

Asi se define un arreglo

```
//TODO formas de definir un array
//! let autos = new Array ('bmw','mercedes benz', 'volvo');
// Esto no se usa

// asi se define un arreglos
const autos = ['bmw','mercedes benz', 'volvo'];
console.log(autos); [ 'bmw', 'mercedes benz', 'volvo' ]
```

Asi se acceden

```
console.log (autos[0]);|  bmw
```

Asi se recorre el array

```
for (let i = 0; i < autos.length; i++) {
  console.log(i + ' ' + autos[i]);  0 bmw, 1 mercedes benz, 2 volvo
}
```

```
// acceder a los datos y modificarlos
```

```
autos[1]= 'Mercedes-Benz';
```

```
console.log(autos[1])| Mercedes-Benz
```

```
// agregar valores a un arreglo, y se agrega al final
autos.push('Audi');
```

Otras formas de agregar valores a un arreglo

```
//! Cuidado FORMAS de agregar elementos al vector
// esto es para conocer la cantidad de elementos del arreglo
console.log(autos.length); 4

autos[autos.length] = 'Cadillac';
console.log(autos); [ 'bmw', 'Mercedes-Benz', 'volvo', 'Audi', 'Cadillac' ]

autos[6]= 'Ford';
console.log(autos); [ 'bmw', 'Mercedes-Benz', 'volvo', 'Audi', 'Cadillac', , 'Ford' ]
```

Como puedes ver, el mismo agrega un espacio en blanco y luego la posición que se indico

Preguntar si es un array

```
console.log(Array.isArray(autos)); true
console.log(autos instanceof Array); true
```

Funciones:

```
// declaracion de la funcion puede ir con o sin argumentos
function miFuncion(a, b){
  console.log('Suma: '+(a+b)); Suma: 5
}

//Llamado a la funcion
miFuncion(2,3);
```

```
// funciones de tipo expresion.
let sumar= function(a,b){return a + b};

resultado = sumar(1,2);
console.log(resultado); 3
```

```
// funciones tipo self o que se llaman por si sola

(function (){
  console.log('Ejecutando la funcion'); Ejecutando la funcion
})();
```

No es reutilizable, si es operable si le doy parámetros pero no se puede volver a utilizar en otra parte ya que no tiene ninguna variable anexada.

**Funciones como objetos:**

```
function miFuncion(a, b){
  console.log(arguments.length); 2
  return a + b;
}
```

Como vemos esta función tiene argumentos

Y también es tratada para convertirla a texto como lo haríamos con una clase:

```
var miFuncionTexto = miFuncion.toString();
console.log(miFuncionTexto); ... f(1); $ _
```

```
function miFuncion(a, b) {
  var _$c = _$wf(1);
  _$w(1, 0, _$c), _$tracer.log(arguments.length, 'arguments.length', 1, 0);
  return _$w(1, 1, _$c), a + b;
}
```

## FUNCION TIPO FLECHA

```
30 let sumar= function(a,b){return a + b};
31
32 resultado = sumar(1,2);
33 console.log(resultado); 3
34
35 const sumarFuncionTipoFlecha = (a,b) => (a + b);
36
37 resultado = sumarFuncionTipoFlecha(3,5);
38
39 console.log(resultado); 8
```

Con esta función que emita los pasos de la función de la línea 8 nada más que podemos observar que, para la función `sumarFuncionTipoFlecha` se elimina la palabra reservada `function` y se colocan directamente los parámetros luego en lugar de abrir llaves y ejecutar el proceso que debe realizar con esos parámetros y su respectivo `return`. Lo que se hace es dibujar la flecha y luego indicar la tarea que queremos que realice la función.

Por último decimos que es constante para evitar usarla y modificarla

Luego damos al igual

Obtener los valores que fueron pasados por parámetros a la función, como podemos ver la función es tratada como un array. Y obtenemos los argumentos de dicha función

```
let sumar= function(a,b){  
  console.log(arguments[0]); 1  
  console.log(arguments[1]); 3  
  return a + b  
};  
  
resultado = sumar(1,3);  
console.log(resultado); 4
```

```
let sumar= function(a,b){  
  console.log(arguments[0]); 1  
  console.log(arguments[1]); 3  
  console.log(arguments[2]); 4  
  return a + b + arguments[2];  
};  
  
resultado = sumar(1,3,4);  
console.log(resultado); 8
```

Aca vemos como obtener parámetros que fueron obtenidos por argumentos y no por los parámetros que la función soporta luego se sumo y se ejecuto el código con dicha devolución.

El return es el que retoma el valor a sumar y este valor ya transformado por la función es guardado en la variable resultado y luego este es mostrado en pantalla

Aca estamos trabajando para sumar elementos en una función. Teniendo en cuenta agarrar los argumentos pasados sin especificarlo en la función.

```
let resultado = sumarTodo (5, 4, 13, 10, 9);  
console.log(resultado) 41  
  
function sumarTodo(){  
  let sumar = 0;  
  for (let i = 0; i < arguments.length; i++) {  
    sumar += arguments[i]; // sumar = suma + arguments [i]  
  }  
  return sumar;  
}
```

Paso por valor, como podemos ver a no logro cambiar el valor de x. Sino que encima de que no puedo. En un momento dado tuvo el mismo valor de x pero obtuvo una copia. Y x contuvo aun su valor pero a fue modificado y esto rompió su igualdad. Como podemos ver abajo

Como podemos ver. Luego a, al salir

De la función esta se destruye.

```
let x = 10;

function cambiarValor(a){
  a = (arguments[0] = 20);
  return;
};

cambiarValor(x);
console.log(x);    10
console.log(a);    a is not defined
```

