JAVASCRIPT

¿Qué es Java Script?

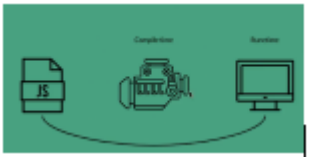
Para saber que es JS primero debemos ver de dónde viene. javascript nace de la necesidad de darle dinamismo a las páginas web en su momento.

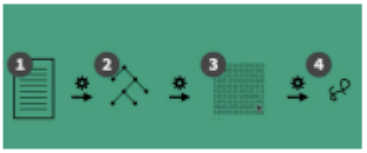
Como lenguaje es, es un lenguaje interpretado, orientado a objetos, débilmente tipado y dinámico.

- Débilmente tipado se refiere a que no es tan exigente en cuanto algunas reglas y no es tan sensible a errores y los puede interpretar de otra forma

```
4 + "7";  // 47
4 * "7";  // 28
2 + true;  // 3
false - 3;  // -3
```

- Dinámico se refiere a que no es necesario que sea compilado o en otras palabras traducido para que lo entienda el dispositivo, puedo ir haciendo pruebas fácilmente sin necesidad de esperar que se tenga que compilar, sin embargo mi código si pasa por una interpretación por el motor de js





Backwards y Forwards

Js es Backwards compatible quiere decir que todas las versiones nuevas no van a romper tu código sin embargo no vas a poder usarlas en dicho proyecto, pero para poder usarlos podemos utilizar herramientas compiladores como BABEL Que traducen el código de nuevas versiones a versiones estándares que entiende el navegador

¿Por qué Javascript?

Si bien sabemos que js es un lenguaje muy dinámico, también es una lenguaje muy versátil ya que nos permite desarrollar productos para la web, aplicaciones móviles, de escritorio y además trabajar con el backend.



Javascript cuenta con dos componentes principales que son:

- La información (DATA) que guardas en memoria
- Las funciones con las que manipulamos esa DATA

VALORES

Valores Primitivos

Para los valores en js existen de cuatro tipos:

- Números comunes
- Strings que son cadenas de texto y se guardan dentro de comillas
- Booleanos, que son los valores TRUE y FALSE que significan 0 y 1
- También existen empty values que son espacios varios reservados para otro valor.

Valores No Primitivos o valores de tipo Objetos

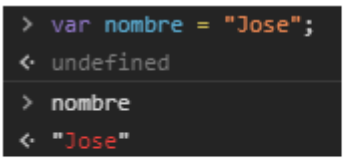
- Tipo Array [ ]
- Tipo objeto { }

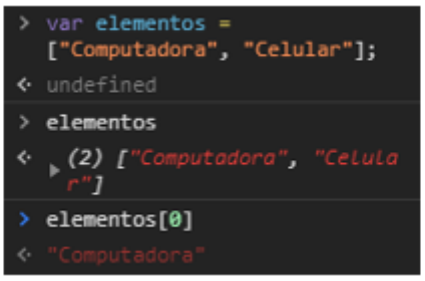
Existe una palabra clave reservada en js que me dice el tipo de dato que estoy ingresando que se llama "TYPEOF"

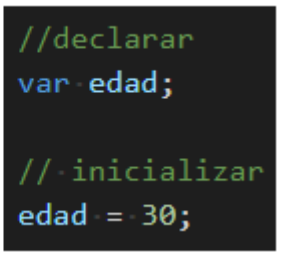


VARIABLES

Es la representación de algún lugar en memoria que vamos a reservar para guardar un valor, estas se guardan con la palabra VAR, podemos guardar valores de todo tipo como arrays, de tipo objeto, o simplemente strings o números.

