Ej. sigiente linea,

mensaje = "Estoy aprendiendo a " + verbo + " con freeCodeCamp";

Tambien podemos concatenar dos variables de una forma mas concisa con +=

var mensajeFinal = "El invierno llego"; Var = " con un frío muy cruel";

esto podria ser en una situacion real que esa parte del mensaje se personalisada

o que el usuario tenga que ingresar algun valor para tener la cadena completa.

Longitud de una CdC Esto es muy comun en programacion Ej→ var miCadena = "A";

Para aberiguar cuantos caracteres tiene una cadena con console.log() seria,

console.log(miCadena.length) Esta propiedad→lenght nos va a permitir saber

cuantos caracteres tiene la cadena.

Nota: Los espacios se cuentan como un caracter.

Notación de corchetes en JS, Esta es una notación que nos permite acceder

a los caracteres de una cadena. Definimos var lenguajeDeProgramacion;

lenguajeDeProgramacion = “Javascript”; Esa cadena tiene una estructura interna

que nos va a permitir acceder a sus caracteres, veamos un diagrama de EJ.

Cadena: → J A V A S C R I P T

Índices:→ 0 1 2 3 4 5 6 7 8 9

c/u de esos caracteres tiene un índice asociado y podemos usar ese índice para

acceder a ese carácter usando corchetes en console.log(variable[0]);

en este caso es 0 por que es el primer índice.

Inmutabilidad de cadenas→ no se pueden cambiar, es solo lectura.

Nota: En una cadena el último índice es longitud -1 Ej. var miCadena;

miCadena = "JavaScript" consol.log(miCadena.length -1); se muestra el numero

del ultimo índice por que no está envuelta por corchetes. Si agregamos los

corchetes se mostrara la última letra. E j→ variable[variable.length -1]

También se puede inicializar una variable para agregarla a continuación de length

Var n; n = 4 variable[variable.length variable]

Palabras en Blanco: Concatenar cadenas compuestas de caracteres y variables con

espacios en blanco. Declaramos las variables.

var miSustantivo = "perro";

var miAdgetivo = "negro";

var miVerbo = "corrió";

var miAdverbio = "rápidamente";

var miEspacio " ";

oracion = "El " + miSustantivo + miEspacio + miAdgetivo + miEspacio + miVerbo + miEspacio + miAdverbio + " a la tienda";

resultado→ El perro negro corrió rápidamente a la tienda

Siguiente tema: Arreglos (Arrays) Los arrays son estructura de datos que nos permiten agregar múltiples valores en una misma estructura. A diferencia de las cadenas de la caracteres que son Inmutabes, cabe destacar que los arrays si se pueden modificar sus índices

los array son mutables.

Para crear un arrays escribimos corchetes [] y dentro de los corchetes tendríamos los elementos de ese arrays Ej. definimos una variable var miArrays; miArrays = [“Daniel”, 24];

0: "Daniel" El nombre esta en el índice cero

1: 24 La Edad esta en el índice uno

length: 2 El largo de l arrays es de dos

Los arrays son estructuras muy poderosas para trabajar con valores relacionados

Arrays anidados: Algo muy interesante de los arrays es que pueden contener cualquier tipo de datos, caracteres, números, decimales y los podemos anidar usando una variable. Ej.

var listaDeEstudiantes;

listaDeEstudiantes = [["Nora", 18], ["Manuel", 24]];

console.log(listaDeEstudiantes);

var listaDeProductos = [["Camisas", 15.25, "s-32"], ["Zapatos", 30.45, "s-11"], ["Pantalones", 28.50, "s-33" ]];

console.log(listaDeProductos); Ahora pasamos al siguiente tema:

Acceder a los elementos de un Arrays:

Definimos una var miAcceso = [10, 20, 30];

Diagrama

Arreglos: 10, 20, 30

Índices: 0 1 2

El diagrama nos muestra que igual que en las CdC teníamos un índice

para c/carácter en el diagrama tenemos un índice para cada elemento. Acceder a los elementos de un array es parte fundamental de la programación. EJ. inicializamos una variable var suma;

suma = miAcceso[0] + miAcceso[1] + miAcceso[2] ;

console.log(suma) //resultado 60

Modificar los elementos de un array: Es otra parte importante en programación

var miModificacion; miModificacion = [10, 20, 30]; accedemos al elemento a modificar de la misma manera. Escribimos la variable, enlazamos entre corchetes con el número del índice del elemento a modificar. EJ→ miModificacion [0] seguido del signo de asignación = seguido del nuevo valor. miModificacion [0] = 40; También podemos acceder al índice [1] y modificar su elemento por una cadena de caracteres. Ej→ miModificacion [1] = “Manuel” Otra cosa que se permite en JS es tomar un elemento del arrays, digamos el índice [3] y asignarle elementos anidados. El→ miModificacion [2] = [1, 2, 3];

Acceder arrays multidimencionale:

Creamos la, var accesosArraysAnidados; para asignar un arreglo multidimensional. Ej.

accesosArraysAnidados = [[1 ,2, 3], [4, 5, 6], [7, 8. 9] ];

Para acceder a los elementos mostramos el siguiente diagrama.

Acceso : [[1 , 2, 3], [4, 5, 6], [7, 8. 9] ]; ↑ ↑ ↑ ↑

Índices: 0 1 2 ↑ ↑ ↑ ↑

Índices internos: [0 1 2] [0 1 2] [0 1 2]

La explicación grafica nos muestra como accedemos al arrays multidimensional

para eso tendremos que usar dos índices. ¿Cómo accedemos al elemento 8?

Primero usamos el índice 2 que nos permite acceder al elemento 3→[7, 8. 9]

y seguidamente el índice 1 que deja el puntero en el num. 8 para tratarlo.

El codigo seria:

var accesosArraysAnidados;

accesosArraysAnidados = [[1 ,2, 3], [4, 5, 6], [7, 8, 9] ];

// se muestra el acceso a cada elmento

console.log(accesosArraysAnidados[0]);

console.log(accesosArraysAnidados[1]);

console.log(accesosArraysAnidados[2]);

Para acceder algunos de los elementos anidados en c/u de esos array usaremos

el segundo índice que vimos en el diagrama. Digamos que queremos acceder al

elemento 6 de segundo array [4, 5, 6] accesosArraysAnidados[1] [2];

console.log(accesosArraysAnidados[1] [2]); //vemos el elemento 6

**Métodos:**

Los arrays son estructuras de datos muy poderosas , no solo podemos acceder y

modificar los elementos de un array si no que podemos agregar elementos.

Digamos que tenemos una lista de cadenas de caracteres en este caso son las

estaciones del año a la cual le falta el verano.

var estaciones = ["invierno"; "otoño"; "primavera"];

¿Cómo podemos agregar esa cadena de caracteres al arreglo? Con algo llamado método,

El método es como una función que podemos llamar para hacer algo con un elemento

**.push()** este método nos permite añadir un elemento al final del arrays

variable→ estaciones.push(“verano”) //["invierno"; "otoño"; "primavera"; "verano" ];

**.pop()** este metodo nos permite remover el último elemento de un array que nos

entrega un retorno que podemos asignarlo a una variable para su

taratamiento en el proyecto.

var estación = estaciones.pop() → ["invierno"; "otoño"; "primavera"] se extrajo “verano”

**.shift()** este metodo nos permite remover el primer elemento de un array

estaciones.shift();

["invierno"; "otoño"; "primavera";"verano"] En este array se extrae "invierno"

**Lista de compra:** (mini proyecto) de una lista de compra.

miListaDeCompras = [["Arroz" 1], ["Aceite" 2], ["Fideos" 3], ["Café" 1]];

CODIGO:

miListaDeCompras = [["Arroz", 1], ["Aceite", 2], ["Fideos", 3], ["Café", 1]];

console.log("Voy a comprar " + miListaDeCompras[0][1] + " kg de " +

miListaDeCompras[0][0] + " tambien " + miListaDeCompras[1][1] + " Litros de "

+ miListaDeCompras[1][0] + " y " + miListaDeCompras[2][1] + " paquetes de " +

miListaDeCompras[2][0] + " por ultimo " + miListaDeCompras[3][1] +

" un paquetes de 1/2 kg de " + miListaDeCompras[3][0]);

Resultado en pantalla:

Voy a comprar 1 kg de Arroz también 2 Litros de Aceite y 3 paquetes

de Fideos por ultimo 1 un paquetes de 1/2 kg de Café

**FUNCIONES:**

Con las funciones podemos escribir código que podemos reutilizar en nuestro programa

Declaración → function mostrarMensaje()

{

console.log(“Hola Manu);

}

mostrarMensaje();

mostrarMensaje();

**ARGUMENTOS:**

Lo primero que necesitamos saber para una operación, es que operando vamos a usar y que valores vamos a tener.→ Function sumar(**a, b**) lo que envuelven los paréntesis se los denominan **parámetros,** son cualquier nombre que tú le quieras asignar a un valor que se va a recibir o a tomar cuando llamemos a la función. La parte de la función que va entre las llaves se denomina cuerpo. Si vamos a sumar a+b que vamos hacer en el cuerpo de la ¿function? Vamos a tomar los valores de **a** y de **b** esos valores que en estos momentos no existen en el cuerpo de la fuction declaramos la variable suma y le asignamos los parámetros a+ b.

function sumar(a, b )

{

var suma = a + b;

console.log(“El resultado de “ + a “ + ” b + “ es ” + suma);

}

Ahora vamos a llamar a la function. Sabemos que tenemos dos parámetros, tenemos que pasar un valor para cada uno. Los valores se asignan en el mismo orden en que aparecen en la lista de parámetros en la definición de la function. Para a es 5 y para b es 3

sumar(5, 3);

Para ilustrar por que las funciones nos ayudan a escribir código que es más conciso y fácil de leer de modo tal que evita la repetición de código. Podemos llamar a esta function varias veces.

sumar(57, 3);

sumar(31, 83);

sumar(11, 43);

Estos valores que estamos pasando se denominan argumentos. Los argumentos son los valores que asignamos a los parámetros. Esos dos conceptos trabajan juntos para permitir que las function sea la herramienta poderosísima que es en programación.

Veamos otra variación que podemos tener cuando llamamos a una function En lugar de escribir los valores directamente en la lista de argumentos →(5, 3 )← también podemos asignarles a variables y pasarlas como argumentos. Ej. var x = 5; var y = 3; sumar(x, y); y el resultado sería el mismo. No solo podemos pasar números. Vamos a definir otra función para un ej. de CdC

function concatenarTresCadenas(cadena1, cadena2, cadena 3)

{

console.log(cadena1 + " " + cadena2 + " " + cadena3);

}

Luego cuando llamemos a la function y debemos pasar un valor para cada una de las cadenas de lo contrario tendríamos un error

concatenarTresCadenas("Estoy", "aprendiendo", "a programar");

Resultado en pantalla→ Estoy aprendiendo a programar

**Ámbito Global:**

Algo muy importante e JS es que no todas las variables pueden usarcé en todas las distintas partes del programa. Tenemos dos tipos de variables GLOBALES y LOCALES.

Una variable con ámbito **global** está definida dentro del programa y no dentro de la function.

Var miVariableGlobal = 5; global por que la definimos dentro del programa y no dentro de una function. Si necesitamos acceder a su valor, usarla en el programa podemos hacerlo.

console.log(miVariableGlobal);

Y también podemos usarla dentro de una function

function miFuncion()

{

console.log(miVariableGlobal);

}

Y también va a estar definida luego de la definición de la function, básicamente en cualquier lugar del programa. Para comprobar que si está definida debemos llamar a la function.

function miFuncion();

console.log(miVariableGlobal);

**AMBITO LOCAL:**

Ahora veamos cómo funcionan las variables locales. Estos tipos de variable las definimos dentro de una function que solo se pueden utilizar dentro de esa function.

Function miFunction()

{

Var miVariableLocal = 4;

console.loc(miVariableLocal );

}

Vamos a llamar a la function para ver que esa variable si está definida

miFunction();

Pero luego veamos que queremos acceder a esa variable local fuera de la function.

Esto nos dará un error de no definición de la variable

Si declaras una variable dentro de una function solo se podrá usar solo dentro del cuerpo de la function.

**Ámbito Local vs Ámbito Global:**

Veamos que sucede cuando tenemos una v/local y una v /global con el mismo nombre.

var miNombre = "Nora"

function mostrarMiNombre()

{

var miNombre = "Ricardo";

console.log(miNombre);

}

Cuando intentamos verificar la variable llamando la function, vemos que predomina la

v/local sobre la v/global

mostrarMiNombre(); //resultado Ricardo

Pero si tratamos de mostrar la variable global en el programa, lo podemos hacer y la podemos usar en cualquier lugar del proyecto.

console.log(miNombre);

**RETORNAR UN VALOR:**

Las function pueden retornar un valor además de recibir valores con propiedades especiales para poder interactuar con el programa principal.

function sumar(a, b)

{

return a + b; // no lleva coma

}

//si nosotros llamamos a la function

sumar(5, 3); //no se muestra nada, para eso usamos

console.log(sumar(5, 3)); //resultado retorna 8

**UNDEFINED:**

Así cómo podemos retornar un valor específico en una function con returna, también podemos omitir esa sentencia pero aun así la function va a retornar un valor por defecto ese valor va a ser UNDEFINED. Si definimos una function que en vez de retornar el lugar de la suma lo vamos a mostrar en la consola

function sumar(a, b)

{

Console.log(a + b);

//return a + b; // no lleva coma

}

Como no estamos retornando ningún valor explícitamente

console.log(5, +3); // vemos aquí UNDEFINED

**ASIGNAR UN VALOR RETORNANDO**

Cuando una fuction retorna un valor podemos asignarle ese valor a una variable

function sumar(a, b)

{

return a + b; // no lleva coma

}

var resultado = Sumar(5, 3); //resultado ahora vale 8

console.log(resultado); //se muestra 8

Otro ejemplo, podemos definir una function con una cadena de caracteres con nuestra meta.

¿Cuál va a ser nuestra meta? Vamos a retornar una cadena de caracteres

function crearCadenaConMeta(lenguajeDeProgramacion)

{

return "Mi meta es aprender " + lenguajeDeProgramacion;

}

Si llamamos a la function y pasamos la cadena de caracteres “JavaScript” el parámetro va a tener ese valor cuando llamemos a la function y lo asignamos a una variable para su tratamiento

var miMeta = crearCadenaConMeta("JavaScript");

console.log(miMeta); //resultado: Mi meta es aprender JavaScript

**Permanece en Fila**

Vamos hacer una variación de una cola (queu) es una estructura de datos que nos va a permitir agregar elementos a una fila. Vamos a tener una estructura de datos abstracta, en la cual los elementos se mantienen en orden.Los nuevos elementos se pueden añadir al final de la cola y los elementos previos se retiran al principio de la cola. Vamos a definir una function que se llame próximo en la fila, esa function debe tomar un arreglo, que se va a llamar arreglo←ese va a ser el nombre del parámetro y va a tomar un número que se llamara →elemento ese va a ser el parámetro, ambos van a ser los argumentos. La tarea es agregar el numero al final del argumento →arreglo. Y luego la function debe eliminar l primer elemento del arreglo. Es un ciclo vamos agregarlo al final y si se agrega un elemento significa que el primero debe ser eliminado. La function proximoEnLaFila(){ } debe retornar el elemento que fue removido.

¿Qué es lo primero que queremos hacer cuando llamamos a la function? Agregar el número al parámetro elemento al final del array con el método .push . Luego que agregamos al parámetro del elemento en el array. Debemos eliminar el primer elemento con el método .shitf() no toma ningún argumento los paréntesis vacíos. Y la parte final de lo que debe cumplir la function, es que debe retornar el elemento que fue removido. Recordemos que el método .shift también retorna ese elemento es ahí donde anteponemos return

function proximoEnLaFila(arreglo, elemento)

{

arreglo.push(elemento);//agregar al final del array

return arreglo.shift() ; //shift es el que va a remover el primer elemento del array

}

Definimos una variable

var miArreglo = [1, 2, 3, 4, 5,]

Queremos ver el estatus de nuestro arreglo antes de llamar a la function. Nota:JSON.stringify()

es una funcion muy util que nos muestra los array en la consola en un formato especifico

console.log("Antes: " + JSON.stringify(miArreglo));

Llamamos a la function, y el primer elemento que toma es el arreglo y ¿cuál argumento

pasamos? La variable miArreglo como argumento → se cambia a [1, 2, 3, 4, 5,] y luego

que tenemos ese arreglo queremos agregar un elemento que será → 6 El efecto de llamar

a esta function es que se elimina el primer elemento →1 y se agrega el último elemento →6

[2, 3, 4, 5, 6]

console.log(proximoEnLaFila(miArreglo, 6));// aquí realizamos el cambio

console.log(proximoEnLaFila(miArreglo, 6));// aquí realizamos el cambio

Luego de esto mostramos después de hacer el cambio

console.log("Después: " + JSON.stringify(miArreglo));

**VALORES BOOLEANOS**

Verdadero Falso

console.log(true) console.log(false)

**Operadores de igualdad**

Esto que tenemos entre parentesis se denomina una exprecion, va a evaluar

true o false

console.log(5 == 5) true / consol.loge(6 == 5) false

De igual forma podemos comparar cadenas de caracteres la comparacion tambien

se hace en base de mayusculas o minusculas

console.log("Hola" == "Hola") true / console.log("Hola" == "hola")false

Es importante saber que no se deve comparar array con este operador

console.log([1,2,3]==[1,2,3]); El resultado va a ser false, ¿por qué? porque

no compara los elemento de los array sino que compara si los arreglos en la memoria

representan el mismo objeto.