Cambiar valor por referencia. Utilizando un objeto persona y moficiando su nombre por el espacio en memoria. Luego La variable p1 se destruye. Apuntan al mismo espacio en persona en un momento dado. Y Uno sobrescribe y da el valor al objeto persona el valor nuevo

The screenshot shows a Visual Studio Code editor window with the following code:

```
12
13 const persona = {
14   nombre: 'Juan',
15   apellido: 'Perez'
16 }
17
18 function cambiarValorObjeto(p1) {
19   p1.nombre = 'Carlos';
20   p1.apellido = 'Lara';
21 }
22
23 //Paso por referencia
24 cambiarValorObjeto( persona );
25 console.log( persona ); { nombre: 'Carlos', apellido: 'Lara' }
```

Hand-drawn annotations in blue ink illustrate memory references:

- A box labeled `persona` points to a memory address `0x333`.
- Inside the `0x333` box, there are labels `n.` and `a.` with arrows pointing to `Juan` and `Perez` respectively.
- Below the `0x333` box, the names `Carlos` and `Lara` are written, with arrows pointing to the `n.` and `a.` labels respectively, indicating the update of the object's properties.

```
// Definir un objeto

let persona = {
  nombre : 'Juan',
  apellido : 'Perez',
  email : 'jperez@mail.com',
  edad : 28
}

// se crea un objeto en memoria y se le asigna a la variable de persona que contiene nuestros datos
// asi accedemos a los nombres
persona.nombre
```

Asi se define un objeto y asi se accede a los atributos del objeto. El cual dps puedo imprimir

```
console.log(persona.nombre); Juan
console.log(persona.edad); 28
console.log(persona.apellido); Perez

// de este modo accedo a todo el atributo:
console.log(persona); { nombre: 'Juan', apellido: 'Perez', email: 'jperez@mail.com', edad: 28 }
```

```
let persona = {
  nombre : 'Juan',
  apellido : 'Perez',
  email : 'jperez@mail.com',
  edad : 28,
  nombreCompleto : function(){
    return this.nombre + ' ' + this.apellido;
  }
}
```

Funcion dentro de un objeto. El cual luego será tomado como un atributo mas del objeto

```
let persona = {
  nombre : 'Juan',
  apellido : 'Perez',
  email : 'jperez@mail.com',
  edad : 28,
  nombreCompleto : function(){
    return this.nombre + ' ' + this.apellido;
  },
  // miFuncion: function(){}
}
```

Asi accedo a la función o la muestro:

```
// Al crear una funcion
console.log(persona.nombreCompleto()); Juan Perez
```

Como es una función para acceder a esta debo poner los paréntesis para mandar a llamarla.

Es importante aclarar que la función es igual a que decir un método de la clase

#### Asi se crea un objeto:

```
// crear un objeto
let persona2 = new Object();
persona2.nombre = 'carlos';
persona2.direccion = 'Saturno 15';
persona2.tel = '55443322';
```

```
console.log(persona2.tel); 55443322
```

Asi es como lo muestro.

Acceder a una parte del objeto tratandolo como un arreglo:

```
//! Acceder a las propiedades de un objeto como si fuera un arreglo
console.log(persona.apellido); Perez
console.log(persona['apellido']); Perez
```

Siempre debe estar indicado con comillas

Formas de recorrer un objeto con for in.

```
//For in. Este es un for que recorre el objeto.
for (nombrePropiedad in persona) {
  console.log(nombrePropiedad); nombre, apellido, email, edad, nombreCompleto
  console.log(persona[nombrePropiedad]); Juan, Perez, jperez@mail.com, 28, [λ: nombreCompleto]
}
```

Lo que hacemos aquí es definir una propiedad con “nombrepropiedad” dentro de cierto objeto en este caso. Persona. Dps mandamos a imprimir los valores con la propiedad que ya definimos

Ahora si queremos acceder al valor del objeto. Ponemos entre corchetes el nombrePropiedad el cual se va a encargar de recorrerlo

Agregar y quitar propiedades de un objeto.

```
// Con esto agrego un nuevo atributo al objeto.
//                                     - No es muy recomendable-
persona.tel = '55443322';
// Si agregamos otro atributo como tel, va a pisar el anterior tel.

//! Asi eliminamos una atributo de nuestro objeto.
delete persona.tel;
```

Como imprimir un objeto en nuestro navegador web

```
// Concatenar cada valor de cada propiedad:
console.log(persona.nombre + ' ' + persona.apellido); Juan Perez

for (NombrePropiedad in persona) {
  console.log(persona[NombrePropiedad]) Juan, Perez, jperez@mail.com, 28, [λ: nombreCompleto]
}

// este nos regresa nuestro objeto como un arreglo
let personaArray = Object.values(persona);
console.log(personaArray); [ 'Juan', 'Perez', 'jperez@mail.com', 28, [λ: nombreCompleto] ]
```

En el ultimo caso: estamos viendo el método values. Que nos regresa nuestro objeto pero como un arreglo

Asi que se define una variable a la que nombramos personaArray ya que como se menciona al hacer object.values(pasamos el objeto) este nos devuelve un array y a este array hay que almacenarlo para poder tratarlo.