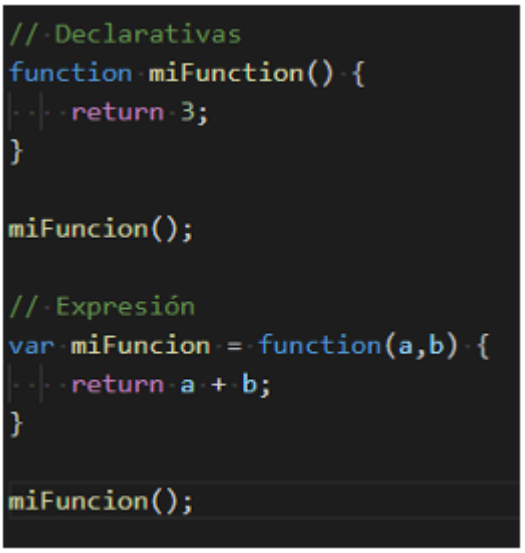






FUNCIONES

Son un conjunto de sentencias que usamos para generar ciertas acciones con los datos que guardamos previamente.

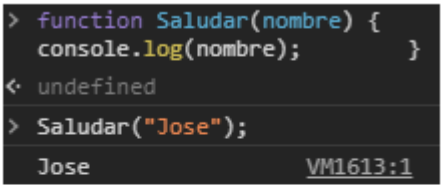


```
// Declarativas
function miFunction() {
    return 3;
}

miFuncion();

// Expresión
var miFuncion = function(a,b) {
    return a + b;
}

miFuncion();
```



```
> function Saludar(nombre) {
  console.log(nombre);        }
← undefined
> Saludar("Jose");
  Jose                        VM1613:1
```

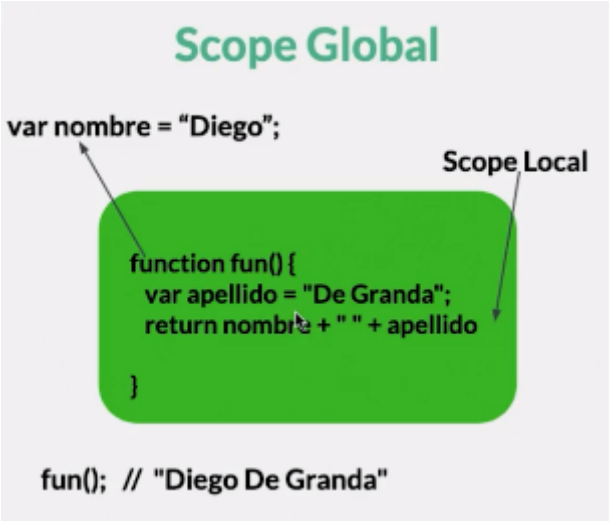
Existen dos tipos de funciones, funciones declarativas y de expresión.

La diferencia entre estos dos tipos de funciones es que a las funciones declarativas se les aplica hoisting, y a la expresión de función, no. Ya que el hoisting solo se aplica en las palabras reservadas var y function.
Lo que quiere decir que con las funciones declarativas, podemos mandar llamar la función antes de que ésta sea declarada, y con la expresión de función, no tendríamos que declararla primero, y después mandarla llamar.
Otra diferencia es que las funciones de expresión son anónimas y se llaman con el nombre de la variable donde fueron guardadas.

NOTA: Para regresar valores de una función puedo usar la palabra reservada RETURN o puedo decirle que me mande el resultado a la consola con CONSOLE.LOG

SCOPE

El Scope es el alcance que tienen las variables ya que depende de donde declares o inicialices una variable si vas a tener acceso a ella o no



Con mi scope local puedo acceder a lo que esté en el global pero no al revés, quiere decir que si quiero usar una variable declarada en una función por fuera de ella no la voy a poder usar

HOISTING

Es cuando las variables y las funciones se declaran antes de que se procese cualquier tipo de código, sin embargo el hoisting solo ocurre en versiones pasadas, ya que en la versión ECMA SCRIPT 6 en adelante ya no sucede esto, como conclusión debemos declarar todas nuestras variables al principio del código así no tendremos este tipo de errores

```
> hey();

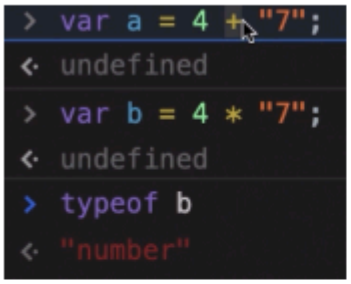
  function hey() {
      console.log("Hola " miNombre);
  }

  var miNombre = "Diego"
✖ Uncaught SyntaxError: missing ) after argument list
```