**Operador de igualdad estricta:**

Este operador nos permite comparar si ambos tipos de datos son los mismos.

Veamos la diferencia ente igualdad = = e igualdad estricta. = = =

console.log(9 = = 9); // Resultado true

Pero si tenemos dos valores que son de distinto tipos de datos pero que representan el

mismo número el resultado también va a ser true. ¿Qué ocurre con este operador?

Antes de realizar la comparación ambos valores se convierten a un tipo de dato común

Por eso es que el resultado nos dice true

console.log(9 = = “9”); // Resultado true

Pero si queremos que la comparación también determine que el tipo de dato es

el mismo o no, es ahí donde usamos el operador de igualdad estricta

console.log(9 = = = “9”); // Resultado es false

**Operador de desigualdad**

Este operador compara dos valores y retorna true si su valor es distinto y retorna

false si son iguales

console.log(9 != 6)//Resultado true

console.log(9 != 9)//Resultado false

De igual manera sucede con cadena de caracteres, pero en el caso de los array

Podemos intentar comparar array con este operador

console.log([1,2,3] != [1,2,3]);//Resultado es true

pero no los va a comparar en base a sus elementos, sino en base a como están

representados como objetos en la memoria del dispositivo que es algo relacionado a como se almacenan en la memoria

**Operador de desigualdad estricta:**

En el tema anterior con el operador de desigualdad si nosotros comparábamos si los siguientes valores (1 != “1”) eran distinto el resultado sería **false** porque **!=** ←este operador convierte ambos valores a un tipo de datos común antes de hacer la comparación y detecta que ambos números son iguales. Pero que ocurre con el operador != = ←de desigualdad estricta (1 != = “1”); ahora el resultado sería **true** porque también está comparando si ambos valores son del mismo tipo de datos, no lo convierte a un tipo de dato común.

console.log(1 != "1");// true está mal

console.log(1 !== "1");// false está bien

**Operador mayor que:**

En javaScipt además de comparar valore si son iguales o diferentes también podemos comparar el valor relativo de los valores. Si un valor es mayor o menor que el otro o menor igual o mayor igual.

Operador mayor que > Digamos que queremos saber si (6 > 5); ←true / (3 > 10); ←false

También podemos comparar cadena de caracteres. El criterio con que JavaScript compara las cadenas de caracteres es según el orden alfabético. console.log("B" > "A"); true ←Estamos diciendo que si la letra “B” es mayor que la letra “A” según el orden alfabético como estamos usando este operador mayor que > Cuando aplicamos a cadena actúa como si estuviera más allá adentro del diccionario. Si tuviéramos otra cadena console.log("A,C,B" > "A","B","C"); //true ←

Como javascript compara esta cadena. Compara cada uno de los caracteres en su secuencia.

A =A / C=B compara **C** viene después que **B** sí. Ahí se determina el orden de la cadena porque el primer par de caracteres desiguale que encuentra y compara define el resultado, de manera tal que es true. Si una cadena tiene más caracteres que otra verificamos→ ("AB" > "A");true← porque tiene más caracteres. Por orden alfabético también podemos comparar palabras ("Mundo" > "Hola");true← Por orden alfabético **M** viene después que la letra **H** en el alfabeto asi a modo de ejemplo comparamos ("M" > "H");true← Otro dato interesante es que podemos usar estos datos de comparación con variables. var az = 15; / var by = 8;

console.log(az > by);//true / console.log(by > az);//false

**Mayor o igual que:**

Este operador >= nos permite incluir el caso en que ambos valores sea iguales,

(5 > 5); //false (5 >= 5); //true En el caso de cadenas de caracteres también aplica

cuando las cadena sean iguales el valor va a ser true

**Menor que:** Este operador < nos permite verificar (5 < 6); //true, si el número a la

izquierda es mayor (10 < 8);//false. También se puede usar para cadena de caracteres

siempre la comparación de cadena será por orden alfabético ("A" < "B");true porque

A esta antes que B en el alfabeto. En este ejemplo ("A ,C, B" > "A","B","C");//true←

el proceso va a ser igual que en ejemplos anteriores. También podemos comparar los valores

de variables

Operador menor o igual que: basicamente funciona igual que menor que <

(5 < 5); //false / (5 >= 5); //true.

**Operadores lógicos "and"**

Estos operadores lógicos nos permiten expresar distintas expresiones para formar

condiciones un poco más elaboradas o más complejas

Tabla de verdad de operador AND←→&&

| x | z | x && tenemos dos expresiones x y z y los unimos x && z

↓ ↓ ↓

-----------------

↓ ↓ ↓

| T | T | T |Si x es true y z es true toda esa expresión es true

| T | F | F |En cambio si alguno de ellos es false o ambos

| F | T | F |entonces la expresión va a ser

| F | F | F |←false

La operación solo es verdadera cuando ambos operadores son verdaderos.

Operadores lógicos "and" Estos operadores lógicos nos permiten expresar distintas

expresiones para formar condiciones un poco más elaboradas o más complejas

Tabla de verdad de operador Y→AND←→&&←en JS se expresa con &&

| x | z | x && Z |tenemos dos expresiones x y z y los unimos x && z

| ↓ ↓ ↓ |

├----------------┤

| ↓ ↓ ↓ |

| T | T | T |Si x es true y z es true toda esa expresión es true

| T | F | F |En cambio si alguno de ellos es false o ambos

| F | T | F |entonces la expresión va a ser

| F | F | F |←false

La operación solo es verdadera cuando ambos operadores son verdaderos.

**Operador logico "or"**

Tabla de verdad del operador O→OR←→||←en JS se expresa con ||

| x | z | x && Z |tenemos dos expresiones x y z y los unimos x || z

| ↓ ↓ ↓ |

├----------------┤

| ↓ ↓ ↓ |En el operador OR si cualquiera de los operando es true entonces

| T | T | T |el resultado es verdadero

| T | F | T |←true

| F | T | T |←true

| F | F | F |←false

La operación solo es verdadera si alguno de los dos operando o ambos son true

**Operador logico "not"**

Tabla de verdad del operador NO→NOT←→!←en JS se expresa con !

Para: !x

| x | !x|

| ↓ ↓ |

├-------┤

| ↓ ↓ |

| T | F | Si la expresión es true agregando el operador not la convertimos en false

| F | T |Si la expresión es false agregando el operador not la convertimos en true

**Setencias Condicionales:**

Las condiciónale determinan que rumbo va a continuar nuestro código, en el caso de (if) o si en español siguiendo el ejemplo seria if(condición, la cual si es verdadera){ejecutara lo que esta ente las llaves} o cuerpo de la condicional. También se pueden usar los operadores lógicos→ && || !← para condiciones más elaboradas.

var cond = 5

if (cond > 2) // la condición se cumple

{

console.log("La condición es verdadera"); // ejecuta el código

}

if((cond > 2) && (cond < 10)); //condiciones más complejas

{

console.log("La condición es verdadera"); // ejecuta el codigo

}

var estación = "invierno"

if(estación == "invierno") // la condición se cumple

{

console.log("¡Si! Me encanta el invierno");// ejecuta el código

}

if (cond < 2) // la condición NO se cumple

{

console.log("La condición es verdadera"); // NO ejecuta el código

}

console.log("La condición no es verdadera"); // Continua la ejecución saltando el if

**Clausula “else”**

También podemos expandir nuestras condicionales si la condición de la condicional es false if (false){

console.log(“si la condición es **true** se ejecuta este código”);

} else{

console.log(“si la condición es false ignora el **if** y se ejecuta **else** con este código”);

}

**Clausula else if:**

Los condicionales también pueden verificar y manejar varias condiciones, con la cláusula else if que nos permite manejar exactamente eso, manejar condiciones alternativa y decidir por la primera condición que encuentre y sea verdadera→ true. Para el ejemplo es conveniente trabajar con funciones: Esta función va a tomar un

valor como un parámetro y va a determinar si el valor es divisible entre 2 o 3

o si no es divisible entre ninguno.

function clasificarValor(valor)

{

if(valor % 2 == 0)//genera un resto igual a cero es true

{

console.log("Divisible entre 2");

}

else if(valor % 3 == 0)

{

console.log("Divisible entre 3");

}

else

{

console.log("NO es divisible entre las opciones");

}

}

clasificarValor(2) //caso 1 se ejecuta porque es el primer true que encuentra

clasificarValor(15)//caso 2 se ejecuta porque caso 1 es false

clasificarValor(7) //caso 3 se ejecuta porque caso 1 y 2 son false

var elcasoUno = "caso 1 se ejecuta porque es el primer true que encuentra"

var elCasoDos = "caso 2 se ejecuta porque caso 1 es false"

var elCasoTres = "caso 3 se ejecuta porque caso 1 y 2 son false"

console.log(elcasoUno);

console.log(elCasoDos);

console.log(elCasoTres);

**Condicionales Orden lógico,** **se llama a la function y se ejecuta según el caso**

clasificarValor(2) //caso 1 se ejecuta por que es el primer true que encuentra

clasificarValor(15)//caso 2 se ejecuta por que caso 1 es false

clasificarValor(7) //caso 3 se ejecuta por que caso 1 y 2 son false

var elcasoUno = "caso 1 se ejecuta por que es el primer true que encuentra"

var elCasoDos = "caso 2 se ejecuta por que caso 1 es false"

var elCasoTres = "caso 3 se ejecuta por que caso 1 y 2 son false"

console.log(elcasoUno);

console.log(elCasoDos);

console.log(elCasoTres);

**Encadenar sentencias "if else"**

Vamos a ver un ejemplo con dos clausulas else if creando una funcion

function interpretarIMC(indiceDeMasaCorporal)

{

if(indiceDeMasaCorporal < 18.5)

{

console.log("Bajo peso");

}

else if(indiceDeMasaCorporal <= 24.9 )

{

console.log("peso Normal");

}

else if(indiceDeMasaCorporal <= 29.9)

{

console.log("Sobrepeso");

}

else

{

console.log("Obeso");

}

}

interpretarIMC(17.8);//caso 1 se ejecuta porque es el primer true que encuentra

interpretarIMC(22.2);//caso 2 se ejecuta porque caso 1 es false

interpretarIMC(28.5);//caso 3 se ejecuta porque caso 1 caso 2 es false

interpretarIMC(32.2);//caso 4 se ejecuta porque no encuentra ningún true

var casoIF = "Caso 1 se ejecuta porque es el primer true que encuentra"

var casoPrimerElseIf = "Caso 2 se ejecuta porque caso 1 es false"

var casoSegundoElseIf = "Caso 3 se ejecuta porque caso 1 caso 2 es false"

var casoElse = "Caso 4 se ejecuta porque no encuentra ningún true"

console.log(casiIF);

console.log(casoPrimerElseIf);

console.log(casoSegundoElseIf);

console.log(casoElse)

**Código de golf**

Definimos una función que va a llevar como parámetro→par que es el numero

promedio de goles esperados y→golpes que es el número de golpes ejecutados.

para proseguir con este mini proyecto deberemos usar condicionales para implementar

esta tabla.

Golpes Retornar

---------------------------------

1 "Hole-in-one!"

<= par - 2 "Eagle"

par - 1 "Birdie"

par "Par"

par + 1 "Bogey"

par + 2 "Doble Bogey"

>= par + 3 "Go Home!"

Nota: par y golpes siempre serán numéricos y positivos, el parámetro par es→4

function PuntageDeGolf(par, golpes)

{

if(golpes == 1)

{

return "Hole-in-one!";

}

else if(golpes <= par - 2)

{

return "Eagle";

}

else if(golpes == par - 1)

{

return "Birdie";

}

else if(golpes == par)

{

return "Par";

}

else if(golpes == par + 1)

{

return "Bogey";

}

else if(golpes == par + 2)

{

return "Double Bogey";

}

else if(>= par + 3)

{

return "Go Home!";

}

}

console.log(puntajeDeGolf(4, 1)); // par 4 retorna 1 se muestra "Hole-in-one!"

console.log(puntajeDeGolf(4, 2)); //par 4 <= par -2 retorna 2 se muestra "Eagle"

console.log(puntajeDeGolf(4, 3)); // par 4 par -1 retorna 3 se muestra "Birdie"

console.log(puntajeDeGolf(4, 4)); // par 4 par retorna 4 se muestra "Par"

console.log(puntajeDeGolf(4, 5)); //par 4 par + 1 retorna 5 se muestra "Bogey"

console.log(puntajeDeGolf(4, 6)); // par 4 par +2 retorna 6 se muestra "Doble Bogey"

console.log(puntajeDeGolf(4, 8)); // par 4 >= par + 3 retorna 7 se muestra "Go Home!"

**Sentencia Switch:**

Ahora vamos a ver otra estructura que en cierta forma remplaza los condicionales en JS

Definimos una función con un parámetro →valor, este parámetro va a ser un numero entre

1 y 4 cada uno de esos valores va a estar asociados a una cadena específica. A una cadena de caracteres, la función va a retornar esa clasificación, esa cadena a partir del número en lugar de condicionales vamos a usar la sentencia switch que nos permite escribir código que es muy similar a una condicional. Pero swifch nos permite a partir de un valor decidir que va a pasar en base a ese valor.

function clasificarValor(valor)

{

var respuesta;//variable local

switch (valor)

{

case 1:

respuesta = "alpha";

break;

case 2:

respuesta = "beta";

break;

case 3:

respuesta = "gamma";

break;

case 4:

respuesta = "delta";

break;

}

return respuesta;

}

Y finalmente vamos a retornar el valor respuesta. return se coloca fuera de las llaves porque no es parte del switch . Finalmente llamamos a la función para pasarle los valore que podemos asignarle a una variable o para mostrarlo en pantalla

console.log(clasificarValor(1));//valor=1 se ejecuta caso1 respuesta vale→"alpha"

console.log(clasificarValor(2));//valor=2 se ejecuta caso2 respuesta vale→"beta"

console.log(clasificarValor(3));//valor=3 se ejecuta caso3 respuesta vale→"gamma"

console.log(clasificarValor(4));//valor=4 se ejecuta caso4 respuesta vale→"delta"

Ahora vamos a usar switch con una cadena de caracteres solo declarando una variable

var producto = "helado"

switch (producto)

{

case "pizza":

console.log("La pizza basica cuesta $550");

break;

case "hambueguesa":

console.log("Las hamburguesas cuestan $250");

break;

case "helado":

console.log("El helado cuesta $95");

break;

}

//Como la variable valor se asignó "helado" se muestra→ ("El helado cuesta $95")

}

console.log("Luego del switch continua ejecutando el código");

**Sentencia Switch opción predeterminada**

También podemos agregar una opción predeterminada a la sentencia switch que es el equivalente a else en una condicional if. Esa opción se va a ejecutar si ningunos de los valores es el valor de la variable o de la expresión. Esta función va a seleccionar el idioma que corresponda a un valor especifico.

function seleccionarIdioma(valor)

{

var idioma;

switch(valor)//dentro de la sentencia agregamos el parametro

{

case 1:

idioma = "Español";

break;

case 2:

idioma = "Frances";

break;

case 3:

idioma = "Italiano";

break;

default:

idioma = "Ingles";

break;// es opcional pero por norma lo usamos

}

return idioma;

}

console.log(seleccionarIdioma(1));//valor=1 se ejecuta caso1 var idioma es "Español"

console.log(seleccionarIdioma(2));//valor=1 se ejecuta caso1 var idioma es "Frances"

console.log(seleccionarIdioma(3));//valor=1 se ejecuta caso1 var idioma es "Italiano"

console.log(seleccionarIdioma(4));//se ejecuta por defecto var es "Ingles"