Dps podemos imprimirlo como si fuera un array

```
// este nos regresa nuestro objeto como un arreglo
let personaArray = Object.values(persona);
console.log(personaArray ); [ 'Juan', 'Perez', 'jperez@mail.com', 28, [λ: nombreCompleto] ]
console.log(personaArray [0]); Juan
```

Y por ultimo podemos utilizar también una cadena, por ejemplo asignando una variable como personaString y utilizando el método JSON.stringify(con el object a convertir)

```
let personaString = JSON.stringify(persona);
console.log(personaString); {"nombre":"Juan","apellido":"Perez","email":"jperez@mail.com","edad":28}
```

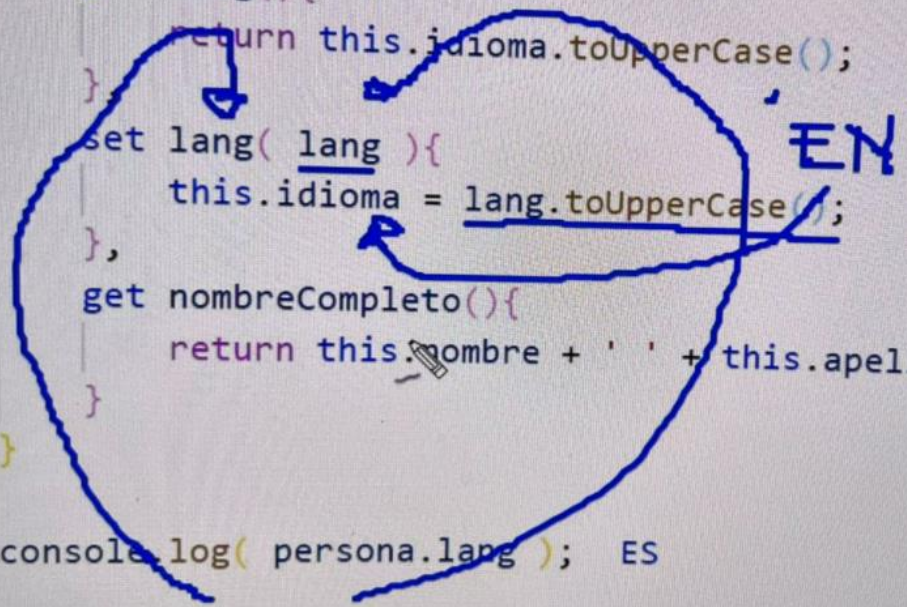


### METODO GET:

```
let persona = {  
  nombre : 'Juan',  
  apellido : 'Perez',  
  email : 'jperez@mail.com',  
  edad : 28,  
  get nombreCompleto(){  
    return this.nombre + ' ' + this.apellido;  
  },  
  // miFuncion: function(){}  
}  
  
console.log(persona.nombreCompleto); Juan Perez
```

Elimina la function y además simplifica el tratado de atributos del objeto persona.

### Metodo SET:



```
get lang(){  
  return this.idioma.toUpperCase();  
},  
set lang( lang ){  
  this.idioma = lang.toUpperCase();  
},  
get nombreCompleto(){  
  return this.nombre + ' ' + this.apellido;  
}  
}  
  
console.log( persona.lang ); ES  
  
persona.lang = 'en';
```

The image shows a code snippet with a handwritten blue circle and arrows. The circle starts at the `set lang(lang)` method, goes to the `this.idioma = lang.toUpperCase();` line, then loops back to the `get nombreCompleto()` method. To the right of the circle, the text `'EN'` is written in blue ink. Below the code, the text `console.log( persona.lang ); ES` and `persona.lang = 'en';` are visible.

Como actúa el método SET en nuestro objeto.

```

let persona = {
  nombre : 'Juan',
  apellido : 'Perez',
  email : 'jperez@mail.com',
  edad : 28,
  idioma: 'es',
  get lang(){
    return this.idioma.toUpperCase();
  },
  set lang(lang){
    this.idioma = lang.toUpperCase();
  },
  get nombreCompleto(){
    return this.nombre + ' ' + this.apellido;
  },
  // miFuncion: function(){}
}

console.log(persona.nombreCompleto); Juan Perez
console.log(persona.lang); ES

persona.lang = 'es_ar';
console.log(persona.lang); ES_AR
// Con esta propiedad vamos a confirmar si realmente funciona de manera correcta.
console.log(persona.idioma); ES_AR

```

Creacion de un constructor:

```

// Creacion de un constructor. Siempre se escribe el constructor en mayuscula
function Persona(nombre, apellido, email){
  // aca se crea la propiedad y se crea la variable. O sea la variable y la propiedad
  this.nombre = nombre;
  this.apellido = apellido;
  this.email = email;
  // seria como que asignamos el valor del parametro que estamos recibiendo
}

```

Gracias al constructor que nos genera un objeto por cada nuevo objeto que le pasemos

```

// Creacion de un constructor. Siempre se escribe el constructor en mayuscula
function Persona(nombre, apellido, email){
  // aca se crea la propiedad y se crea la variable. O sea la variable y la propiedad
  this.nombre = nombre;
  this.apellido = apellido;
  this.email = email;
  // seria como que asignamos el valor del parametro que estamos recibiendo
}

let padre = new Persona('Juan', 'Perez', 'jperez@mail.com');
console.log (padre); Persona { nombre: 'Juan', apellido: 'Perez', email: 'jperez@mail.com' }

let madre = new Persona ('Laura', 'Quintero', 'Lquintero@mail.com');
console.log (madre); Persona { nombre: 'Laura', apellido: 'Quintero', email: 'Lquintero@mail.com' }

```

Por ejemplo aca utilizamos el constructor Persona que generara un objeto según lo que le pasemos. Si creamos un objeto padre y le pasamos a la función persona los valores del padre este nos creara un objeto.

Cada vez que usamos new se crea un nuevo objeto en memoria.