COERCIÓN

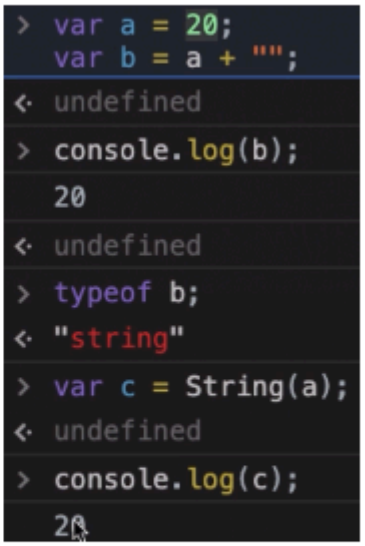
Coerción es la forma en que podemos cambiar un valor de un tipo a otro existe de forma implícita es cuando el lenguaje nos ayuda cambiándolo automáticamente y de forma explícita es cuando forzamos a que cambie el tipo de dato.

IMPLICITA



En este ejemplo el lenguaje Automáticamente me cambia El tipo de valor.

EXPLICITA



Aquí de forma explícita gracias a la propiedad string cambia el valor de un número a una cadena de texto

VALORES: TRUTHY AND FALSY

Existen cierto tipo de valores que son verdaderos y falsos.

Valores falsos

- Cuando no hay un valor definido
- El numero 0
- El valor null
- El valor NAN
- El valor Undefined
- La palabra false
- Un string vacío ""

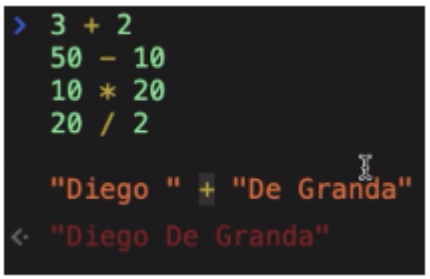
Valores verdaderos

- Un valor definido dentro de un string
- Un número arriba del 1
- Un array incluso si está vacío
- Un objeto incluso si está vacío
- Cualquier tipo de función
- La palabra true

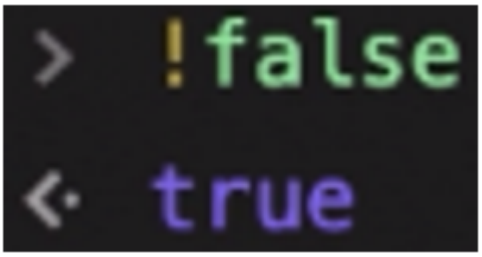
Todo esto lo puedo probar con la función BOOLEAN()

OPERADORES: ASIGNACIÓN, COMPARACIÓN Y ARITMÉTICOS

Operadores binarios:



Operadores unitarios:



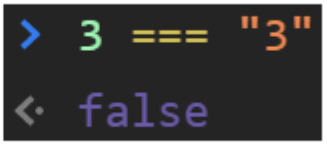
Operadores de asignación:

El signo = me asigna un Valor a una variable



Operadores de comparación:

El signo == me compara que dos valores sean iguales sin importar el tipo de dato



Sin embargo el valor === estrictamente me Compara que dos valores además de ser el mismo tiene que ser del mismo tipo de dato

Otro tipo de valores:

&& = y si a y b se cumplen entonces es verdad

Con el operador ++ puedo sumar 1

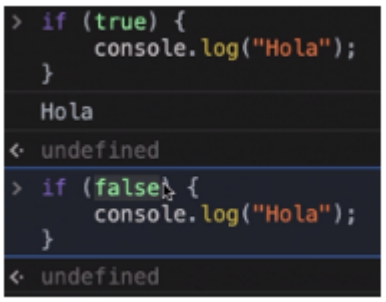
|| = o si a o b se cumplen entonces es verdad