**Sentencia switch múltiples casos:**

En las sentencias switch también podemos optar ejecutar un código específico para varios valores vamos a tener números entre 1 y 6 que van a representar los valores en el dispositivo. El parámetro se va a llamar valor, el parámetro que va a representar el volumen que va ser una cadena de caracteres que va a definir la intensidad del volumen. Para eso usaremos la sentencia switch , esa sentencia va a depender del valor del parámetro. Pero si queremos que el volumen intermedio sea el caso 2 o el 3 lo podemos hacer dentro de un solo break dos valores pueden realizar la misma acción. También podemos tener más de dos casos manejado de la misma forma

caso 4, 5 y 6 y si es valor esta fuera de rango, digamos 7 se ejecuta por defecto

function clacificarVolumen(valor)

{

var volumen;

switch(valor)

{

case 1:

volumen = "Bajo";

break;

case 2:

case 3:

volumen = "Intermedio";

break;

case 4:

case 5:

case 6:

volumen = "Alto";

break;

default:

volumen = "Sin volumen";

break;

}

return volumen;

}

console.log(clacificarVolumen(1));//valor=1 se ejecuta caso1 var volumen "Bajo"

console.log(clacificarVolumen(2));//valor=2 se ejecuta caso2,3 var volumen "Intermedio"

console.log(clacificarVolumen(4));//valor=4 se ejecuta caso4,5,6 var volumen "Alto"

console.log(clacificarVolumen(7));//valor=7 se ejecuta default var volumen "Sin volumen"

**Reemplazar “if… else” por “switch”**

En ciertos casos podemos remplazar condicionales con la sentencia switch en el ejemplo vemos que el nombre de la función y el parámetro al igual que la variable no cambia al igual que en el final el retorno el mismo

function seleccionarIdioma(valor)

{

var idioma;

switch(valor) if(valor == 1

{ {

case 1: idioma = “Español”;

idioma = "Español"; } else if (valor == 2) {

break; idioma = “Frances”;

case 2: } else if(valor == 3) {

idioma = "Frances"; idioma = “Italiano”;

break; } else {

case 3: idioma = “ingles”;

idioma = "Italiano"; }

break;

default:

idioma = "Ingles";

break;// es opcional pero por norma lo usamos

}

return idioma;

}

**Retornar valores Vooleanos:**

Como retornar valores booleano de una forma muy concisa desde una función, inicialmente lo podíamos iniciar con un condicional que funciona y va a cumplir su propósito pero lo mismo lo podemos hacer en una sola línea simplemente retornar el valor de una comparación.

→return a < b; ←nota la diferencia de lo conciso que es en comparación de usar condicional

function esMenorQue(a, b)

{

If(a < b)

{

return true;

}

Else

{

return false;

}

}

**Patrón de retorno anticipado**

Cuando retornamos el valor de una función, en ese momento que retornamos el valor la función se detiene completamente. Cualquier línea que esta luego del return no se va a ejecutar. En el ejemplo vamos a intentar mostrar otro mensaje después de return→Esa línea nunca se va a ejecutar y la función se detiene en el return. Eso es algo importante porque ese patrón anticipado es muy útil, específicamente cuando necesitamos detener la función.

function miFuncion()

{

console.log("Hola")

return "Mundo";

console.log("Adios");//unreachable code detecte (7027) ←codigo inalcansable detectado

}

console.log(miFuncion());// aquí nos muestra "hola" "Mundo"

**Patron de valor anticipado**

A continuación en este ejemplo definimos una función y le damos un parámetro→ num, La raíz cuadrada de un número negativo no existe, así que si esa condición es verdadera num<0 no podemos calcular la raíz cuadrada. En ese caso lo que hacemos es detener la función y retornamos un valor que sería→ udefefined En el caso de que el valor sea valido→ num>0. Digamos que pasamos 25, la condición del if no se cumple la raíz cuadrada seria 5 pero si pasamos un numero negativo la condición if es true y se ejecuta retrun undefined

function raizCuadrada(num)

{

if(num < 0)//num no es menor que 0 y el resultado es 5 pero si pasamos -5 ahi entra al if

// y se ejecuta return undefined

{

return undefined//aqui se detiene la funcion si la condicion es true

}

return Math.sqrt(num);//el parametro num calculamos la raiz cuadrada

}

console.log(raizCuadrada(25));//pasamos a num el valor 25 que nos devueve su raiz 5

console.log(raizCuadrada(-5))//pasamos a num el valor -5 que nos devueve undefined

**Juego de Cartas:** La meta es definir una función para contar cartas

Lo primero que definimos es la variable global luego la función con un parametro→carta, en la función declaramos una variable local decisión, esa va a ser la decisión final. ¡Si apostar o esperar!

pero primero tenemos que actualizar el valor del conteo en la sentencia switch, esa sentencia va a evaluar el valor del parametro→carta y dependiendo de su valor actualizamos el valor del conteo. Ahi podemos ver en la tabala que si el valor es 2, 3, 4, 5, 6 tenemos que aumentar el valor de la variable→conteo en uno con un contador ++ y luego break. Para la segunda categoría de valores 7, 8, 9 no tenemos que cambiar el valor del conteo por eso no lo incluimos en la sentenci switch. Si miramos la tabla veremos que si tenemos que incluir los casos→10,"J","Q","K","A" en cualquiera de estos caso tenemos que reducir el valor en uno con un contador -- y break que nos indica fin de casos. Fuera de switch creamos un condicional if si la variable conteo > 0→true la variable local decision será "Apostar" si no sera "Esperar" dependiendo del condicional sera el retorno que obtenemos.

Tabla

Cambio de conteo carta

+1 2, 3, 4, 5, 6

0 7, 8, 9

-1 10, “J” “Q” “K” “A”

var conteo = 0;

Function contarCartas(carta)

{

var decision;

switch()

{

case2:

case3:

case4:

case5:

case6:

conteo ++;

break;

case10:

case”J”:

case”Q”:

case”K”:

case”A”:

conteo--;

brek;

}

If(conteo > 0)

{

Decision = “Apostar”

}

Else

{

Decision = “Esperar”

}

return conteo + “ “ + “Decisión ”

}

**Crear Objetos:**

Los objetos nos permiten guardar la secuencia o un conjunto de propiedades que están relacionadas con sus correspondientes valores. En el ejemplo digamos que queremos representar las propiedades de nuestro perro.

Secuencia que en verdad representa el objeto

//Se usa connotación de punto

var miPerro = {

"nombre": "camilo",

"edad": 3,

"peso": 6,

"rasa": "sharpei"

};

Esta combinación de propiedades→a la izquierda y de valores→ a la derecha, lo denominamos un objeto. Digamos que es como una forma de agruparlas todas bajo una misma estructura de datos, que podemos usar y acceder a través de la variable→miPerro. Nota: Las propiedades y los valores no necesaria mente siendo una sola palabra se usan la comilla s. Propiedades de los valores se separan con→ **:** y al final de la línea va una→ , También se pueden usar números como propiedades→5: “cinco”,

**Acceder a propiedades notación de puntos:**

Si necesitamos acceder al objeto que es camilo, podemos acceder a ese valor con la notación de punto. Escribimos el nombre de la variable seguido de un punto escribimos el nombre de la propiedad Ej. console.log(**miPerro.edad**); se muestra **3**

**Acceder a propiedades notación de corchetes:**

Una notación alternativa que podemos usar para acceder a las propiedades de un objeto, en el siguiente código como podemos notar que hay propiedades que son cadenas de caracteres con espacios, en ese caso si o si van entre comillas y debemos usar la notación de corchetes que es equivalente con la notación de punto. Esto quiere decir que podemos acceder a cualquiera de las propiedades del objeto

//Se usa notación de corchetes

var miCuaderno = {

"color": "verde"

"categoria": 3,

"numero de paginas": 200,

"numero de hojas": 100

};

**Acceder a propiedades Variables**

Como acceder a una propiedad de un objeto usando una variable. Vamos a escribir el valor entre corchetes usando una variable, para poder acceder a un lugar específico de esos valores definimos la variable→ posición. Como podemos acceder: Escribimos el nombre de la variable que contiene a ese objeto→resultado y entre corchete pasamos en vez de el nombre de la propiedad pasamos el nombre de la variable→posicion esto es equivalente a escribir 4 pero como estamos escribiendo una variable podemos trabajar con esa variable en nuestro programa haciendo que el acceso sea más dinámico. Para mostrar ese resultado llamamos a consol.log usando la notación de corchetes

var resultados = {

1: "Manuel5711",

2: "Nora3578",

3: "Lucas2115",

4: "Cecilia4598"

}

var posicion = (4);

console.log(resultados[posicion]);

**Actualizar Propiedades:**

Además de crear un objeto en el proyecto y acceder a sus propiedades, también puedes actualizar

el valor de esas propiedades para permitir que tu objeto cambie durante la ejecución del

programa. Vamos a ver un ej. que llamaremos mochila. Con console.log escribimos

la variable mochila separada por un punto→ color resultado azul

var mochila = {

"valor": "azul",

"tamaño":"mediano",

"contenido" :["botella de agua", "y cuaderno"]

};

console.log(mochila.color);// resultado azul

Una vez que ya tenemos estas propiedades definidas, digamos que queremos cambiar el color de

la mochila vamos a acceder a ellas y la actualizaremos. Para acceder a la propiedad color

Lo que hacemos es escribir el nombre de la variable que contiene el objeto seguido de un

punto, el nombre de la propiedad que queremos actualizar y luego le asignamos el

valor→"verde"

Ej.

mochila.color = "verde";//reasignamos el valor

También podemos trabajar con sus valores para modificarlos. Si queremos agregar algo a la mochila, tendríamos que acceder al valor de la propiedad contenido. Para actualizar este valor podemos usar notación de punto→**mochila.contenido.** consolle.log(mochila.contenido);

// vemos ["botella de agua", "y cuaderno"]

Digamos que queremos agregar un lápiz a nuestra mochila→ mochila.contenido.push(); llamamos al método push que agrega un elemento al final del arreglo

mochila.contenido.push("lapiz");//agregamos lápiz

console.log(mochila.contenido);//vemos ["botella de agua", "cuaderno,", "lapiz"]

Nota: pop() este metodo nos permite remover el primer elemento de un array

**Agregar Propiedades:**

También puedes agregar propiedades a un objeto en JS. Vamos a tener un objeto que va a representar un curso

var curso = {

"titulo": "Aprende JavaScript desde cero",

"idioma": "Español",

"duracion": 30;

Como podemos agregar una cuarta propiedad? La propiedad se va a llamar vista, va ser el número de vista desde que se publicó el curso. La sintaxis es la misma que utilizamos para cambiar una propiedad pero en este caso vamos a escribir el nombre de la propiedad que queremos agregar. Primero el nombre de la variable seguida de un punto, el nombre de la propiedad nueva y luego el valor que queremos asignar.

curso.vista = 34500

console.log(curso.vista);//resultado 34500

Nota: Si intentamos acceder a una propiedad que no existe el resultado será **undefine**

También podemos en vez de usar notación de punto, podemos usar notación de corchetes.

console.log([“vista”]); //resultado 34500

**Eliminar Propiedades:**

Continuando como ejemplo el objeto curso. Ahora vamos a ver cómo puedes eliminar una

propiedad de este objeto. La palabra clave aquí es→ DELETE que significa sacar o eliminar

en español. Digamos que ya no queremos tener un registro de la propiedad duración en el

objeto curso, tenemos que escribir anteponiendo delete seguido de un espacio, el nombre de la variable separando esta con un punto y la propiedad que queremos eliminar.

La sintaxis seria: delete curso.duracion;

console.log(curso.duracion);//vista antes de borrar 30

delete curso.duracion;// nombre del obj un punto y la propiedad a eliminar

console.log(curso.duracion);//despues de borrar→ undefined

Si cambiamos la línea de código anterior por una línea de código que nos muestre el obj completo

console.log(curso);/\* este es el obj que se nos presenta:

{"titulo": "Aprende JavaScript desde cero", "idioma": "Español"} \*/

**Objetos para Búsquedas:**

Según el caso podemos usar un objeto para remplazar una sentencia switch

Tenemos una función (buscar elemento químico). Si analizamos el elemento switch podemos ver que a partir del valor de **símbolo** vamos a asociar ese símbolo con su nombre completo→"Aluminio" Es decir que **"Al"** estaría asociado **"Aluminio"** y así será hasta finalizar la sentencia. Cuando tienes una sentencia switch con esta estructura que asocia dos valores podemos escribir código más conciso con un **objeto** recuerda que los obj nos permiten asociar una **propiedad** con su **valor.**

El código que a continuación se presenta usando switch perfectamente puede ser remplazado por un objeto

Function buscarElementoQimico(simbolo)

{

Var simbolosQuimicos = "";

switch(simblo)

{

case "Al":

elementoQuimico = "Aluminio";

break;

case "S":

elementoQuimico = "Azufre";

break;

case "Cl":

elementoQuimico = "Cloro";

break;

case "He":

elementoQuimico = "Helio";

break;

case "B":

elementoQuimico = "Boro";

break;

case "Li":

elementoQuimico = "Litio";

break;

}

return elementoQuimico;

}

En el siguiente ej. veremos cómo podemos remplazar switch por un objeto:

Function buscarElementoQimico(simbolo)

{

simbolosQuimicos = {

"Al" : "Aluminio",

"S" : "Azufre",

"Cl" : "Cloro",

"Hd" : "Helio",

"B" : "Boro",

"Li" : "Litio";

};

Return simboloQuimico[simbolo]esto retorna el nombre completo del elemento símbolo

};

Ya tenemos el objeto al cual hemos restructurando nuestro código de una forma específica y que realiza exactamente lo mismo que switch pero de forma más concisa. Podemos acceder a las propiedades del obj con notación de corchetes y pasar el valor del parametro →simbolo a la variable→simbolosQuimicos esto nos va a retornar el valor de correspondiente a ese simblo.

Ahora veamos si retorna el valor apropiado

console.log(buscarElementoQimico("Al"));

console.log(buscarElementoQimico("S"));

console.log(buscarElementoQimico("Cl"));

console.log(buscarElementoQimico("Hd"));

console.log(buscarElementoQimico("B"));

console.log(buscarElementoQimico("Li"));

**Verificar Propiedades:**

Otra operación muy importante es verificar si el obj tiene una propiedad o no, si tratamos de acceder a una propiedad que no existe en el obj el resultado será undefined. Podemos verificar si la propiedad existe o no antes de usarla. Para averiguar si el obj tiene una propiedad especifica escribimos el nombre de la variable que contiene el obj→miCofre y llamamos al método→ hasOwnProperty() Este método es muy útil si estamos trabajando con un condicional.

var miCofre = {

"oculto": "cajon",

"color": "verde",

"guardo": "recuerdos"

};

miCofre.hasOwnProperty("color");//pegunta si la propiedad existe

Para ilustrar como este método que es muy útil en un condicional vamos a definir una función que va a verificar si un obj tiene una propiedad o no y va a mostrar un mensaje, La funcion va a tomar dos parámetros. En el if retorna una cadena, que va ser la unión de una cadena”propiedad “ con el valor de la propiedad en ese objeto

function verificarPropiedad(obj, propiedad)//propiedad del obj function verificarPropiedad(obj, propiedad)//propiedad del obj

{

if(obj.hasOwnProperty("propiedad"))// es true por que el obj tiene esa propiedad

{

return "propiedad " + obj[propiedad]; //propiedad es el parámetro “color”

}

else

{

return "El objeto no tiene esta propiedad"//la propieda que pasamos no existe false

}

console.log(verificarPropiedad(miCofre, color))

**Objetos Complejos:**

Estos tipos de objetos tienen combinaciones de distintos tipos de datos. Como sus valores, los valores de las propiedades, puedes ver que la variable→ ordenesDePizzas contiene un array[]. Esto nos dice que es un array de objetos porque están las llaves que abren y sierran el objeto y dentro de cada objeto que tenemos aquí, en nuestro array tenemos secuencias de propiedades y sus valore que siguen un formato muy similar, las mismas propiedades asociadas a distintos valores. Estos es un formato que normalmente se usa muy similar al formato JSON→

(JavaScript, Object, Notacion). Primero que nada debemos entender que inicialmente estamos trabajando con un array, asique cada uno de esos objetos tienen un lugar en ese array, como si fuera un elemento, como si tuviéramos dos elementos con una cadena de caracteres, o dos números. Pero en este caso tenemos dos objetos que c/u de ellos tiene su conjunto de propiedades y los valores de esas propiedades. Veamos su estructura: El primer obj representa una pizza **margarita**. Pero en el segundo obj representa una pizza de **cuatro quesos**. Ambos obj tienen la misma propiedad pero sus valores son distintos. A continuación en el primer obj tenemos el **tamaño** de la pizza **individual** y en el segundo obj es **familiar.** Retornando al primer obj sigue precio **5,67** y en el segundo obj precio **18.34** esto nos demuestra que podemos albergar valores de distintos tipos. Luego vemos toppings que están representados como otro array[] En consecuencia tenemos un array externos que contiene a los dos obj y dentro del primer y segundo obj tenemos otro array como el valor de una propiedad→ **Toppings[]** con este ejemplo puedes ver que los obj son muy versátiles y con ellos podemos trabajar para cualquier dato que se necesite. Aquí representamos toppings como un **array** porque necesitamos varios elementos.luego para llevar es un valor booleano, true si se ordenó para llevar o false si la pizza es para comer en el lugar.