Y al utilizar la función Persona que es un constructor y pasando los parámetros vamos a tener un objeto padre con los atributos que le pasamos

Asi editamos el constructor:

```
padre.nombre = 'carlos';  
console.log ( padre );| Persona { nombre: 'carlos', apellido: 'Perez', email: 'jperez@mail.com' }
```

Función para dentro de un constructor.

```
this.NombreCompleto = function(){  
|   return this.nombre + ' ' + this.apellido;  
| }  
}
```

Formas de crear un objeto.

```
// Formas de crear un objeto:  
  
let miObjeto = new Object();  
let miObjeto2 = {};  
  
let miCadena1 = new String ('hola');  
let miCadena2 = 'Hola';  
  
let miNumero = new Number (1);  
let miNumero1 = 1;  
  
let miBoolean = new Boolean (false);  
let miBoolean2 = false;  
  
let miArreglo1 = new Array();  
let miArreglo2 = [];  
  
let miFuncion = new Function();  
let miFuncion2 = function (){};|
```

## Prototype:

Agregar un nuevo atributo a nuestro constructor y a su vez así poder agregarlo o modificarlo en nuestros objetos.

```
// Aca agregamos un nuevo atributo.
Persona.prototype.tel = '44332211';

// y luego podemos modificarlo.

padre.tel = '11224455';
madre.tel = '66778899';

console.log(padre.tel + " Padre / Madre " + madre.tel); 11224455 Padre / Madre 66778899
```

## Funcion call

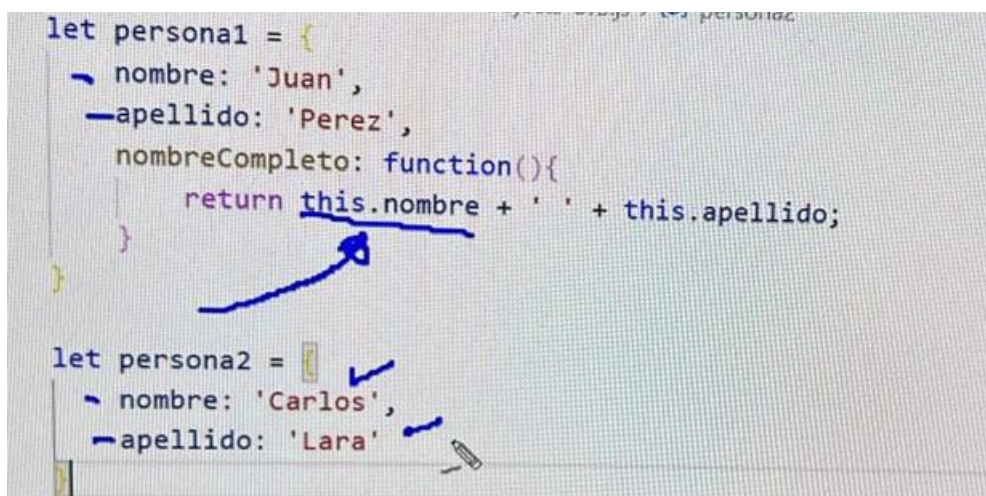
```
let persona1 = {
  nombre : 'Juan',
  apellido : 'Perez',
  nombreCompleto : function(){
    return this.nombre + ' ' + this.apellido;
  }
}

let persona2 = {
  nombre : 'Carlos',
  apellido : 'Lara'
}

// Uso de call para usar el metodo de persona1.nombreCompleto con los datos de la persona 2
console.log(persona1.nombreCompleto()); Juan Perez

console.log(persona1.nombreCompleto.call(persona2)); Carlos Lara
```

Con esta función vamos a poder llamar a la persona1 con los atributos de la persona2 ya que como estos comparten nombre y apellido. Podemos hacerlo



```
let persona1 = {
  nombre: 'Juan',
  apellido: 'Perez',
  nombreCompleto: function(){
    return this.nombre + ' ' + this.apellido;
  }
}

let persona2 = {
  nombre: 'Carlos',
  apellido: 'Lara'
}
```

The image shows the same code as the previous block but with handwritten blue annotations. A blue arrow points from the `this` keyword in the `nombreCompleto` function to the `persona1` object definition. Another blue arrow points from the `persona2` object to the `call(persona2)` argument in the console log. There are also blue checkmarks next to the `nombre` and `apellido` properties in both objects, indicating they are shared.

Con la función call también podemos pasar argumentos que reciba la función

```

let persona1 = {
  nombre: 'Juan',
  apellido: 'Perez',
  nombreCompleto: function(titulo, tel){
    return titulo + ': ' + this.nombre + ' ' + this.apellido + ', ' + tel;
  }
}

let persona2 = {
  nombre: 'Carlos',
  apellido: 'Lara'
}

//Uso de call para usar
//el metodo persona1.nombreCompleto con los datos del persona2
console.log( persona1.nombreCompleto('Lic.', '44332288') );

console.log( persona1.nombreCompleto.call( persona2, 'Ing', '5544332211' ) );

```

Como podemos ver. Pasamos dos parámetros que la función los entendera y tratara :

```

console.log( persona1.nombreCompleto.call( persona2, 'Ing', '5544332211' ) );  Ing: Carlos Lara, 5544332211

```

Uso de apply : En resumen este método en lugar de pasarlo junto a la llamada de persona. Lo que hacemos es aplicar a esa persona un arreglo y a ese arreglo será pasado como parámetro

```

let arreglo = ['Ing', '55443322'];
console.log( persona1.nombreCompleto.apply( persona2, arreglo ) );  Ing: Carlos Lara, 55443322