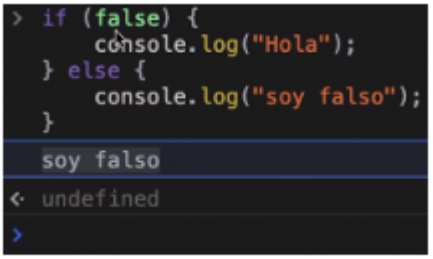
CONDICIONALES

IF/ELSE

Me valida si una sentencia es verdadera si es así ejecuta el código de lo contrario no
Sin embargo si añado ELSE cuando no se cumpla la primera sentencia se ejecutará el código que exista en el ELSE

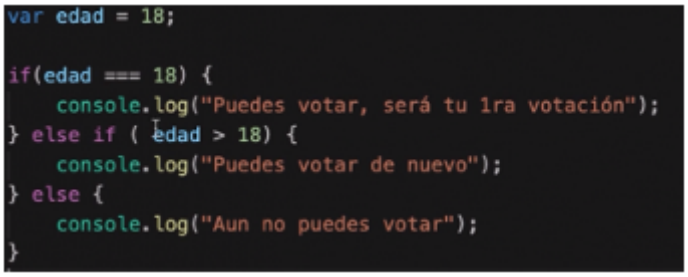
IF ("Sentencia") {Código que se ejecuta}





ELSE IF

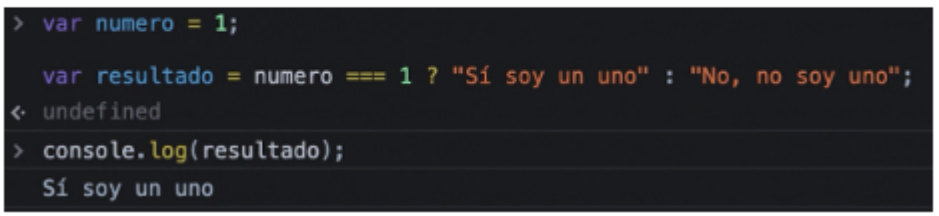
Sin embargo si quiero validar más de una valor y me valide varias sentencias puedo usar ELSE IF



CONDITION

Existe un operador ternario que sería lo mismo que un If y else juntos esta es su sintaxis:

```
condition ? true : false;
```



SWITCH

Switch es otra forma de validar o crear una condición y su sintaxis es la siguiente:

ELSE ("Sentencia") {Código que se ejecuta}

```
> var numero = 1;

  switch (numero) {
    case 1:
      console.log("Soy uno!");
      break;

    case 10:
      console.log("Soy un diez!");
      break;

    default:
      console.log("No soy nada");
  }
  Soy uno!
```

ARRAYS

Es una estructura de datos, un tipo objeto es un valor que va a guardar más valores dentro que pueden ser de todo tipo, esto es estructura de datos
Sintaxis : VAR = VARIABLE [];

```
var frutas = ["Manzana", "Plátano", "Cereza", "Fresa"];

console.log(frutas);
```

Yo puedo conocer el tamaño de un array, saber cuantos valores guardados tienen con la propiedad LENGTH

```
> console.log(frutas.length);
  4
```

MÉTODOS PARA MODIFICAR UN ARRAY

Para modificar el contenido de un array tenemos algunos métodos que nos sirven para ello.

- Para agregar nuevos objetos a mi array en la última posición puedo usar la propiedad "array.push"

```
var masFrutas =
frutas.push("Papaya");
```

- Para eliminar objetos que estén ubicados en la primera posición podemos usar la propiedad "array.shift"

```
var borrarFruta =
frutas.shift("Peras");
```

- Para eliminar objetos en la última posición de mi array puedo usar la propiedad "array.pop"

```
var menosFrutas =
frutas.pop("Fresas");
```

- Para agregar objetos a mi array pero en la primera posición puedo usar la propiedad "array.unshift"

```
var nuevoValordePrimero =
frutas.unshift("Peras");
```

Cuando quiero traerme un objeto de un array en el caso que fuese muy grande y no supiera el índex de dicho objeto pero sé el nombre del objeto, lo puedo buscar con la propiedad Array.indexOf

```
> console.log(frutas);
                                    VM2520:1
  (5) ["Manzana", "Plátano", "Ce
  reza", "Fresa", "Uvas"]
< undefined
> frutas.indexOf("Cereza")
< 2
>
```