Como podemos acceder a estos obj? La estructura de datos principal que contiene los obj es un **array** es por eso que el primer obj esta en el índice cero, el segundo obj está en el índice uno y los obj están separados con una coma→esto es muy importante porque son elemento distinto del array. Para acceder al primer obj usamos el indise[0] y de igual manera para el segundo obj pero usamos el indise[1] Digamos que queremos acceder al tipo de pizza del primer obj. Para eso se pueden usar tanto notación de punto como notación de corchetes. Se accede a la primera propidad→tipo del primer obj y se muestra por consola el valor margarita XX a diferencia de notación de punto. La notación de corchete la podemos usar pero a no olvidar que el valor de la propiedad va con comillas “tipo”. También podemos acceder a la propiedad  **toppings.**

Como agregar una tercera orden de pizza:

Agregamos una coma en la llave de cierre del segundo obj y en la siguiente línea un par de llaves copiamos y pegamos el segundo obj para modificarlo y ya tenemos un tercer obj con el índice 2

var ordenesDePizzas = [

{

"tipo": "margarita",

"tamaño": "individual",

"precio": "5.67",

"toppings": [

"extra queso",

"chanpiñones",

"piña"

],

"paraLlevar": true

},

{

"tipo": "cuatro quesos",

"tamaño": "familiar",

"precio": "18.34",

"toppings": [

"extra queso",

"pimenton",

],

"paraLlevar": false

}

];

console.log(ordenesDePizzas[0]);//se accede al primer obj y se muestra por consola

console.log(ordenesDePizzas[1]); //se accede al segundo obj y se muestra por consola

console.log(ordenesDePizzas[0].tipo);//se accede a la primer propidad→tipo del primer obj y se muestra por consola el valor margarita

console.log(ordenesDePizzas[1].toppings)// veremos el array del segundo obj[“extra queso”, “pimentón”

console.log(ordenesDePizzas[2].tipo);//se accede a la primer propidad→tipo del tercer obj y se muestra por consola el valor napolitana

**Objetos Anidados**

Son objetos dentro de otros objetos, similares a los array anidados. En el ejercicio tenemos un obj principal que está asignado a la variable mi receta. Ese obj tiene 3 propiedades con sus elementos cuya primera propiedad del obj se denomina→descripcion, su segunda propiedad→costo y en la tercera propiedad que denominamos→ingredientes cuyo valor es un obj, es ahí donde este nuevo obj se contituye como anidado y a su vez tenemos masa y cobertura que se le asignan otros obj como propiedades.

var miReceta = {

"descripcion": "mi postre favorito",

"costo": 15.6,

"ingredientes": { //obj ingrediente

"masa": { //sub\_obj de ingrediente masa

"harina": "100 grs",

"sal": "2 cucharadita",

"agua": "1 taza"

},//cierre de sub\_obj masa

"cobertura": { //obj cobertura

"azucar": "120 grs",

"chocolate": "4 cucharadas",

"mantequilla": "200 grs"

} //cierre obj cobertura

} //cierre obj ingredientes

} // cierre obj principal miReceta

Con notación de punto accedemos a las distintas propiedades que mostrara por pantalla el elemento que corresponda a es propiedad. Cabe destacar que si usamos notación de corchetes remplazamos el punto por corchetes y la propiedad entre comillas y el resultado será exactamente igual. La ventaja de usar notación de corchetes es que si la propiedad está asignada a una variable la podemos usar. Y es obligatorio usar corchetes si la propiedad es una cadena de caracteres con espacios.

Var = "harina"

console.log(miReceta.descripcion);//con notación de punto

console.log(miReceta.costo);//con notación de punto

console.log(miReceta.ingredientes);//con notación de punto

console.log(miReceta.ingredientes["masa"]);//con notación de corchetes siguiendo el paht

console.log(miReceta.ingredientes.masa["harina"]);//con notación de corchetes siguiendo el paht

console.log(miReceta.ingredientes.masa["sal"]);//con notación de corchetes, array con espacios

**Array Anidados**

Los array anidados son estructuras más complejas. Como por ejemplo dentro de un obj que es parte de un array. Veamos un array que va a contener dos obj con listas de plantas. El primer obj va a contener dos propiedades, estas no se rodean con comillas porque el nombre de las propiedades son una sola palabra, **tipo y lista** El segundo obj vamos a usar las mismas propiedades→**tipo y lista.** Para ambos obj la propiedad→**lista** el valor será un array. De esta manera tenemos un array que alberga dos objetos y dentro de esos obj los array de las propidades→**lista** de manera tal que son lo que llamamos array anidados. Los array son parte de un obj y a su vez ese obj es parte de otro array. La sintaxis que tenemos que usar para acceder a sus elementos es distinta. La estructura principal que contiene lo los dos obj es un array de modo que el primer obj está en el índice [0] y el segundo obj en el índice [1]. Si queremos acceder al primer elemento de la primera propieda del obj [0] en este caso→flores el path seria misPlantas[0].lista[0] el puntero se sitúa en **rosas,**  y adentrándonos más profundo en el array principal para ver el primer elemento del obj[1] el path seria misPlantas[1].lista[1]; el puntero se sitúa en pino.

var misPlantas = [

{

tipo: "flores",

lista: [

"rosas",

"tulipanes",

"diente de leon"

]

},

{

tipo: "arboles",

lista:[

"abeto",

"pino",

"abedul"

]

}

];

var primeraFlor = misPlantas[0].lista[0];

console.log(primeraFlor);//vemos el valor rosas

var segundoArbol = misPlantas[1].lista[1];

console.log(segundoArbol);//vemos el valor pino

**Colección de Discos:**

Comenzaremos con un ejercicio ejemplificador: Tenemos un obj que representa parte de una colección de álbumes musicales. Cada álbum tiene un número de identificación único (id) "id número de identificación único" asociados a otra propiedades. No todos los álbumes tiene la información completa. Este ejercicio está enfocado en actualizar una colección de discos, usaremos obj, condicionales, acceder y eliminar las propiedades de un obj y mucho más. Digamos que este **id** está asociado a un ob anidado. En el código tenemos la colección de discos. Puedes ver que el obj está asignado a una variable y dentro de ese obj tenemos dos propiedades numéricas, son los **id** de cada uno de esos álbumes. Esos números están asociados a esos obj anidados que describen en más detalle el álbum. Tenemos un álbum con la data completa y tenemos otro álbum que no tiene todas esas propiedades.

var coleccionDeDiscos = {

7853 {

tituloDelAlbum: "Bee Gees Greatest",

artista: "Bee Gees",

canciones: ["Stayin Alove"]

},

5430: {

tituloDelAlbum: ABBA Gold"

}

};

A continuación vemos una descripción de cómo debe ser la función. Definimos la función actualizarDisco que tome los siguientes parámetros: -disco (el obj que representa la colección de disco). –id. –propiedad (“artista” o “canciones”). –valor. Tu meta es completar la función implementando las siguientes reglas para modificar el obj pasando a la función: -Si “valor” es una cadena vacía, elimina la propiedad del álbum correspondiente. –Si “propiedad” es igual a la cadena de caracteres “canciones” pero el álbum no tiene una propiedad llamada canciones crea un array vacío y agrega el “valor” a ese array.

Vamos a ir implementando estas reglas paso a paso, ya que el proceso es detallado primero vamos a definir la función. Para implementar la primera regla

function actualizarDiscos(disco, id, propiedad, valor)

{

{

if("valor" === "" )//si disco es exactamente igual a una cadena vacia entonces

{

delete disco[id][propiedad];//borramos la prpiedad del albun corespondiente

}

else if(propiedad === "canciones")

{

disco[id][propiedad] = disco[id][propiedad] || [];//accedemos al obj correspondiente, luego

//accedemos a la propiedad canciones y le acignamos un array vacio.

//Pero que pasa si la propiedad no existe... En ese caso usaremos el operador logico→or que nos

//permite escojer entre dos obciones: Primero se va a acceder, y si este valor no esta definido

// el operador or nos permite como espesificar un valor de respaldo que en este caso va a ser un

// array vasio[] y de forma segura podemos llamar al metodo push()

disco[id][propiedad].push(valor);//agregamos el valor a ese array vacio llamando al metodo push y

//agregamos el valor que pasamos como argumento

}

else

{ //↓cubrimos la cuarta regla

disco[id][propiedad] = valor;

}

}

//Trabajando con el primer obj↓

console.log(coleccionDeDiscos[7853].tituloDelAlbum);// antes de hacer el cambio

actualizarDiscos(coleccionDeDiscos, 7853, "tituloDelAlbum", "");//pasamos como argumento la

//coleccionDeDiscos en este caso NO la usamos como variable, pasamos el id, pasamos el nombre de la

//propiedad y el valor en una cadena vacia"" por que vamos a probar la primera condicion: if(...)

console.log(coleccionDeDiscos.tituloDelAlbum);//despues de hacer el cambio es undefined

console.log(coleccionDeDiscos[7853].canciones);//pasamos como argumento la coleccionDeDiscos

//en este caso NO la usamos como variable, pasamos el id, pasamos el nombre de la

//propiedad y el valor por que vamos a probar la tercra condicion: else{...}

//Trabajando con el segundo obj↓

console.log(coleccionDeDiscos[5430].canciones);// antes de hacer el cambio undefined

actualizarDiscos(coleccionDeDiscos, 5430, "canciones", "mamma mia");//pasamos como argumento la

//coleccionDeDiscos en este caso NO la usamos como variable, pasamos el id, pasamos el nombre de la

//propiedad y el valor que NO existen en el obj y probamos la segunda condicion→else if(...)

console.log(coleccionDeDiscos[5430].canciones);//despues de hacer el cambio ["mamma mia"]

console.log(coleccionDeDiscos[5430].artista);// antes de hacer el cambio undefined

actualizarDiscos(coleccionDeDiscos, 5430, "artista", "ABBA");

console.log(coleccionDeDiscos[5430].artista);//despues de hacer el cambio ["ABBA"]

console.log(coleccionDeDiscos[5430].canciones);//pasamos como argumento la coleccionDeDiscos

//en este caso NO la usamos como variable, pasamos el id, pasamos el nombre de la

//propiedad y el valor por que vamos a probar la tercra condicion: else{...}

**Ciclo “WHILE”**

Esos ciclos o bucles que en ingles se denomina loop nos permiten repetir unas secuencias de

instrucciones un número específico de veces. Tenemos dos tipos de ciclos:→ while o for, comenzaremos

viendo los ciclos while. Este tipo de bucles se los utiliza cuando no tenemos un número preciso de

interacciones. Una (interacción) es una repetición del bloque de código que queremos repetir.

Cuando no sabemos el número preciso que queremos realizar, pero si sabemos que es lo que tiene que

ser cierto para continuar el proceso usamos un bucle while. Nota:entre los while(...) va la condición

que tiene que ser true para continuar el proceso.

Vamos a definir una variable:

var cont = 0

while(cont <= 3)

{

Console.log(“Aprendiendo bucles con white”);

Cont++;

}

console.log("salio del bucle");

**Veamos otros dos ejemplo de un bucle White:**

Ej1) Definimos dos variables, una es un array vacío la otra un contador. Mostramos por consola la

variable miArreglo y veremos→ [[]] ¿por que el array sigue siendo un array vacío? Si si la variable

cont; no esta definida vemos los array vacios

//Llamamos al operador push()

var miArreglo = [];

var cont = 0;

//Si si la variable cont; no esta definida vemos los array vacios [[]]

console.log([miArreglo]);

while(cont <= 10)//hacer mientras se cumpla la condición

{

miArreglo.push(cont);//Con cada loop del bucle, push() agrega un numero

cont++;//incrementa en uno hasta que cont sea mayor que 10

}

console.log(miArreglo);//resultado [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

#Ej2 Llamamos al operador pop()

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

while(numeros.length > 5)//se usa length por que nos da el largo de la cadena

{

numeros.pop();//Con cada loop del bucle, pop() quita un numero hasta que la condicion sea false

}

console.log(numeros);//resultado [ 1, 2, 3, 4, 5 ]

var miArreglo = [];

var conta = 0

for(conta = 1; conta < 20; conta += 2)

{

miArreglo.push(conta);

}

console.log(miArreglo);

**Ciclo "for"**

Vamos hacer algo equivalente con lo que hicimos con el bucle while agregando un número del 0 hasta

el 9 de un array. Pero con el bucle for() el código se escribe diferente. La secuencias de

instrucciones que van entre llaves se denominan el cuerpo del bucle y las instrucciones condicionales

van entre los paréntesis de for(…), en lugar de escribir una condición como escribíamos en el bucle

while, para un bucle for tenemos que especificar tres aspecto de como se va a ejecutarse el bucle.

Lo primero que tenemos que hacer es inicializar el valor de una variable que se va a usar y actualizar

en el bucle. Veamos el primer aspecto →for(var conta = 0;) esa variable la denominamos la variable

del bucle. Esa variable decide si el bucle se sigue ejecutando o NO. Luego el segundo aspecto del

bucle separado por un ; tenemos la condición si el bucle continua o NO→ for(…; conta <10;) y luego

el tercer aspecto tenemos que decirle como queremos actualizar esa variable en cada iteración.

Si queremos incrementar el valor en uno escribimos for(…; …; conta ++) de esta forma quedaría

conformado→ fot(var conta = 0; conta < 10; conta++) {ahora pasamos al cuerpo del bucle lo que va

entre llaves}. Lo que haremos es llamar al operador push()→ miArreglo.push(miArreglo);

var miArreglo = [];

var cont = 0;

for(cont = 0; cont < 10; conta++)

{

miArreglo.push(cont);

}

console.log(miArreglo);//[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

**Ciclo for Numeros Imparaes:**

Declaramos el array para guardar los números impares en ese array. La condición : continua mientras variable < 20 y para obtener los números impares incrementamos la variable en dos. Seguidamente llamamos el método push() y ente los paréntesis la variable **conta**. Por ultimo mostramos el resultado

var miArreglo = [];

var conta = 1

for(conta = 1; conta < 20; conta += 2) {

miArreglo.push(conta);

}

console.log(miArreglo);//resultado→[ 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 ]

**Bucles for “Contar hacia atrás”**

Hasta ahora habíamos incrementado el valor de la variable del bucle, pero a continuación vamos a reducir su valor ya sea de reducirlo de 1 o reducirlo en -= 2 podemos también multiplicarlo o dividirlo entre dependiendo la condición

var miArreglo = [];

var decremento = 0

for(var decremento = 15; decremento >= 10; decremento--)

{

console.log(decremento);//resultado→15, 14,13, 12, 11, 10

}

var miArreglo = [];

var decremento = 0

for(var decremento = 15; decremento >= 10; decremento -=2)

{

console.log(decremento);//resultado→[15, 13, 11]

}

**Iterar Sobre un array con un bucle “for”**

Iterar sobre un array significa que vas a obtener cada uno de los elementos de ese array y los vas a poder usar en el cuerpo del bucle. Los mismos principios que veremos aplican para cualquier tipo de datos que contenga el array, cualquier tipo de elemento, incluyendo obj, cadena de caracteres, array anidados y cualquier otro tipo de dato. Definimos el array con los valore [4, 6, 8, 2] En este caso vamos a sumar los valore usando un bucle y el total de la suma se va a ir acumulando en la variable→**total** que inicial mente tiene un valor cero. Para eso vamos a tener que acceder a cada uno de esos valores en una iteración del bucle. En cada loop tenemos que ir acumulando esos valore en el **total**←[4]+[6]+[8]+[2] que es la variable del bucle. La condición del **for** va ser que mientras la variable **puntero** se menor que la longitud del array entonces continuando el proceso vamos a incrementar el valor de **puntero** en uno antes de la próxima iteración. En el cuerpo del for vamos a implementar que va hacer el bucle, agregaremos un valor a la variable **total** con el operador += vamos a sumar el valor en el índice **puntero** del array. En laprimera interacion el puntero va a ser cero, en la segunda seis en la tercera 8 y en la cuarta dos.