```

Solo es esta la diferencia el objeto persona1 sigue funcionando igual



## Clases:

```
// definir una clase- Debe iniciar con mayusculas :
class Persona{
  // agrego el constructor:
  constructor(nombre, apellido){
    // inicializo los atributos:
    this.nombre = nombre; //dps del igual es el parametro que recibo distinto al atributo propio de la clase
    this.apellido = apellido;
  }
}

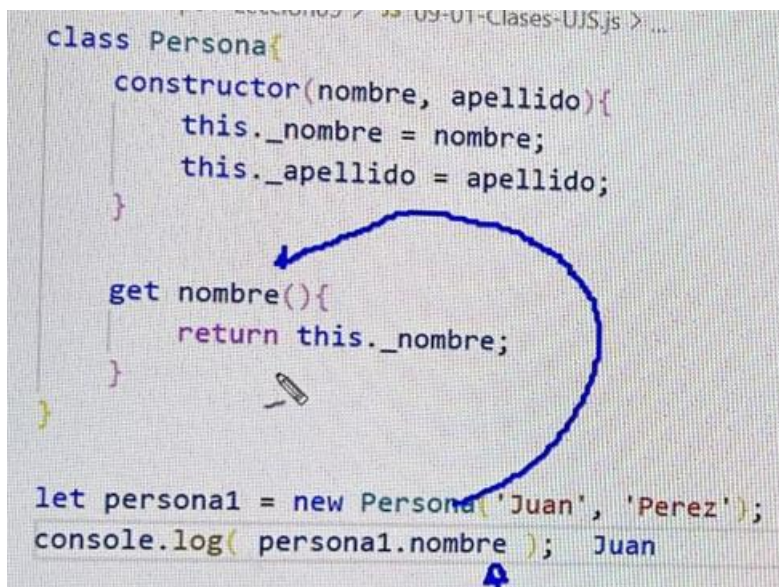
// creo el objeto

let persona1 = new Persona('Carlos', 'Perez');
console.log (persona1);  Persona { nombre: 'Carlos', apellido: 'Perez' }

let persona2 = new Persona ('Juan', 'Lara');
console.log (persona2);  Persona { nombre: 'Juan', apellido: 'Lara' }
```

Muy similar a objetos

Solo que usa el dominio de class.



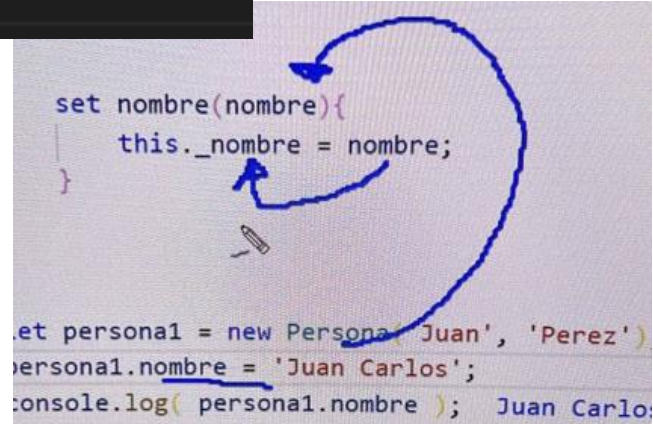
```
class Persona{
  constructor(nombre, apellido){
    this._nombre = nombre;
    this._apellido = apellido;
  }

  get nombre(){
    return this._nombre;
  }
}

let persona1 = new Persona('Juan', 'Perez');
console.log( persona1.nombre );  Juan
```

Asi funciona el método get. Asi es como se le pide que se le muestre y procese y este es lo que hace.

```
let persona1 = new Persona('Carlos', 'Perez');
// console.log (persona1);
//TODO utilizando el metodo GET
console.log(persona1.nombre);  Carlos
//Todo utilizando el set
persona1.nombre = 'Juan Carlos'; //set nombre ('Juan Carlos')
console.log(persona1.nombre);  Juan Carlos
```



```
set nombre(nombre){
  this._nombre = nombre;
}

let persona1 = new Persona('Juan', 'Perez');
persona1.nombre = 'Juan Carlos';
console.log( persona1.nombre );  Juan Carlos
```

Las clases no tienen hoisting. No es posible crear objetos antes de crear las clases

## HERENCIA:

```
class Persona{
  constructor(nombre, apellido){
    this._nombre = nombre;
    this._apellido = apellido;
  }
  get nombre(){
    return this._nombre;
  }
  set nombre(nombre){
    this._nombre = nombre;
  }

  get apellido(){
    return this._apellido;
  }
  set apellido(apellido){
    this._apellido = apellido;
  }
}

// creo la clase hija

class Empleado extends Persona{
  // anexamos los parametros de la clase padre
  constructor(nombre, apellido, departamento){
    // hay que llamar al constructor de la clase padre y la tenemos
    // que utilizar arriba del constructor de la clase hija
    super(nombre, apellido); // llamar al constructor de la clase padre
    this._departamento = departamento;
  }

  get departamento(){
    return this._departamento;
  }
  set nombre(nombre){
    this._departamento = departamento;
  }
}

let persona1 = new Persona('Juan', 'Perez');
console.log (persona1);

let empleado1 = new Empleado('Maria', 'Jimenez', 'Sistemasas');

console.log(empleado1);
console.log(empleado1._nombre);
```



### Defino la clase

```
class Persona{
    constructor(nombre, apellido){
        this._nombre = nombre;
        this._apellido = apellido;
    }
    get nombre(){
        return this._nombre;
    }
    set nombre(nombre){
        this._nombre = nombre;
    }

    get apellido(){
        return this._apellido;
    }
    set apellido(apellido){
        this._apellido = apellido;
    }
}
```

Realizo la herencia teniendo en cuenta el super. Para que la clase herede los parámetros y los atributos

```
class Empleado extends Persona{
    // anexamos los parametros de la clase padre
    constructor(nombre, apellido, departamento){
        // hay que llamar al constructor de la clase padre
        super(nombre, apellido); // llamar al constructor de
        this._departamento = departamento;
    }

    get departamento(){
        return this._departamento;
    }
    set nombre(nombre){
        this._departamento = departamento;
    }
}
```

### Ahora creo un objeto persona1 y un objeto empleado1

```
let persona1 = new Persona('Juan', 'Perez');
console.log (persona1);  Persona { _nombre: 'Juan', _apellido: 'Perez' }

let empleado1 = new Empleado('Maria', 'Jimenez', 'Sistemasas');

console.log(empleado1);  Empleado { _nombre: 'Maria', _apellido: 'Jimenez', _departamento: 'Sistemasas' }
console.log(empleado1._nombre);  Maria
```