LOOPS / CICLOS

Los ciclos nos permiten repetir una operación/acción/tarea siempre y cuando la condición la cual ejecuta el ciclo se cumpla, existen varios tipos de ciclos

FOR

Es un ciclo muy útil que me permite recorrer arrays y su sintaxis es:

```
for (let index = 0; index < array.length; index++) {
  const element = array[index];
```

EJEMPLO:

```
> var estudiantes = ["Maria", "Sergio", "Rosa", "Daniel"];

  function saludarEstudiantes(estudiante){
      console.log(`Hola, ${estudiante}`);
  }

  for(var i = 0; i < estudiantes.length; i++){
      saludarEstudiantes(estudiantes[i]);
  }

  Hola, Maria
  Hola, Sergio
  Hola, Rosa
  Hola, Daniel
```

FOR OF

Me permite también recorrer arrays y su sintaxis es la siguiente:

```
for (const iterator of object) {
  ..
}
```

EJEMPLO:

```
> var estudiantes = ["Maria", "Sergio", "Rosa", "Daniel"];

  function saludarEstudiantes(estudiante){
      console.log(`Hola, ${estudiante}`);
  }

  for(var estudiante of estudiantes){
      saludarEstudiantes(estudiante);
  }

  Hola, Maria
  Hola, Sergio
  Hola, Rosa
  Hola, Daniel
```

CICLO WHILE

El ciclo while me permite que mientras se cumpla la condicion que estoy declarando va a ejecutarse algo que yo defina, su sintaxis es:

```
while (condition) {
  ..|
}
```

EJEMPLO:

```
var estudiantes = ["Maria", "Sergio", "Rosa", "Daniel"];

function saludarEstudiantes(estudiante){
    console.log(`Hola, ${estudiante}`);
}

while(estudiantes.length > 0){
    var estudiante = estudiantes.shift();
    saludarEstudiantes(estudiante);
}
```

OBJETOS

Es una variable que puede tener muchos datos / propiedades de dicha variable guardadas en sí misma.

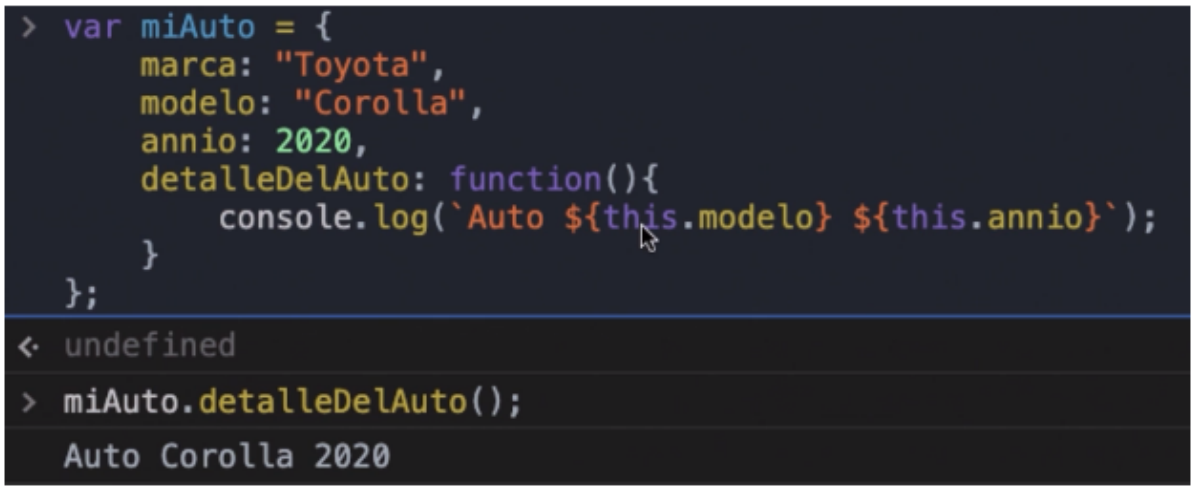
```
> var miAuto = {
      marca: "Toyota",
      modelo: "Corolla",
      annio: 2020
  };
< undefined
> miAuto.marca
< "Toyota"
> miAuto.annio
< 2020
```

Esto es un ejemplo de la sintaxis que utilizan los objetos, yo puedo mandar a llamar una propiedad específica de dicho objeto escribiendo "objeto.eldatoespecifico", sin embargo puede también guardar variables que usen datos del mismo objeto como otra característica de dicho objeto.