[0, 1, 2, 3]

↑ ↑ ↑ ↑

4 6 8 2

Y podemos mostrar en la consola luego del proceso Nota: Usar console.log dentro del bucle es muy útil para ver cómo funciona y como se van actualizando los valeres a medida que se ejecutan las interaciones.

mir miArreglo = [4, 6, 8, 2];

var total = 0;

for(var puntero = 0; puntero < miArreglo.length; puntero++)

{

total += miArreglo[puntero];

console.log("Interacin " + puntero);

console.log(miArreglo[puntero]);

}

console.log(total);

En otro ejemplo de iteración con un array: Digamos que tenemos un array de un conjunto con cadenas de caracteres de lenguajes de programación en minúscula y con un bucle for la queremos convertir en mayusculas entonces podemos iterar sobre el array. Inicializamos la variable →**i** en cero para usarla como índice y vamos a continuar iterando mientas el valor de→i sea menor que la longitud del arreglo **lenguaje**, esa condición es prácticamente fija si comienzas desde cero y quieres iterar sobre todos los elementos de un array asique esto se va a volver algo rutinario que se va a usar→(var i = 0; i < lenguajes.length; i++) por ultimo vamos a incrementar el valor de i++ ahora vamos a mostrar el elemento pero en mayúscula. Paraeso vamos a llamar un método especifico, primero accedemos al elemento de la cadena de caracteres que contiene el array. Para eso escribimos el nombre del array→ **lenguaje** **[i]** el índice que es la variable→**i** y para convertir esa cadena de caracteres en mayúscula usamos el método **toUpperCase().**

**En otro ejemplo: vamos a trabajar en un bucle for dentro de una función:**

Vamos a definir una función que va a contar la cantidad de números pares en el array va a tomar un parámetro→arreglo y va a mantener un acumulado del total de números pares que se han encontrado hasta el momento. Ese total se va actualizar a medida que consigamos números pares en el array por que vamos a iterar sobre todos sus elementos. Dentro de la función podemos escribir un bucle for y cómo vamos iterar sobe el array? En la condicional del **for** usamos el mismo patrón que ya hemos visto→ **for(var i = 0; i < lenguajes.length; i++).** Dentro de la función inicializamos una variable→total en cero y escribimos un condicional **if** (arreglo[y] % 2 == 0) que le pedimos? que el elemento[y] que corresponde a esa iteración es par si el resto de su división entre dos es igual a cero entonces incrementamos la variable→ total en uno y cuando el bucle termine y hayamos procesados todos los elementos del array retornamos el valor de la cantidad de números pares. En el ejemplo como resultado el valor es 4

function contarNumerosPares(arreglo)

{

var total = 0;

for(var y = 0; y < arreglo.length; y++)

{

if(arreglo[y] % 2 == 0)

{

total++

}

}

return total;

}

console.log(contarNumerosPares([1, 8, 3, 6, 5, 4, 2]));//resultado 4

**Bucles for anidados:**

También es muy útil aprender bucles dentro de otros bucles comúnmente denominados bucles anidados. Ej. de un bucle for anidado: Son muy útiles para procesar estructuras de datos multidimensionales. Tenemos un array que dentro del mismo contiene otros array→ [[], [], []] Como podemos hacer alguna operación con cada uno de los elementos de cada uno de los array, si usamos un bucle for vamos a usar el mismo patrón para iterar sobre los elementos de un array →for(var indiceI = 0; indiceI < nidosObj.length; indiceI++) Recordemos: la variable del índice que en este caso es→**indiceI** la inicializamos en cero, seguidamente separado por **;** decimos que mientras **indiceI** sea menor que la longitud del array incremente **indice I** en uno por cada loop . Vamos a mostrar cada uno de esos elementos que obtenemos. miArreglo[**indiceI**] que por el momento es cero ya que tenemos tres array con índice 0 1 2, se recuerda que como son array anidado cada uno tiene un índice especifico. Si usamos console.log(nidosObj[indiceI]); el resultad por consola seria[1,2,3] [4,5,6][7,8,9] esto es útil cuando trabajamos con el array anidado. Pero qué pasa si tenemos que trabajar con los elementos de ese array anidado. Algo que podemos hacer es asignar ese array a una variable→ **miArregloAnidadi** =**nidoObj[indiceI];**  y luego podemos iterar sobre los elementos usando otro bucle **for()** sobre ese array anidado. Aquí ya se pude ver el patrón, Vamos hacer lo mismo que con el array externo pero con el array anidado, es el mismo patrón pero con una variable distinta→ **for( var nidoUno = 0; nidoUno < arregloAnidado.length; nidoUno++)** De igual manera que en el bucle anterior vamos a mostrar cada uno de esos elementos que obtenemos del arreglo anidado **console.log(nidosObj[nidoUno]);**→este es un Elemento del array anidado. En el ejercio al pie se pude apreciar el desarrollo completo.

var nidosObj = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

for(var indiceI = 0; indiceI < nidosObj.length; indiceI++)

{

console.log(">> Nueva iteracion");

var arregloAnidado = nidosObj[indiceI];//Array

console.log("Arreglo " + arregloAnidado);

for( var nidoUno = 0; nidoUno < arregloAnidado.length; nidoUno++)

{

console.log(">>> Bucle anidado");

console.log("Elemento:");

console.log(arregloAnidado[nidoUno]);//Elemento

}

}

Resultado de los console.log

>> Nueva iteracion

Arreglo 1,2,3

>>> Bucle anidado

Elemento:

1

>>> Bucle anidado

Elemento:

2

>>> Bucle anidado

Elemento:

3

>> Nueva iteracion

Arreglo 4,5,6

>>> Bucle anidado

Elemento:

4

>>> Bucle anidado

Elemento:

5

>>> Bucle anidado

Elemento:

6

>> Nueva iteracion

Arreglo 7,8,9

>>> Bucle anidado

Elemento:

7

>>> Bucle anidado

Elemento:

8

>>> Bucle anidado

Elemento:

9

**Ciclos o Bucles “Do…While”**

Vamos a ver un tercer tipo de bucle que podemos usar el JS que es básicamente un bucle while pero con un pequeño cambio es el bucle→**do while** que traducido seria (hacer mientras) comencemos con un ejemplo del bucle **while.**

Ej. bucle while

var x;

x = 5;

while(x < 10)

{

console.log(x);

x++;

} while(x < 10)

El bucle **do while** nos permite otra cosa, la diferencia de este bucle con el bucle **while** es que al menos una vez siempre se va a ejecutar las secuencias de instrucciones que están ente las llaves por que la condición esta al final. Si esa condición es true hará un loop y si es false continua.

Ahora bien, paraqué puede llegar a servir un bucle **do while** Una de las situaciones ideales para un bucle **do while** es cuando necesitas que el usuario ingresé un valor y se tiene que verificar si ese valor es válido o no.

var x;

x = 5;

Do

{

console.log(x);

x++;

} while(x < 10);

**Búsqueda de perfil, Mini proyecto**

Ahora vamos a combinar array, objetos, bucles y funciones. Vamos a definir una función que nos va a permitir buscar en una lista un contacto. Digamos que tenemos en este mini proyecto una variable que contiene un array y dentro de ese array hay tres objetos que cada uno de ellos ocupa un índice específico en ese array. Dichos objetos alberga una lista de propiedades y elementos que lo denominaremos contactos.

var contactos = [

{

"nombre": "Nora",

"apellido": "Nav",

"numero": "0543236543",

"gustos": "" ["Pizza", "Programacion"]

},

{

"nombre": "Harry",

"apellido": "Potter",

"numero": "0994372684",

"gustos": "" ["Hogwarts", "Magia"]

},

{

"nombre": "Sherlock",

"apellido": "Holmes",

"numero": "0487345643",

"gustos": "" ["Casos interesantes", "Violin"]

}

];

function buscarPerfil(nombreP, propiedad)

{

for(var i = 0; i < contactos.length; i++)

{

if(contactos[i].nombre === nombreP)//propiedad === parametro

{

return contactos[i][propiedad] || "La propiedad no existe";

}

}

return "El contacto no está en la lista";

}

console.log(buscarPerfil("Nora", "nombre"));//indice cero

console.log(buscarPerfil("Harry", "apellido"));//indice uno

console.log(buscarPerfil("Sherlock", "numero"));//indice dos

console.log(buscarPerfil("Manuel", "apellido"));

**Definición de la función:**

Esta function va a permitirnos conseguir una propiedad de un contacto, en base a su nombre vamos a obtener su valor, obviamente si la propiedad existe. Primero dentro de la función escribimos un bucle for para poder iterar sobre los elementos del array, vamos a acceder al array directamente con la variabe→contactos que es una variable global y la podemos usar dentro de las funciones del programa. for(var i = 0; i < contactos.length; i++ ) puedes ver el mismo patrón que hemos usado anteriormente para iterar con los elementos de un array. Dentro del bucle **for** que vamos hacer tenemos un condicional con el cual vamos acceder a c/u de los contactos de los objetos usando el índice→i que inicialmente su valor es **cero** luego cuando se actualice a **uno** podemos acceder al segundo objeto y de igual manera al tercero. Escrivimos un condicional →**if(contactos[i].nombre === nombreP)** //propiedad === parámetro. Que le estamos pidiendo al if? Si la variable→contacto[i]←es0 entonces con notación de punto preguntamos si la propiedad→**nombre** es igual al parámetro→nombreP que solicitamos en este caso el elemento→**“Nora”.** Si eso es true tenemos que retornar el valor de la propiedad que corresponde y en el **return** verificar si la propiedad existe.

**return contactos[i][propiedad] || "La propiedad no existe";** Vamos a tomar ese contacto[i]→el que estamos verificando y vamos acceder a la propiedad que especificamos como parámetro. Aquí no podemos usar notación de punto porque [propiedad] es una variable, si usáramos notación de punto en lugar de notación de corchetes se buscaría una propiedad cuyo nombre sea propiedad. O sea no se va a remplazar el valor de la variable si no que se va a buscar una propiedad con este nombre y eso no es lo que queremos. En realidad el objetivo es que se remplace aquí el valor de esta variable del parámetro. Si esa propiedad existe su valor se va a retornar, pero también tenemos que manejar el caso que la propiedad no exista. En ese caso retornamos una cadena de caracteres que avise lo ocurrido En JS hay una forma muy sencilla y concisa de escribirlo. Decimos si la propiedad existe retorna esa propiedad pero si esa propiedad no existe ese valor va a ser **ondefine** y por lo tanto debemos usar el operador or→||y retornamos→"La propiedad no existe"; También está la posibilidad del que contacto no exista. En ese caso el bucle for va a ejecutar todas sus iteraciones, una por cada contacto en la lista. La condición del if nunca va a ser verdadera y por lo tanto se va a completar este bucle y se va a pasar a la siguiente línea→ return "El contacto no está en la lista";

De este modo la función está terminada y ahora comprobaremos con console.log distinto requerimientos.

console.log(buscarPerfil("Nora", "nombre"));//indice cero

console.log(buscarPerfil("Harry", "apellido"));//indice uno

console.log(buscarPerfil("Sherlock", "numero"));//indice dos

console.log(buscarPerfil("Manuel", "apellido"));//no existe

**Números Aleatoreos:**

Como generamos un número aleatorio en JS, simple mente tenemos que usar la palabra MATH vamos a generar una función no va a tomar argumentos o sea sin parámetro si no que va a retornar un valor aleatorio. Ese valor lo vamos a generar con **Math** con la M mayúscula representa un obj que nos da funciones matemáticas básicas y constantes, una de esas funciones matemáticas que podemos usar es la función **random()** que significa aleatorio esta función no toma ningún parámetro para retornar un valor aleatorio. El método retorna un valor aleatorio entre cero y uno, pero el uno no está incluido en los posibles resultados. En notación matemática seria→**[0, 1)** asique para generar un rango tendremos que hacer algo específico. Con console.log() llamamos a la función y el resultado es un numero con muchos decimales→0.7174485152982049 ese número aleatorio va a ser muy útil para generar números aleatorio en un rango especifico, por ejemplo si necesitamos un numero ente 3 y 5 entre 10 y15 vamos a poder hacerlo en base a los valores decimales que habrá que transformar, como ya dijimos, nunca el resulta del número entero del decimal va a ser uno. No necesariamente tenemos que llamar a Math.random dentro de una función. Podemos usar la siguiente código, lo asignamos a una variable var numeroAleatorio = Math.random(); esto va a generar el valor. Y si mostramos console.log(numeroAleatoreo) dará como resultado 0.7174485152982049.

function generarFuncionAleatoreo()//no necesariamente se usa una función

{

return Math.random();

}

console.log(generarFuncionAleatoreo());

var numeroAleatorio = Math.random();//Directamente usamos una variable

console.log(numeroAleatorio);

**Números Enteros Aleatorios**

Así como puedes generar números decimales aleatorios también podemos escribir una expresión que nos permita convertir esos números decimales a números enteros dentro de un rango específico. Para obtener un número aleatorio entre 0 y 19 escribimos el siguiente código

**var numeroAleatoreoEntero0y19 = Math.floor();** Math vimos que es un obj que nos permite usar funciones y constantes matemáticas esenciales. Una de esas funciones representada como un método **es floor()** que retorna el mayor número entero que es menor o igual a su argumento numérico al número que pasamos entre paréntesis . Lo que hace **floor()** técnicamente truca la parte decimal o sea básicamente toma la parte entera de lo que está entre paréntesis. Es ahí donde tenemos que generar ese número aleatorio Math.random()\*20); recuerda que ese número aleatorio va a estar entre 0 y 1 cero va a ser uno de esos valores, pero 1 nunca va a ser esos valores. Como podemos transformar ese valor en un entero y así tener el rango que queremos. Hay que multiplicarlo por un entero. Veamos la lógica: este valor→ Math.random va a ser un decimal ente 0 y 1 sin incluir 1 como no puede ser uno este producto nunca va a ser 20→Math.random()\*20); va a ser 19 punto y algo y como estamos aplicando la función **floor()** ese número va a ser procesado a 19 lo que se va asignar a esta variable→ numeroAleatoreoEntero0y19, ahí ya tenemos un límite superior para el rango de valores que podemos obtener y el valor de Math.random puede ser 0 es el mínimo valor que puede tener. Así que 0x20=0 y cuando le aplicamos floor el resultado seria 0 y el valor en la variable seria 0 así que estamos obteniendo un rango de valores que va de 0 a 19 inclusive y es exactamente lo que se necesita.

Sintaxis de código→**var numeroAleatoreoEntero0y19 = Math.floor(Math.random()\*20);**

Si mostramos este número en la consola podemos verlo: resultado aleatorio ente 0 y 19

var numeroAleatoreoEntero0y19 = Math.floor(Math.random()\*20);

console.log(numeroAleatoreoEntero0y19);//resultado aleatorio ente 0 y 19

**Algo interesante para tener en cuenta:**

Vamos a definir una función que genere un número entero aleatorio, esa función va a tener un solo parámetro, el límite superior del rango de número que queremos generar vas a escribir la expresión que tenemos antes de forma general pero esta vez en lugar de multiplicar por 20 que era un número fijo vamos a multiplicarlo por el valor del parámetro que es el límite superior→ return Math.floor(Math.random() \* limiteSuperior); Entonces eso ya nos está permitiendo generar valore enteros entre cero y el valor que le asignemos al parámetro→limiteSuperior (sin incluirlo)

function generarEnteroAleatorio(limiteSuperior)

{

//Generar un valor entero entre 0 y el limite superior (sin incluirlo)

return Math.floor(Math.random() \* limiteSuperior);

}

Si queremos intentar generar muchos números aleatorios. Para esto podemos agregar un bucle for() digamos que queremos repetirla 5 veces

for(var nv = 0; nv < 5; nv++)

{

console.log(generarEnteroAleatorio(1225));//5 resultados aleatorios entre 0 y 1224

}

**Números Enteros Aleatorios en un Rango**

Anterior mente estábamos en parte limitados a generar un numero entre cero y un límite superior pero también podemos generar números enteros aleatorios en un rango especifico con un límite inferior y un límite superior, digamos ente 5 y 15, o cualquier otro rango de números enteros. Para eso definimos una función que va a tener dos parámetros. Para obtener un número entero en ese rango aleatorio vamos a tener que escribir una expresión parecida a la anterior pero con ciertos cambios. Primero tenemos otra vez Math.floor() ese si lo vamos a volver a usar, también tenemos que generar un numero aleatorio entre 0 y 1 pero en este caso en lugar de solo multiplicar por el límite superior vamos a tener que multiplicar x una expresión un poco más compleja. Esa expresión es el limiteSuperior menos el limiteInferior + 1 y luego a ese resultado luego de haber redondeado ese resultado→Math.floor(Math.random() \* (limiteSuperior - limiteInferior + 1) se le suma el limiteInferior y esta es la expresión que nos permite genera un entero en ese rango. Seguimos usando el método floor() y el método random() pero lo multiplicamos por los valores de la resta de los parámetros→(limiteInferior, limiteSuperor+1) eso nos da como el tamaño, el intervalo más 1 porque queremos incluir el límite inferior y luego a ese valor que se genera le vamos a sumar el límite inferior, eso nos va a retornar un entero en ese rango. Eso lo vamos a comprobar con algunas pruebas, como en el ejerció anterior usaremos un bucle for() En este caso vamos a ver que el límite superior si es posible que se incluya en los resultado.