Como podemos ver un atributo se crea utilizando solo la clase persona y el otro con la herencia de la clase EMPLEADO. Que tiene los mismos atributos que persona pero con un departamento

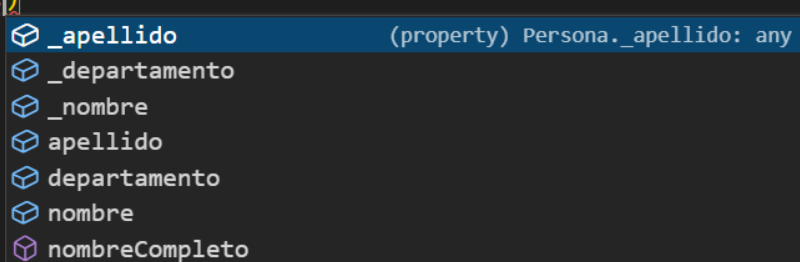
### Heredar métodos:

Definir un método en la clase padre para heredarlo en la clase hija

```
// esto es una funcion
nombreCompleto(){
    return this._nombre + ' ' + this._apellido;
}
```

Definiendo la función de este modo, la misma va a ser heredada por la clase hija

```
console.log(empleado1.)
```



Aca podemos ver como podemos acceder al método

### Sobreescritura:

En la clase padre tenemos definidos los atributos nombre y apellido pero desconoce completamente departamento ya que el mismo pertenece a la clase hija

Para esto, se aplica la sobreescritura. Desde el punto de la clase hija la clase padre no esta completa:

```

nombreCompleto(){
    return this._nombre + ' ' + this._apellido + ', ' + this._departamento; // este ultimo es el de la clase hija
}

let persona1 = new Persona('Juan', 'Perez');
console.log (persona1); Persona { _nombre: 'Juan', _apellido: 'Perez' }

let empleado1 = new Empleado('Maria', 'Jimenez', 'Sistemas');

console.log(empleado1); Empleado { _nombre: 'Maria', _apellido: 'Jimenez', _departamento: 'Sistemas' }
console.log(empleado1._nombre); Maria

// metodo que fue utilizado para la herencia por metodo
console.log(empleado1.nombreCompleto()); Maria undefined, Sistemas

```

Esta seria una forma de hacerlo.

```

nombreCompleto(){
    return super.nombreCompleto + ', ' + this._departamento; // este ultimo es el de la clase hija
}

```

Gracias a “SUPER”. (aca damos la función que va a herdar) gracias a esto podemos reutilizar la función de la clase padre. Sin necesidad de volver a escribir todo.

La clase object es la clase padre de todas las clases de js

Al saber esto, podemos decir lo siguiente, la clase object contiene diferentes métodos y uno de ellos es el método toString.

Y esto nos sirve para imprimir los valores actuales en la que se encuentra el método.

Prototype : nos permite agregar atributos de manera dinámica

### Metodo toString:

```

// esto es una funcion
nombreCompleto(){
    return this._nombre + ' ' + this._apellido;
}

// Sobreescritura: ya que nombre completo se hereda en las dos clases
toString(){
    // se aplica polimorfismo (multiples formas en mismo tiempo de ejecucion)
    // el metodo que se ejecuta depende si es una referencia de tipo padre o de tipo hijo
    // dependiendo de que se ejecuta. Tendemos un resultado distinto. Dependiendo del parametro que se reciba
    return this.nombreCompleto();
}

```

```

let persona1 = new Persona('Juan', 'Perez');
console.log (persona1); // get nombre Persona { _nombre: 'Juan', _apellido: 'Perez' }

// metodo toString
console.log(persona1.toString()); Juan Perez

let empleado1 = new Empleado('Maria', 'Jimenez', 'Sistemas');

console.log(empleado1); Empleado { _nombre: 'Maria', _apellido: 'Jimenez', _departamento: 'Sistemas' }
console.log(empleado1._nombre); Maria

// metodo que fue utilizado para la herencia por metodo
console.log(empleado1.nombreCompleto()); Maria Jimenez, Sistemas

// para mostrar en consola o en pantalla el estado del metodo pero seteandolo y dandole un mejor formato.
// tambien podemos ver que como nos trae tambien el departamento de la persona. Quiere decir que la misma esta siendo heredada por la clase hija

console.log(empleado1.toString()) Maria Jimenez, Sistemas

```

**Como vemos la clase hija no hace nada. Todo se hereda de la clase padre**

## Metodo static SE ASOCIA CON NUESTRA CLASE

Este método solo se asocia a una clase y solo a la clase y no con los objetos:

```
static saludar(){  
    console.log('Saludos desde la prueba')  
}
```

y no con los objetos:

```
persona1.saludar();
```

Esto nos da error porque ahí estamos haciendolo con el objeto. Intentando usar saludar que esta dentro de la clase Persona

Para esto hay que hacer el saludo desde la clase:

```
// persona1.saludar(); no es posible llamar a un metodo static desde un objeto  
Persona.saludar();
```

Ahora si queremos que esto si funcione, debemos pasarlo como parámetro es por eso que hacemos lo siguiente:

```
static saludar2(Persona){  
    console.log(Persona.nombre); Juan  
}
```

```
static saludar2(personales){  
    console.log(personales.nombre); Juan  
}
```

Como vemos, acá podemos entender que el parámetro que indiquemos adentro puede ser cualquier nombre que nos haga referencia

Ya que se tomará al nosotros escribir esto:

```
// saludar pero con parametros y desde un objeto  
Persona.saludar2(persona1);
```

Entonces nosotros pasamos el objeto que creamos anteriormente

Este es el objeto creado anteriormente `let persona1 = new Persona('Juan', 'Perez');`