Las funciones que están guardadas dentro de un objeto son llamadas MÉTODOS, los métodos puedes usar valores que están dentro de su objeto padre con la palabra THIS, cuando yo escribo this me estoy refiriendo a mi objeto padre para acceder a sus datos.

NOTA: this funciona de esta forma solo en el concepto de objetos :D

EJEMPLO DE MÉTODO:



FUNCIONES CONSTRUCTORAS

Cuando quiero crear muchos objetos o una colección de ellos sería un poco tedioso tener que crear objeto por objeto así que para ello puedo crear una función en la cual cree un tipo de template o plantilla en la cual yo le paso los parámetros que quiero asignarle a mi nuevo objeto estos objetos van a ser llamadas instancias de mi función constructora



MÉTODOS DE RECORRIDOS DE ARRAYS

Estos métodos de recorrido me permiten obtener propiedades específicas que se encuentren dentro de nuestros arrays.

Para ejemplificar vamos a trabajar con este array:

```
var articulos = [
    { nombre: "Bici", costo: 3000 },
    { nombre: "Tv", costo: 2500 },
    { nombre: "Libro", costo: 320 },
    { nombre: "Celular", costo: 10000 },
    { nombre: "Laptop", costo: 20000 },
    { nombre: "Teclado", costo: 500 },
    { nombre: "Audifonos", costo: 1700 },
  ];
```

Array.filter();

Este método me ayudará a filtrar ciertos cosas validando si algo es verdadero o falso y metiéndoselo en un nuevo array sin sacarme datos de mi array original

Sintaxis:

```
> var articulosFiltrados = articulos.filter(function(){});
```

Ejemplo:

```
var articulosFiltrados =
articulos.filter(function (articulo) {
  return articulo.costo <= 500;
});
undefined
articulosFiltrados
▼ (2) [{…}, {…}]
  ▼ 0:
      costo: 320
      nombre: "Libro"
    ▶ __proto__: Object
  ▼ 1:
      costo: 500
      nombre: "Teclado"
    ▶ __proto__: Object
    length: 2
  ▶ __proto__: Array(0)
```

Array.map();

Esta función me ayuda a mapear ciertos artículos de un array, este método tampoco modificará mi array.

Ejemplo:

```
var nombreArticulos =
articulos.map(function (articulo) {
  return articulo.nombre;
});
undefined
nombreArticulos
▶ (7) ["Bici", "Tv", "Libro", "Celula
  r", "Laptop", "Teclado", "Audifonos"]
```

.FIND()

Este me validará me buscará dentro de mi array algo que yo le pida, si existe dicho objeto de lo trae como un nuevo array

Ejemplo:

```
> var encontrarArticulo = articulos.find(function(articulo) {
    return articulo.nombre === "Laptop"
  });
< undefined
> encontrarArticulo
< ▶ {nombre: "Laptop", costo: 20000}
```

.FOREACH()

Este método no me genera ningún array nuevo solo me da unos valores sin modificarme la función la cual estoy recorriendo

Ejemplo:

```
articulos.forEach(function(articulo) {
    console.log(articulo.nombre);
});
Bici
Tv
Libro
Celular
Laptop
Teclado
Audifonos
```

.SOME()

Este método me regresa la validación (True or False) que cumplen con la validación que le pido y me crea un nuevo array con dicha validación

Ejemplo:

```
var articulosBaratos = articulos.some(function (articulo) {
  return articulo.costo <= 700;
});
undefined
articulosBaratos
true
```

En el caso que no existiera un valor mayor o igual a 700 lo que retornaría seria false.