Math.random()

function rangoAleatorio(limiteInferior, limiteSuperor)

{

Math.floor(Math.random() \* (limiteSuperior - limiteInferior + 1)) + limiteInferior;

}

for(var rango = 0; rango <= 5; rango++)

{

console.log(rangoAleatorio(3, 8));

}

**Función parselnt()**

En JS podemos usar una función específica para convertir una cadena de caracteres que representa un número e un número entero usando→console.log(parseint(“5”)); La nostramo en consola y vemos 5. De igual forma es equivalente para números negativos, el cero y decimales, pero un detalles con el numero decimal es que elimina la parte decimal y nos devuelve un entero. Veamos en la práctica las ventajas de usar esta función. Tenemos dos variable que están asignadas con cadenas de caracteres var a = “5”; var b = “7”; Pero en realidad queremos trabajar con enteros. Si necesitamos sumar los valores tal cual están asignados optendriamos una concatenación de dos cadenas a+b resultado 57. En ese caso lo que tenemos que hacer es convertir las cadenas de caracteres. var a = parseint(“5”); var b = parseint(“7”); llamando a esa función el valor retornado va a ser un entero con todas sus propiedades y el resultado seria 12

**Funcion parseint() con una Base**

Tambien se puede usar la función parseint() de una forma más potente porque puedes convertir una cadena de caracteres en otro sistema numérico al sistema decimal como un entero. Veamos un ejemplo: Vamos a mostrar el resultado con console.log y vamos a llamr a la función parseint() pero en este caso vamos a pasar una cadena de caracteres con ceros y unos o sea un sistema vinario. Para espesificar que el sistemas es vinario tenemos que pasar un segundo argumento con un valor 2 console.log(parseint(110111, 2)); el resultado de la cadena vinaria del sistema decimal es el número entero 55 Si en vez de colocar en números vinarios colocamaos el stingt “55” el resultado seria 55. Veamos un ejemplo con el sistema hexadecimal cuya base es 16 console.log(parseint(“3E0A”, 16)); En el sistema hexadecimal podemos usar números y letras y base 16, aquí como tenemos letras necesitamos representarlo como una cadena de caracteres “ ” El resultado en este caso es el numero entero 15882

**Operador Condicional (Ternario)**

El operador condicional también llamado ternario nos permite compactar lo que sería un condicional en una sola línea. En el ejemplo definimos una función llamada retornar mínimo esta función va a tomar dos parámetros y va a retornar el mínimo de los valores que contengan los parámetros usando un condicional If()

function retornarMinimo(x, y)

{

If(x < y)

{

return x;

}

}

Else

{

Return y;

}

El condicional if va a funcionar correctamente, pero hay una forma más concisa de escribir estas secuencias de instrucciones y es el operador condicional no tan larga o en una sola línea. Veamos

function retornarMinimo(x, y)

{

Return x < y ? x : y;//(? seria entonses) (): seria si no)

}

Si la condición es true se remplaza por el valor de (x ) y si es false se remplaza por el valor de (y)

Si en vez de return vemos esto con consle.log (x < y ? x : y); lo que sucedería seria mostrar cualquiera de los valores de las variables dependiendo de la condición.

function retornarMinimo(x, y)

{

//return x < y ? x : y;→(?→seria entonces) (:→seria si no)

console.log(x < y ? x : y);

}

retornarMinimo(9, 11); resultado 9

retornarMinimo(8, 2); resultado 2

retornarMinimo(5, 5); resultado 5

**En otro ejemplo**

var v\_A = 7;

var v\_B = 9;

console.log(v\_A > v\_B ? v\_A + 2 : v\_B \* 3);// el resultado es el valor de v\_B 27

**Múltiples Operadores Condiciónale o Ternarios:**

Se pueden combinar los operadores condicionales o ternarios escribiendo múltiples operadores escribiendo uno dentro del otro, similar al concepto de anidar. Para esto vamos a escribir una función que va a comparar dos número mediante los parámetros p\_A y p\_B si estos valores son iguales es que tenemos garantizado que son del mismo tipo de datos. Eso lo especificamos en la documentación de la función. En ese caso retornamos que p\_A y p\_B son iguale, si ese no es el caso y p\_A > p\_B retornamos una cadena que diga “Que p\_A es > que p\_B y por ultimo si p\_A es < p\_B retornamos una cadena indicando que p\_A es < p\_B. Escribimos la función en cuyo cuerpo contiene el condicional if() para saber que queremos lograr.

function compararNumeros(p\_A, p\_B)

{

if(p\_A == p\_B)

return "p\_A y p\_B son iguales";

else if(p\_A > p\_B)

{

return "p\_A es mayor p\_B";

}

else

{

return "p\_A es menor p\_B";

}

}

Bien ya tenemos nuestra función con el condicional **if()**  ¿Cómo podemos remplazarlo con un operador ternario? Lo primero es en el cuerpo de la función escribir la sentencia **return** porque vamos a devolver el valor del operador condicional. Después de **return** lo primero es escribir la **condición p\_A = = p\_B** seguido de→ **?** si es **true** escribimos la cade na de caracteres **“p\_A es iguales p\_B”** esto vendría a ser la condición del **if.** Seguidamente con un salto de linea colocamos los dos puntos→ **:** nuevamente la condición que vendría a ser la del **elfe If**→(p\_A > p\_B) seguido de→ **?** y la cadena de carecteres "p\_A es mayor p\_B" y por ultimo con un nuevo salto de línea nuevamente los dos punto **:** seguido de la cadena de caracteres"p\_A es menor que p\_B"; cerrando ahí con un **;** esto es lo que vendría a ser el **else** Al final mostramos por consola las distintas cadenas segun los valores asignados

function compararNumeros(p\_A, p\_B)

{

return p\_A == p\_B ? "p\_A y p\_B son iguales"

: p\_A > p\_B ? "p\_A es mayor que p\_B"

: "p\_A es menor que p\_B";

}

console.log(compararNumeros(7, 7));

console.log(compararNumeros(12, 8));

console.log(compararNumeros(5, 9));

**Tipo de variables: var vs. let**

Vamos a ver la diferencia de declarar una variable con **var** o declararla con let. Una de las diferencias principales es que declarar una variable con **var** es que podemos repetir la declaración una o más veces con el mismo nombre**. var campista = Manu/ var campista = José** y JS no va a tirar ningún error. Pero cuando usamos **let** la situación es diferente porque con **let** no podemos declarar una variable más de una vez. Si lo hiciéramos JS mediante un aviso te lo recordaría como error de sintaxis, lo que si se puede hacer es reasignarle el valor.

**Ámbito de var vs. Let**

La diferencia en el ámbito de las variables que son declaradas con var y con let . Cuando usamos **var** estamos creando alguna de estas dos opciones: Una variable global si está dentro del programa principal dicha variable la podemos usar dentro *de cualquier ámbito* o una variable local si está dentro del ámbito de una función y solo ahí podremos usarla. Cuando usamos ***let*** estamos restringiendo aún más el ámbito de la variable, en bloque o donde se usa el bloque de código, su alcance está limitado a esa expresión, al bloque o donde se define. Digamos que queremos escribir un bucle for() en el programa. Si usamos ***var*** en el bucle esa variable va a tener el ámbito que corresponda donde está definido el bucle. En el ejemplo estamos en el ámbito global del programa principal podemos usar la variable aunque el bucle haya finalizado. A diferencia de ***var*** escribimos ***let*** la variable solo se podrá usar dentro del bucle y en consecuencia en el console.Log fuera de bucle nos daría un error de que la variable no esta definida

for(var vY = 0, vY < 3, vY++)

{

console.log(vY);

}

console.log(“Variable:” vY);

También hay una diferencia notable cuando usamos ***let*** en un condicional

var mostrarColor = true;

if(let color = “verde”)

{

console.log(“Mi color favorito es” + color);//resultado verde

}

console.log(“Mi color favorito es” + color);//resultado ReferenceError: color is not defined

Lo anteriormente visto son las diferencias principales entre ***var*** y ***let***

**Tercera opción de variable *const***

Esta tercera opción de variable en JS, la palabra reservada ***const*** que es una abreviación de constante. Son variables de solo lectura, solo pueden ser usadas para acceder a ese valor pero no podemos modificar el valor cuando haya sido asignado por primera vez. Veamos un ejemplo: Si declaramos una variable y le asignamos un valor→ const miConstante = 35; con cosole.logo() si podemos acceder al valor y mostrarlo por consola cosole.logo(miConstante) ; pero que sucede si queremos cambiarle el valor a esa variable miConstante = 15; lo que va a suceder es que JS nos retornara un error→TypeError: “miConstante” is read-only Como norma los desarrolladores a la variables declaradas ***const*** la escriben con mayúscula con un guion bajo→MI\_CONSTANTE

Ahora vamos a ver un ejemplo de una función para calcular el área de un círculo usando la declaración de variable ***const*** Si no definimos PI con una ***const*** se podría en algún momento cambiar el valor que contiene y los cálculos serian erróneos

Nota: Dos asteriscos es el operador que nos permite elevar al cuadrado, o cualquier otra potencia

function calcularAreaCirculo(radio)

{

const PI = 3.14;

if(radio < 0)

{

return undefined;

}

return PI \* (radio \*\* 2);

}

console.log(calcularAreaCirculo(10));//resultado 314

\*/

**Mutar Arrays Declarados con** ***const***

Ahora vamos a ver un aspecto de ***const*** que debes tener presente cuando lo uses en JS. Cuando declares una variable constante eso no significa que ese valor no puede cambiar!! Significa que no puedes asignar un valor nuevo a esa variable. Para ilustrar esto vamos a ver cómo podemos mutar un array que fue declarado con una constante. Usamos ***const*** con la convención establecida para declarar MI\_ARREGLO Si intentamos reasignar el valor ~~MI\_ARREGLO = [5, 6, 7, 8];~~ va a ocurrir un error ya que las constantes son solo lectura. Lo que se debe tener presente es que si podemos cambia el contenido del array con la notación de corchetes→ Ejemplo:

const MI\_ARREGLO = [1, 2, 3, 4];

MI\_ARREGLO[0] = 5;

MI\_ARREGLO[1] = 6;

MI\_ARREGLO[2] = 7;

MI\_ARREGLO[3] = 8;

console.log(MI\_ARREGLO);//resultado [ 5, 6, 7, 8 ]

**Crear un Objeto Inmutable**

Una forma de hacer que un objeto sea inmutable en JS . Veamos esto con un ejemplo. Definamos un obj con ***let*** simplemente para usar este tipo de variables. Vamos a asociar las propiedades colores con su elemento con el código hexadecimal. Digamos que no queremos que ese obj mute en ningún momento tampoco que se altere su contenido, básicamente queremos congelarlo. Precisamente esa posibilidad está disponible en JS con el método ***freeze***. Lo haremos con ***object*** cuya descripción: nos da funcionalidad que es común para todos los objetos en JS pero en particular nos da acceso al método ***freeze*** este es un método que previene la modificación de atributos de propiedades existentes y sus valores y la incorporación o adición de nuevas propiedades al objeto Si intentamos agregar una propiedad tendríamos un error de tipo por que el obj es inaccesible.

let colores = {

"verde": "#10e04b",

"azul": "#1b50e0",

"negro": "#000000",

"blanco": "#ffffff"

}

object.freeze(colores);//pasamos la variable que contiene el obj

**Funciones Flechas**

Las funciones flechas => son un tipo más compacto de funciones que podemos escribir en JS que normalmente se utilizan cuando necesitamos definir funciones anónimas o sea que no cuentan con un nombre especifico. En el ejemplo vemos una función tradicional que solo crea un obj con la fecha actual. Esta misma función la vamos a remplazar por una función flecha.

Quitamos la palabra clave function, dejamos los paréntesis y luego quitamos las llaves y la sentencia return en este caso solo porque tenemos una sola línea y por ultimo escribimos el símbolo flecha que no es ni nada más ni nada menos que un igual seguido de mayor que =>

const fecha = function()

{

return new DATE();

};

El símbolo que nosotros denominamos función flecha toma los parámetros que están enlazados por paréntesis que en este caso es ninguno y retorna el obj a si es como definimos una función mucho más compacta. Estas funciones son muy útiles cuando necesitamos pasar una función como argumento a otra función, lo que hacemos es escribirla como argumento

const fecha = () => new DATE();

**Funciones Flechas con Parametros:**

En el ejemplo anterior la función flecha no tomaba ningún parámetro solo retornaba un objeto. En este ejemplo vamos a definir una función anónima que va a tomar el valor p\_X y va a retornar el valor pX + 3

//Código tradicional

const sumarTres = function(pX)

{

return pX + 3;

};

console.log(sumarTres(5));//El valor retornado es 8 porque pX vale 3 y se suma el valor ingresado

//Código flecha

const sumar\_Tres = (p\_X) => p\_X + 3;

console.log(sumar\_Tres(4));//El valor retornado 7 porque p\_X vale 3 y se suma el valor ingresado

Ahora vamos a escribir una función anónima la cual va a contener dos parámetros y vamos a concatenar esos valores usando el método concat()←este es un método que puedes llamar para unir un array con otro.

//Código tradicional

const concatenarArray = function(arreglo1, arreglo2)

{

return arreglo1.concat(arreglo2);//metodo concat() une un array con otro

}

console.log(concatenarArray([1, 2], [3, 4, 5]));

//Código flecha

const concatenar\_Array = (arr1, arr2) => arr1.concat(arr2);//metodo concat() une un array con otro

console.log(concatenar\_Array([6, 7], [8, 9, 10]));

Estos casos que estamos viendo son funciones que en el cuerpo solo tenían una línea que retornaban un valor. Si la función original tiene más de una línea tenemos que mantener las llaves eliminamos function y después de los paréntesis agregamos =>

Veamos un ejemplo:

const sumar = (p\_M, p\_N) => {

let num = 6;

return p\_M + p\_N + num;

};

console.log(sumar(1, 1));//resultado es 8

**Valores por defectos para Parámetros:**

También puedes asignar un valor por defecto a los parámetros de una función flex, esto es útil cuando el usuario omita algún parámetro para usar el valor por defecto. En el ejemplo definimos una función flex.

**const incrementar = (num, valor = 1) => num + valor;**

**console.log(incrementar(5));//resultado es 6**

Esta función cuenta con dos parámetro y vamos a retornar el num + el valor del incremento. para asignar un valor por defecto al parámetro valor, simplemente mediante el signo de asignación = colocamos el valor, así si omitimos ese parámetro se va asignar por defecto en este caso 1. Este principio también aplica a las funciones normales y a las funciones anónimas.