Ahora vamos a ver ATRIBUTOS estáticos static:

```
// utilizar el atributo static
// ya que la variable se esta asignando a la clase
console.log(Persona.contadorObjetosPersona); 2

// es 2 porque nosotros creamos dos objetos. Uno de la clase hija
// que utiliza por herencia el constructor de la clase padre.
```

```
class Persona{
  // Atributos statics:
  static contadorObjetosPersona = 0;

  constructor(nombre, apellido){
    this._nombre = nombre;
    this._apellido = apellido;
    // no se usa el this para acceder a variables staticas propias de la clase
    // Se usa directamente la clase
    Persona.contadorObjetosPersona++;
  }
}
```

Atributos no estáticos: no se asocian con nuestra clase sino que son atributos de nuestro objeto:

```
email = "Valor default email"; //Atributo de nuestros objetos
```

```
4
5 // Atributo no estatico.
6 console.log(persona1.email); Valor default email
7 console.log(empleador1.email); Valor default email
```

Variables estáticas de solo lecturas.

```
static get MAX_OBJ(){
  return 5;
}
```

```
if (Persona.contadorPersona < Persona.MAX_OBJ) {
  this._idPersona = ++Persona.contadorPersona;
}else{
  console.log("Maximo de personas (5)"); Maximo de personas (5)
}
```

```

console.log(Persona.MAX_OBJ); 5

let persona2 = new Persona('Elian', 'Enria');

let empleado2 = new Empleado('Licha', 'Lopez');

let persona3 = new Persona('Enzo', 'Comba');

let persona6 = new Persona('Martin', 'Husbaldo');

// objeto sin desborde
console.log(persona3.toString()); 5 Enzo Comba

// objeto desbordado o sin id
console.log(persona6.toString()); undefined Martin Husbaldo

```

EJERCICIO DE PRUEBA DIAGRAMA DE CLASE CON UML ESTA EN LA LECCION 06 modelado de clases y programación

Para declarar el modo stricto en js:

Se escribe “use strict”

```

"use strict"
// variable sin definir y error en el modo estricto
// x = 10;
let x = 10;
console.log(x); 10

miFuncion();

function miFuncion(){
  //TODO Tambien lo podemos utilizar aqui para que se aplique dentro de la funcion
  // "use strict"
  // sin definir
  // y = 15;
  let y = 15;
}

```

Sobreescritura.

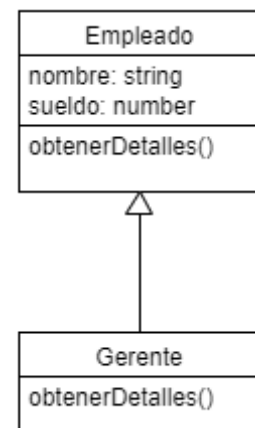
Para que primero se cumpla, necesitamos una clase padre y una clase hija. O sea herencia

```
    obtenerDetalles(){
        return `Empleado: Nombre: ${this._nombre}, sueldo: ${this._sueldo}`;
    }
}

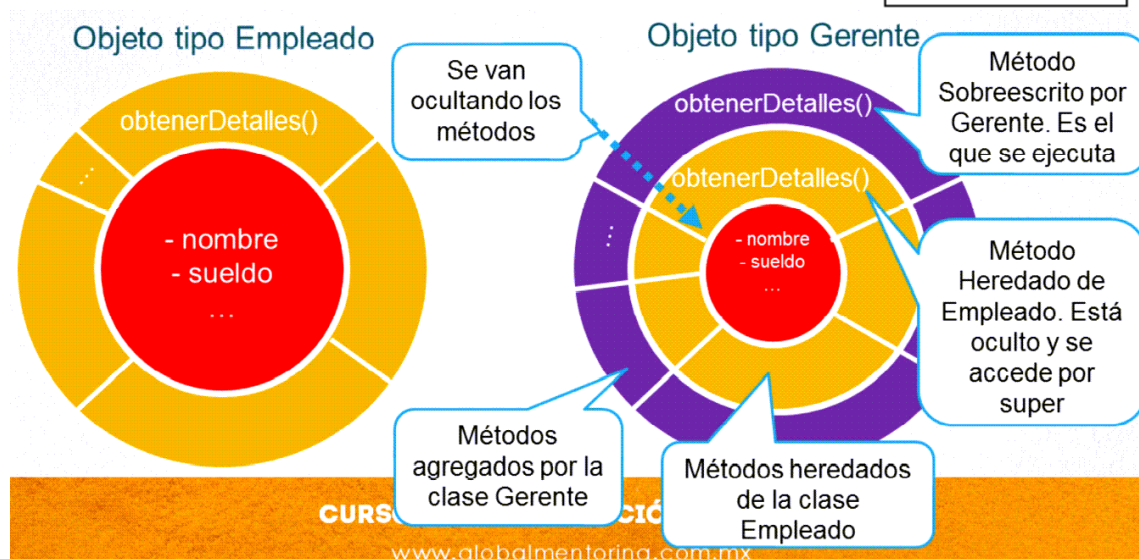
class Gerente extends Empleado{
    constructor (nombre, sueldo, departamento){
        super(nombre,sueldo, departamento);
        this._departamento = departamento;
    }
    // Aca estamos sobre escribiendo el metodo de la clase padre y agregandole los datos de la clase hija
    obtenerDetalles(){
        return `Gerente: ${super.obtenerDetalles()} + depto: ${this._departamento}`;
    }
}

let gerente1 = new Gerente ('Carlos',5000,'Sistemas');
console.log(gerente1.obtenerDetalles()); Gerente: Empleado: Nombre: Carlos, sueldo: 5000 + depto: Sistemas
```

Según la clase empleado y gerente este seria como se escribe la herencia y como se aplica la sobre escritura en un método de la clase heredada



Para mandar a llamar a la clase padre, y para acceder a los métodos  
Se utiliza la clase padre.