**Operador rest:**

Este operador te va a permitir escribir funciones muy versátiles. Este operador te permite pasar cualquier número de argumento a una función y que esos argumentos se agrupen como un array. Ese número de argumentos no va a ser fijo asique la técnica que hemos estado usando de escribir parámetros en la lista no nos va a funcionar porque al momento de llamar a la función cada uno de esos parámetros debe tener un valor y no podemos pasar más valores o menos valores de lo que explícitamente tenemos declarados en la lista. Pero para solucionar esto para crear funciones más versátiles usamos el operador ***rest()*** que consiste en escribir tres puntos **…** que representa al operador ***rest()*** seguido de una palabra cualquiera ya que estamos creando una variable. Se usa ***args*** porque representa argumentos. Como ***args*** es una variable va a ser un array podemos usar también su propiedad ***leng th*** para trabajar con ese array en la función y también esta propiedad es muy útil si queremos procesar un bucle que es lo que normalmente haríamos para procesar c/u de esos argumentos.

function miFuncion(...args)

{

console.log(args);

console.log(args.length);

}

miFuncion(1, 2, 3, 4, 5);//resultado [ 1, 2, 3, 4, 5 ] tamaño 5

miFuncion([ 1, 2, 3 ], [ 4, 5, 6 ]);//resultado [ [1, 2, 3], [4, 5, 6] ] tamaño 2

/\*

[ 1, 2, 3, 4, 5 ]

5

[ [ 1, 2, 3 ], [ 4, 5, 6 ] ]

2

Operador rest para tomar cualquier número de argumentos

const sumar = (~~x, y ,z~~) => {

~~const args = [x,y,z];~~

return args.reduce((a, b) => a + b, 0;

};

.reduce()→ con estos argumentos suma los elementos del array y retorna los resultados

Pasamos una función→ flecha como argumento

Para adaptar esta función tenemos que modificar la lista de parámetro que contiene la variable sumar que son tres “esta tachado” y lo remplazamos por el operador ***rest*** para tomar cualquier número de parámetros const sumar = (…args) ya como este valor va a ser un array no es necesario tener otro array separado “esta tachado” y por ultimo conservamos la línea de retorno tal cual está en el ejemplo

const sumar = (…args) => {

return return args.reduce((a, b)) => a + b, 0;

}

**Operador Spread()**

Este operador hace exactamente lo contrario que el operador ***rest.*** El operador ***spread()*** toma un array lo descompone en sus elementos individuales para que la función pueda recibirlos y asignarlos a sus parámetros correspondientes.

Vamos a declarar un array con tres números y vamos a definir una función con tres parámetros que toma tres números y retorna la suma. En este caso vamos a usar el operador ***spread()*** porque no se puede pasar directamente ese argumento de la variable números porque contiene un array a la función su\_mar que tiene tres parámetro. si llamamos a la función en lugar de escribir (numero[0], numero[1], numero[1] ) tenemos un operador que nos permite hacer exactamente esto su\_mar(…) como vemos en vez de pasarlo en la lista de parámetros lo pasamos en la llamada a la función→su\_mar(…números)←escribimos el nombre de la variable y **…**←esto básicamente lo que va hacer es descomponer el arreglo en sus elementos individuales y asignarlo en ese mismo orden a los parámetros.

function su\_mar(p\_X, p\_Y, p\_Z)

{

console.log(p\_X);

console.log(p\_Y);

console.log(p\_Z);

return p\_X + p\_Y + p\_Z;

}

console.log(su\_mar(...num\_eros));//resultado de la suma 6

**Sintaxis de Desestructuración**

Esta sintaxis nos permite asignar las propiedades de un objeto a una variable que podemos usar en el programa.

Ej. tenemos un obj que representa un usuario con sus propiedades y elementos. Anteriormente si queríamos asignar las propiedades nombre y edad sus valores en la variables, lo que se hacía es tomar el obj usuario

*const vNombre = usuario.pNombre;*

*const vEdad = usuario.pEdad;*. Pero en el nuevo estándar podemos hacer eso mismo en una sola línea para cualquier número de propiedades que quisiéramos asignar. Lo primero que hacemos es escribir const seguido de un par de llaves y dentro escribimos los nombres de las variables que queremos crear y continuamos escribiendo precedido de un = el nombre del obj usuario. Esto lo que va hacer es buscar dentro del obj usuario y verifica si existe una propiedad nombre asique asigna el valor a la variable y también existe una propiedad edad de igual manera le asigna su valor. *const*

Esto es básicamente la sintaxis de desestructuración más simple que podemos usar pero hay varias opciones.

*{nombre, edadad} = usuario;*

const usuario = {

nombre: "Manuel Orozco",

edad: 66

};

const nombre = usuario.nombre;

const edad = usuario.edad;

const {nombre, edadad} = usuario;

**Vamos a ver otro ejemplo:**

Digamos que tenemos un objeto que representa unas coordenadas. Anterior mente cuando estaba ES5 con el viejo estándar tendríamos que hacer lo siguiente debajo de la llave de cierre del obj. coordenadas

var eje1 = coordenadas.eje1;

var eje2 = coordenadas.eje2;

var eje3 = coordenadas.eje3;

Nota: Las variables llevan el mismo nombre de la propiedad del obj

var coordenadas = {

x: 6,

y: 8,

z: 12

};

var {eje1, eje2, eje3} = coordenadas;

Pero ahora con la sintaxis de desestructuración podemos hacer lo siguiente *var {eje1, eje2, eje3} = coordenadas;*

Declaramos la variables entre las llaves que deberán tener el mismo nombre que las propiedades del obj, a continuación el signo de asignación→= y por último el nombre de obj→coordenadas cerrando con un→**;** y verificamos por consola

**Sintaxis de desestructuración: Objetos Anidados**

Vamos a crear un objeto que representa a un abonado ese obj va a contener una propiedad que representa el nombre del abonado→marioCruz que a su vez el nombre está asociado a un obj anidado con dos propiedades. edad cuyo valor es 35 y email cuyo valor es su dirección de correo. Digamos que queremos asignar la edad y el email a variables individuales en el programa para trabajar con ellas. A continuación veremos la ventaja de usar la sintaxis de desestructuración por que imagínate como seria acceder a los obj anidados de estructuras más complejas varias veces.

Con la sintaxis de desestructuración podemos asignar esas propiedades a variables y trabajar con ellas donde las necesitemos en el programa.

const abonado = {

marioCruz:{

edad: 35,

email: "marioC@gmail.com"

}

};

console.log(edad);

console.log(email);

const {marioCruz: {edad: edadaDelUsuario, email: correoDelUsuario}} = abonado;

Lo que queremos especificar dentro de las llaves es que propiedades vamos a extraer de ese obj. de abonado, de la propiedad marioCruz queremos extraer y asignar a sus variables correspondiente las propiedades edad y correo. Luego de esta linea→*const {marioCruz: {edad, email}} = abonado;* Y por último verificamos con console.log(). También otra alternativa importante para evitar confusiones ya que usamos variables con el mismo nombre de las propiedades. Hay una alternativa para lograr esto. Dentro de las llaves { } podemos asignarles a estas propiedades nombres distintos. Seguido a los valores colocamos→**:** y un nombre referencial y conveniente para tal fin.

*{edad: edadaDelUsuario, email: correoDelUsuario}* Siguiendo con el mismo tema de desestructuración: Objetos Anidados Digamos que tenemos una constante que va a representar el pronóstico local del clima.

const PRONOSTICO\_LOCAL = {

"ayer": {

minima: 32,

maxima: 40

},

"hoy": {

minima: 28,

maxima: 36

},

"mañana": {

minima: 29,

maxima: 38

}

};

/\*

Usando codigo tradicional en Objetos Anidados pero no tan conciso

const minimaHoy = PRONOSTICO\_LOCAL .hoy.minima;

const maximaHoy = PRONOSTICO\_LOCAL .hoy.maxima;

\*/

//Usando Sintaxis de desestructuración en Objetos Anidados en una misma linea

const {hoy: {minima: minimaHoy, maxima: maximaHoy}} = PRONOSTICO\_LOCAL;

console.log(minimaHoy, maximaHoy);

var mostrar = "Temperatura minma hoy " + minimaHoy + " Temperatura maxima hoy " + maximaHoy;

console.log(mostrar); //Usando una variable para concatenar la muestra

Veamos como con la sintaxis de desestructuración podemos acceder a la temperatura mínima y máxima de “hoy”. Vamos a declarar una constante para ver como accederíamos a los valores que necesitemos y asignarlos a una variable. Nótese que las propiedades de los obj. anidado no están entre comillas pero JS por defecto las usara porque son propiedades de una sola palabra. Sintaxis: const variable = nombre del obj principal con notación de punto nombre de la propiedad que contiene el obj. anidado seguido de un punto el valor. Mostramos por console.log(variable); Un truco interesante de console.log es que si separas por una con una coma las variables van aparecer en una misma línea. Bien ya sabemos cómo asignaríamos esos valores a variables. Pero vamos a ver cómo podemos hacer lo mismo con la sintaxis de desestructuración. Nuevamente usamos una constante seguidas de un par de llaves ahí escribimos la propiedad hoy: luego como tenemos un obj anidado nuevamente un par de llaves, queremos acceder a la propiedad mínima: que tenemos aquí pero queremos asignarle otro nombre para nuestra variable→minimaHoy cerramos la llave seguido del signo de asignación = escribimos el nombre de la constante del obj. principal→PRONOSTICO\_LOCAL cerrando con→ **;**

Cuando le asignas un nombre nuevo a la variable eso es lo único que puedes usar, el nombre de la propiedad no es una variable. const {hoy: {minima: minimaHoy} } = PRONOSTICO\_LOCAL; También puedes hacer ambas operaciones en la misma línea. const {hoy: {minima: minimaHoy, máxima: maximaHoy} } = PRONOSTICO\_LOCAL;

**Sintaxis de desestructuración: ARRAY**

La mencionada sintaxis también funciona con arrays y nos permite asignar valores usando array. Por ejemplo si tenemos dos array que el primero contiene dos variable [a,b] y el segundo [1,2] dos valores como asignamos esos valores a las variables. primero declaramos las variables vacias var a; var b; y y el array que contiene las variables se le asignan los valores del segundo array→[a,b]=[1,2]; Los valores se asignan en el mismo orden que aparecen en el arreglo y si tuviéremos una tercera variable se asignaría el tercer valor en el orden. var c; [a,b,c]=[1,2,3]; si nosotros tenemos un array con 5 valores pero un array con dos valores podemos asignar solo dos valores con la sintaxis de desestructuración [a,b]=[1,2,3,4,5] sin ningún inconveniente **a** vale 1 y **b** vale 2 pero que pasa si tenemos una tercera variable **c** y queremos asignarle el valor 5. De manera muy simple esto se logra agregando tantas comas para saltar los valores y llegar al valor deseado→[a,b,,, c]=[1,2,3,4,5] la primera coma corresponde a b la segunda y tercera lo que vendría s se 3 y 4 y el resultado es que c se asigna con 5. La sintaxis de desestructuración también puede usarse para cambiar o intercambiar los valores de dos variables. Si tenemos dos variables a = 8 y b = 6 podemos usar la sintaxis de desestructuración para intercambiar su valor. queremos que a tenga el valor 6 y b tenga el valor 8. Veamos cómo podemos hacerlo, simplemente escribimos un array con las variables a las cuales queremos asignar los valores [b, a] como notaras el orden está invertido por que en un próximo array [a, b] se asignaran los valores en el orden establecido por JS [b, a]=[a,b] de derecha a izquierda.

**Sintaxis de Desestructuración**

**con el Operador rest**

Recuerda que el operador ***rest*** formaba y agrupaba un array cuando lo usábamos con funciones agrupaba los argumentos. En este caso vamos a usar ese operador para resignar un array en varias variables. Primero declaramos las variables a y b y luego vamos a usar la sintaxis de desestructuración [a, b, ]=[1, 2] vemos que en el array uno después de b tenemos una coma seguido de un espacio porque después de esas dos variables vamos asignar el resto de todos los números que vengan después a un array nuevo que vamos a crear con el operador ***rest*** **…arr**→con una variable también la tenemos que declarar en nuestro códigoen la cual se van a asignar segundo array [1,2,3,4,5,6,7]

var varA;

var varB;

var arr;

[varA, varB,...arr] = [1, 2, 3, 4, 5, 6, 7];

console.log(varA);//1

console.log(varB);//2

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

const con\_sola = [varA, varB,...arr] = [1, 2, 3, 4, 5, 6, 7];

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

Veamos otro ejemplo: en el que vamos a usar este principio para remover los tres primeros elementos de un array inicialmente tenemos un array del 1 hasta el 8 y vamos a definir una función con un parámetro que va a remover los tres primeros elementos. En el cuerpo primero declaramos una variable que va a tener el nuevo arreglo que vamos a generar sin los tres primeros elementos. vamos a usar la sintaxis de desestructuración para array con la sintaxis de las comas que usamos para saltar lugares y luego si vamos asignar esos valores con los tres puntos→**…** a nuevoArreglo. En este caso como ya estamos declarando la variable→ var nuevoArreglo; no necesitamos escribir ***const*** pero si queremos una sola línea si escibimos ***const*** y luego la estructura, por eso vemos esa línea tachada. Y que vamos asignar a esta estructura? El parámetro→***arreglo*** que tomamos como argumento. Finalmente retornamos ese valor ***nuevoArreglo***, vamos a llamar a la función pasando el array inicial y mostraremos por consola el ***arregloFinal*** ya sin los tres primeros elementos

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

function removerTresPrimerosElementos(arreglo)

{

~~var nuevoArreglo; se elimina para usar const y la estructura va en una sola línea~~

const [ , , ,...nuevoArreglo] = arreglo;

return nuevoArreglo;

}

const arregloFinal = removerTresPrimerosElementos(arrgeloInicial);

console.log(arregloFinal)//muestra el array sin los tres primeros elementos

**Sintaxis de Desestructuración**

**Pasar obj. Como argumentos**

Esta sintaxis también se puede usar para pasar un obj. como argumento que representa el nuevo perfil de un cliente. Primero que todo necesitamos un objeto.

var nuevoPerfilCliente = {

nombre: "Sergei Lavrov",

edad: 74,

nacionalidad: "Rusia",

ubicacion: "Moscu"

};

Ahora como vamos a actualizar ese perfil, vamos hacerlo con una función flecha actualizarPerfil .

const actualizarPerfil = (informacionDePerfil) => {

const {nombre, edad, nacionalidad, ubicacion} = informacionDePerfil; aquí se aplica la sintaxis de desestructuración

console.log(nombre);

console.log(edad);

console.log(nacionalidad);

console.log(ubicacion);

var queRepresenta = informacionDePerfil;

console.log(queRepresenta);//respuesta: Un objeto

};

actualizarPerfil(nuevoPerfilCliente);

Esa función flecha va a tomar la información del Perfil del cliente y va actualizar el perfil. Pero en realidad no vamos a implementar la actualización del perfil, lo que vamos hacer es ver cómo podemos tomar este por ahora parametro→informacionDePerfil y descomponerlo cuando lo pasamos como un argumento. Si no lo descomponemos y simplemente escribimos este parámetro→ informacionDePerfil que va a ser un obj. podemos descomponerlo ya dentro de la funcion. Escribimos la sintaxis de desestructuración que ya aprendimos: const segido de las llaves y dentro las propiedades del obj→nuevoPerfilCliente precedido por un signo de asignación = escribimos el nombre del ahora objeto →informacionDePerfil y por ultimo con console.log podemos mostramos c/u de los valores de las propiedades del obj. pero falta algo muy importante.→ A continuación fuera de las llaves vamos a llamar a la función flecha y le vamos a pasar como argumento el obj→nuevoPerfilCliente→este objeto se va asignar a al parámetro de la función flecha→(informacionDePerfil) y luego si vamos a desglosar o descomponer sus propiedades que ahora son variables que podemos utilizar en la función por ser variables locales .

Pero en este caso estamos tomando el obj completo, si nosotros mostramos con console.log que representa informaciónDePerfil obtenemos un objeto

console.log(nombre);

console.log(edad);

console.log(nacionalidad);

console.log(ubicacion);

console.log(queRepresenta);//respuesta: Un objeto

Hay otra forma con la sintaxis de desestructuración de descomponer un objeto directamente en la lista de parámetros para asignarlos a variables individuales. Es muy práctico, const actualizarPerfil = (~~informacionDePerfil~~) => {} en lugar del parámetro actual escribiremos ahí un par de llaves ({}) y vamos a escribir el nombre de las propiedad que queremos extraer, en este caso tenemos 4 propiedades ({nombre, edad,nacionalidad,ubicacion}) asi podemos de una vez directamente asignar los valores a actividades independientes a variables que podemos usar en nuestra función y así el cuerpo de la función es más conciso. No necesariamente tenemos que tomar todos los valore como argumentos. Por ejemplo simplemente tomar el nombre y la edad, esto también nos sirve para seleccionar propiedades específicas para la función del obj que vamos a tomar como argumento.

const actualizarPerfil = ({nombre, edad, nacionalidad, ubicacion}) => {

console.log(nombre);

console.log(edad);

console.log(nacionalidad);

console.log(ubicacion);

};

actualizarPerfil(nuevoPerfilCliente);

Ahora vamos a ver otro ejercicio. Como pasar un obj con la sintaxis de desestructuración:

Definimos un obj que se llame estadísticas con sus propiedades y valores.

const estadisticas = {

max: 56.78,

desviacionEstandar: 4.34,

mediana: 34.54,

moda: 23.87,

min: -0.75,

promedio: 35.85

};

Ese obj. lo podemos pasar a una función flecha, digamos que inicialmente tenemos esta función =>flecha que calcula el punto medio del intervalo entre el máximo y el mínimo tomaría un parámetro que se llama estadísticas que no tiene nada que ver con el nombre del obj. Y que vamos hacer con esas estadisticas? Vamos a tomar el max y el minimo lo vamos a sumar para luego dividirlo en 2.0. Eso es lo que va hacer la función =>flecha.

const mitad = (p\_E) => (p\_E.max + p\_E.min) / 2.0;

Pero como podemos reescribir esta función usando la sintaxis de desestructuración. Lo primero que tenemos que analizar es que propiedades de ese obj vamos a necesitar en la función, max y min en este caso asique simplemente escribimos esas propiedades en la lista de parámetros rodeadas por llaves ({max, min}) y como ya son variables con la que podemos trabajar en nuestra función, remplazamos esto => (max + min) / 2.0; Y ahora la sintaxis es la siguiente

const mitad = (max, min) => (max + min) / 2.0; Esto va asgnar los parámetros a sus variables correspondiente. Nota: el paréntesis a la izquierda de => son variables y a la derecha son parámetros. Llamamos a la función mitad pasando el objeto estadística console.log(mitad(estadistica));

**//const mitad = (p\_E) => (p\_E.max + p\_E.min) / 2.0;**

**const mitad = ({max, min}) => (max + min) / 2.0;//Esto remplaza el comentario de lo de arriba**

**console.log(mitad(estadisticas));//llamamos a la funcion=> mitad pasando el obj estadistica y**

**// el resultado promedio es 28.015**

**Plantillas Literales:**

Ahora vamos hablar de un tipo especial de cadena de caracteres que nos permite crear cadenas con varias líneas y remplazar los valores de las variables que necesitemos fácilmente en una cadena.

Plantillas literales o plantillas de cadenas

Características:

- Se usa el acento invertido →backtick este símbolo  **‘** en lugar de comillas.

- Pueden contener comillas simples y dobles.

- Las líneas se preservan como se escriben en el código.

- Para remplazar una variable se escribe ${variable}.

- Dentro de ${} también puedes escribir expresiones. \* variables x 2 o sumar un valor el resulta se va a remplazar en la c

Veamos algunos ejemplos: Si definimos una variable con un valor inicial 6 var a = 6 y mostramos una plantilla de cadena nótese que lo primero que usamos es backtick console.log(‘El sobrino cumple ${a} ’) normalmente hubiésemos concatenado la cadena con la variable→console.log(“El sobrino cumple ” + a); pero con las plantillas de cadenas o plantillas literales ya es mucho más fácil remplazar una variable.

var pL = 6

console.log(`El sobrino cumple ${pL}`);//backtick cadena y remplazo de variable

Veamos otro ejemplo con una cadena de caracteres:

//Ej1

var pL = 6;

console.log(`El sobrino cumple ${pL}`);//backtick cadena y remplazo de variable

//Ej2

var \_nombre = "Nora";

var \_edad = 15;

console.log(`Mi nombre es ${\_nombre} y cumplo ${\_edad} años`);

También podemos remplazar otros tipos de datos, por ejemplos array .

//EJ3

var otroArray = [1,2,3,4,5];

console.log(`El array es ${JSON.stringify(otroArray)}`);

Si queremos mostrar el array de una forma más presentable podemos usar este método JSON.stringify cuyo resultado se mostra envuelto en []. También podemos remplazar el valor de las propiedades de un obj.

//Ej4

var persona = {

alumno:"Daniel Muños",

edad: 12

};

Ahora tenemos un obj en lugar de variables individuales o variables separadas. Con un obj también podemos cambiar una plantilla literal . Vamos a ver otra variación, en lugar de escribir la plantilla literal directamente en consol.log podemos asígnale a una variable const SALUDO = `` seguido del = y backtick, en este caso actúa como una constante por eso la escribimos en mayúscula, bien ya creamos nuestra plantilla que va a ser asignada a nuestra variable. Como lo hacemos? Con notación de punto `Hola mi nombre es ${persona.alumno}` el nombre de la variable que contiene el obj seguido de la propiedad de igual forma acedemos a edad ${persona.edad}

**Crear Objeto de Forma Concisa:**

Declaramos una constante que se le va a signar una función => que se llama crearPersona que contiene tres parámetros esa función va a retornar un obj. Esto es una forma de crear objeto en nuestro programa. Cuando llames a la función nos va a retornar un obj. con los valores de los parámetros, esta va a ser la estructura del objeto

const crearPersona = (p\_Nombre, p\_Edad, p\_Idioma) =>{

return {

p\_Nombre: p\_Nombre,

p\_Edad: p\_Edad,

p\_Idioma: p\_Idioma

};

};

console.log(crearPersona("Leonel Messi", 36, "Español"));

El resultado veremos el obj. { p\_Nombre: 'Leonel Messi', p\_Edad: 36, p\_Idioma: 'Español' }

Pero veremos una forma más concisa de escribir esta función flecha en una sola línea. Lo que podemos hacer es simplemente luego de la flecha escribir el obj. rodeado por paréntesis ({}) dentro escribimos llaves que le van a decir a JS que queremos retornar un obj. y cuál va a ser la estructura del obj? va a contener los valores de los parámetros de crearPersona (p\_Nombre, p\_Edad, p\_Idioma). Esto es básicamente lo que le estamos diciendo a JS rea un obj con esta estructura, las propiedades van a tener los mismos nombres que los parámetros y sus valores se van a remplazar cuando se cree el objeto. El resultados es exactamente el mismo

//Esto remplaza al ejercicio anterior de arriba, crear Objeto de Forma Concisa obteniendo el mismo resultado

const crearPersona = (p\_Nombre, p\_Edad, p\_Idioma) =>({ p\_Nombre, p\_Edad, p\_Idioma});

console.log(crearPersona("Leonel Messi", 36, "Español"));

{ p\_Nombre: 'Leonel Messi', p\_Edad: 36, p\_Idioma: 'Español' }

**Métodos:**

También puedes declarar dentro de los obj. funciones que crees en tus programas de una forma más concisa. Declaramos una constante que va a contener un obj. Este obj. va a tener una función asociada esa es parte de la versatilidad de los objetos. Podemos definir propiedades con sus correspondientes valores y propiedades que sean funciones. Digamos que queremos que Isabel pueda presentarse→ese parámetro va ser una función que vamos a poder llamar a esta función a través del obj. escribimos paréntesis vacíos porque en este caso no va a tomar ningún parámetro y luego llaves donde escribimos el cuerpo de la función. Digamos que queremos retornar una plantilla literal con una cadena de caracteres remplazando el ***nombre\_*** de la propiedad con su valor ***“Isabel”*** pero anteponiendo mediante un **.** la palabra→ ***this*** esta palabra cumple una función muy especial en JS porque se refiere al objeto con el cual estamos trabajando. En este caso ***this.*** se refiere al obj **\_persona.** En resumen: En este caso ***this*** se refiere al obj ***persona*** a la propieda→*nombre\_* que su valor es ***“Isabel”*** por ultimo llamamos a la función de la siguiente forma. Curiosamente vamos a poder usar el nombre de lapropiedad→***presentarse*** como si fuera el nombre de la función con paréntesis vacíos ya que la función en si no se le asigno parámetro, pero como estamo retornando usamos console.log como se muestra en el ejercicio.

Nota: *Si el valor de una propiedad es una función se la denomina método*

const \_Persona = {

nombre\_: "Isabel",

//presentarse: function () {esto se remplaza por el código de abajo

presentarse () {

return `Hola mi nombres ${this.nombre\_}`;

}

};

console.log(persona.presentarse());

Esta es la sintaxis que hubiésemos usado anteriormente pero con el estándar nuevo ES6 podemos simplificar nuestro código aún más para definir esta función. Simplemente quitamos los dos puntos y la palabra clave function que visualmente parecería que la propiedad sería una función.

**Definir una Clase**

Ahora vamos a ver cómo puedes definir una clase en JS . Una clase en JS es algo parecido a un plano de un edificio o de un objeto. Que nos permite crear muchos obj con la misma estructura, la misma propiedad y la misma funcionalidad, es un plano general que nos permite usar código una sola vez y reusarlo para crear tantos obj como necesitemos en nuestro programa. Veamos un ejemplo: Luego de la palabra clave ***class*** por convención escribimos el nombre comenzando con mayúscula el nombre del tipo de obj que vamos a crear. Digamos que queremos representar un TrasbordadorEspacial en nuestro código que va a tener ciertas propiedades. Va a tener un estado inicial que lo podemos inicializar con una función→***constructor()*** asociada al obj que vamos a definir {} Es la función que va a llamar al obj creado con ***class→TrasbordadorEspacial {}***. En la función asosiada→***constructor()*** que no es ni más ni menos que una propiedad le agregamos un parámetro(***planeta***) que va a llamar automáticamente a → ***TrasbordadorEspacial*** y va a tomar el parámetro que nosotros definimos→(***planeta***) De modo que de esta forma sabremos el nombre de ese planeta que definimos. A modo de ejemplo júpiter con trasbordador espacial y lo vamos asignar como una propiedad ese planeta, va a ser una propiedad del obj planeta. Recordemos que ***this*** cumple una función muy especial en JS porque se refiere al objeto con el cual estamos trabajando. Con esa sintaxis→this.planeta = planeta o sea que el objeto trasbordadorEspacial va a tener una propiedad llamada planeta. En resumen cuando escribimos ***this*** nos estamos refiriendo al objeto que estamos creando con class. Básicamente estamos diciendo que vamos asignar el valor del parametro a la propiedad planeta. Para que se entienda planeta1 pasa a ser planeta2 y asigna a planeta3

Algo que debes saber que en realidad cuando usamos la palabra clave class aquí debes tener en cuenta que JS en realidad no está creando una clase completamente funciona como lo haría JAVA o PYTHON u otros lenguajes, sino que simplemente estamos escribiendo en forma más concisa una forma distinta de crear un obj. No tiene todas las capacidades de una clase como en otros lenguajes pero si nos permite crear obj en nuestro programa

Si queremos crear un trasbordador que se llame Zeus, vamos a definir una variable y vamos a declarar un nuevo obj con la palabra clave→new seguido del nombre de la clase TrasbordadorEspacial y entre paréntesis los elementos que requiera el constructor en este caso es el planeta de destino→Jupiter en este caso para este obj en particular.

Ejercicio↓

class TransbordadorEspacial {

constructor(planeta) {

this.planeta = planeta;

}

}

var zeus = new TransbordadorEspacial("Jupiter");

console.log(zeus.planeta);//Resultado Jupiter

var apolo = new TransbordadorEspacial("Marte");

console.log(apolo.planeta);//Resultado Marte

//Otro ejemplo

class Mascota{

constructor(nombre, edad);

};

Asique estamos creando un obj que va a tener una propiedad planeta, esa propiedad va a tener como valor jupiter. de bajo podemos ver como accedemos a la propiedad planeta del obj Zeus y en otro ejemplo apolo destino Marte

var zeus = new TransbordadorEspacial("Jupiter");

console.log(zeus.planeta);//Resultado Jupiter

var apolo = new TransbordadorEspacial("Marte");

console.log(apolo.planeta);//Resultado Marte

En otro ejemplo definimos una class que represente a mi mascota puedo hacerlo con la función constructor() que le asignamos dos parámetros, nombre y edad. Vamos asignar una propiedad llamada nombre para el obj y lo mismo hacemos con la edad. Así cuando creemos una mascota por ejemplo var miMascota = new va a ser un nuevo obj que va a tener por nombre “Lala” su edad es 5 var miMascota = new Mascota(nombre, edad); y ya después de esta línea podemos trabajar con el obj que va a tener una propiedad llamada nombre y otra edad a las cuales podemos acceder, actualizar, eliminar y con las cuales trabajar en general con todas las operaciones que aprendiste durante el curso.

Podemos reutilizar este código de la class cuantas veces necesitemos

class Mascota{

constructor(nombre, edad){

this.nombre = nombre;

this.edad = edad;

}

}

var miMascota = new Mascota(“Lala”, 17);

console.log(miMascota.nombre);

console.log(miMascota.edad);

//reutilizamos el código para otra mascota

var otraMascota = new Mascota(Camilo ,4);

**Getters y Setters**

Ahora expandamos nuestro concepto de un obj, ya sabes que los obj pueden tener funciones asociadas a esos obj, también podemos definir funciones en una class para que cuando creemos un obj ese obj tenga todas esas funciones asociadas. Pero hay dos funciones específicas que se llaman Getters y Setters que nos van a permitir proteger la data de ese obj. Vanos a declarar una class llamada libro, esa class va a tener un constructor el libro va a tener un autor y vamos a crear una propiedad llamada autor. Pero asi como la estamos definiendo en este momento podemos cambiar el valor del autor en cualquier momento y a cualquier valor. Eso podría ser una fuente de errores si lo cambiamos a un valor que no es válido por ejemplo que el autor sea un número o a un autor que no corresponde a ese libro, por eso en JS podemos proteger esa data asiendo que nuestras variables sean privadas por convención agregando un guion bajo \_variable, esto le dice a otros desarrolladores que no deberían modificar ese valor. Solo se debe usar dentro de la class, en la práctica no hace que técnicamente que la propiedad sea privada. Pero podemos dar acceso a esa propiedad, con un Getters y Setters. Los getters son métodos que nos permiten en obtener un valor de una propiedad privada, una propiedad que tiene un guion bajo antes de su nombre. Nos permite acceder a ese valor en forma indirecta asi protejemos el valor; no lo estamos cambiando directamente. Como definimos un getters? get→significa obtener ese método su propósito es retornar ese valor para que podamos usarlo en nuestro programa en forma indirecta. escribimos ***get*** el nombre de la propiedad y paréntesis vacíos seguido de llaves luego retornamos ese valor→ \_autor. Como se ve en el ejercicio solamente estamos retornando ese valor de la propiedad del obj. cuando accedamos a la propiedad vamos acceder a ese valor pero está protegido por que tiene un intermediario durante el proceso, este nos permite obtener un valor. Pero también podemos definir unsetters→***set*** que es una función que nos permite actualizar el valor de una propiedad y son muy útiles porque podemos verificar el valor nuevo antes de asignarlo. Si no usáramos un setters entonces podríamos cambiar ese valor→\_autor a cualquier valor aunque no sea valido. pero con un settes podemos usar condicionale para verificar si es válido. En este ejemplo simplemente vamos asignar un nuevo autor. Tomamos el valor del parámetro→nuevoAutor y lo asignamos a la propiedad→\_autor y así tenemos nuestro getters y nuestro setters. Vamos a probarlo creando un obj nuevo de tipo libro→const libro = new Libro("Anónimo"); y en la consola vemos→Anónimo exactamente lo que especificamos aquí→new Libro("Anónimo"); Puedes ver que aunque en realidad no tenemos una propiedad llamada autor si no que la propiedad se llama \_autor con un guion bajo adelante aun asi podemos acceder a la propiedad simplemente escribiendo autor. Eso es gracias a que definimos un getters y le asignamos ese nombre ***autor*** y ese valor se obtiene indirectamente de ***\_autor*** a través del getters. Podemos confirmarlos si en el get agregamos console.log(Getters); primero vemos el mensaje “Getters” antes de ratonar el valor Anónimo. Ahora vamos a probar cómo funciona el Setters, si queremos actualizar el valor de la propiedad autor podemos hacerlo también directamente y gracias al Setters se va actualizar como puedes ver actúan como intermediarios. Nuevamente luego de la actualización vamos a mostrar el valor del autor

class Libro {

constructor(autor){

this.autor = autor;

}

get autor() {

console.log("Getters");

return this.\_autor;

}

set autor(nuevoAutor){

console.log("Setters");

this.\_autor = nuevoAutor;

}

}

const libro = new Libro("Anónimo");

console.log(libro.autor);

**FIN del CURSO**

\*/