**Polimorfismo:**

```

//! Polimorfismo
// Metodo independiente
function imprimir(tipo){
    // segun el tipo que proporcionemos es lo que se va a imprimir
    // Con una linea de codigo podemos ejecutar un metodo de la clase padre o de la clase hija.
    // dependiendo del tipo que este apuntando

    console.log(tipo.obtenerDetalles()); Empleado: Nombre: Juan, sueldo: 3000, Gerente: Empleado
}

let empleado1 = new Empleado('Juan', 3000);
// console.log(empleado1.obtenerDetalles());

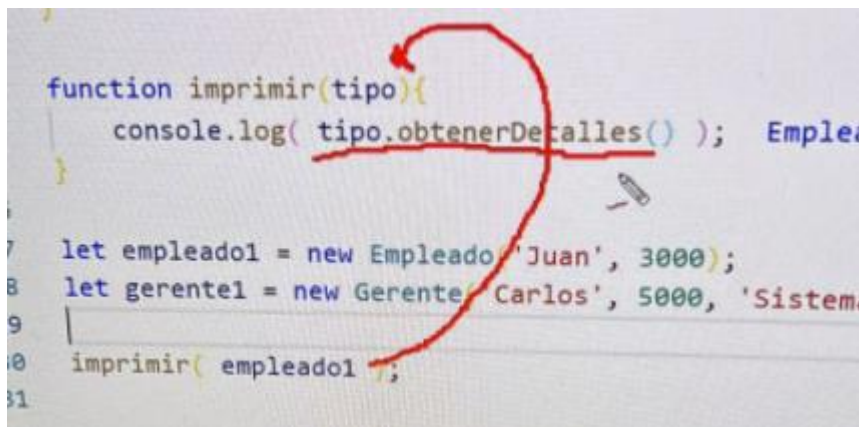
let gerente1 = new Gerente ('Carlos',5000,'Sistemas');
// console.log(gerente1.obtenerDetalles());

imprimir(empleado1);
imprimir(gerente1);

```

Aca podemos ver que al pasar un objeto de la clase padre nos mostrara solo los atributos de la clase padre nombre y sueldo.

Ahora si pasamos, los atributos solamente de la clase hija, esta misma función que aplica la metodología de polimorfismo vamos a tener en este caso el nombre, sueldo y (departamento) siendo este ultimo propio de la clase hija



```

function imprimir(tipo){
    console.log( tipo.obtenerDetalles() ); Empleado
}

let empleado1 = new Empleado('Juan', 3000);
let gerente1 = new Gerente('Carlos', 5000, 'Sistemas');
imprimir(empleado1);

```

Instanceof (instancia de) esta variable reserva no responde que si a las variables de tipo hijo. Solo aplica a las clases tipo padre o tipo hijo ya que una hereda a la otra pero al revés no

```

if (tipo instanceof Gerente){
    console.log('objeto de tipo gerente'); objeto de tipo gerente
}

```

En este caso la clase gerente es de tipo hijo, por lo que la clase padre no tendrá ninguna respuesta y deberá ir por el camino del else. Pero en su lugar la clase fuera empleado. Esta respondería si en ambos casos porque es la clase padre que tiene ambos atributos agregados a estas



Como definir a instanceof . tipo tiene que estar previamente definido y este tiene que ser el parámetro de la función. O que obtenga algún tipo de método de otra clase.

```
console.log(tipo.obtenerDetalles()); Empleado: Nombre: Juan, sueldo: 3000, Gerente: Empleado: Nombre: Carlos, sueldo: 5000 + depto: Sistemas  
if (tipo instanceof Gerente){ // asi se define la clase instance of  
  console.log('objeto de tipo gerente'); objeto de tipo gerente  
}else{console.log('es clase empleado')}; es clase empleado
```

```
function imprimir(tipo){  
  // segun el tipo que proporcionemos es lo que se va a imprimir  
  // Con una línea de código podemos ejecutar un método de la clase padre o de la clase hija.  
  // dependiendo del tipo que este apuntando  
  console.log(tipo.obtenerDetalles()); Empleado: Nombre: Juan, sueldo: 3000, Gerente: Empleado: Nombre: Carlos, sueldo: 5000 + depto: Sistemas  
  if (tipo instanceof Gerente){ // asi se define la clase instance of  
    console.log('Es un objeto de tipo gerente'); Es un objeto de tipo gerente  
    console.log("Departamento de: " + tipo._departamento); Departamento de: Sistemas  
  }else{  
    console.log('es clase empleado ' + tipo._nombre) es clase empleado Juan  
    // pero.  
    console.log("es del departamento de: " + tipo._departamento); es del departamento de: undefined  
  }  
}
```

Como podemos ver, ese atributo no existe a la referencia de tipo departamento en la clase empleado ya que esta es de mayor jerarquía

Entonces la estructura seria de menor jerarquía a mayor

FUNCION FLECHA. Esto lo vamos a ver ya cargado y que inform dentro de la carpeta de lecciones solo dejo las imagenes.

Formas de imprimir y de generar las función flecha

```

let miFuncion = function () {
  console.log('saludos desde mi función');
}

// const miFuncionFlecha = () => {
//   console.log('saludos desde mi función flecha');
// }

//const miFuncionFlecha = () => console.log('saludos desde mi función flecha')

//miFuncionFlecha();

// const saludar = () => {
//   return 'Saludos desde función flecha';
// }

const saludar = () => 'Saludos desde función flecha';

console.log( saludar() );

const regresaObjeto = () => ({nombre: 'Juan', apellido: 'Lara'});
console.log( regresaObjeto());

const funcionConParametrosClasico = function(mensaje){
  console.log(mensaje);
}

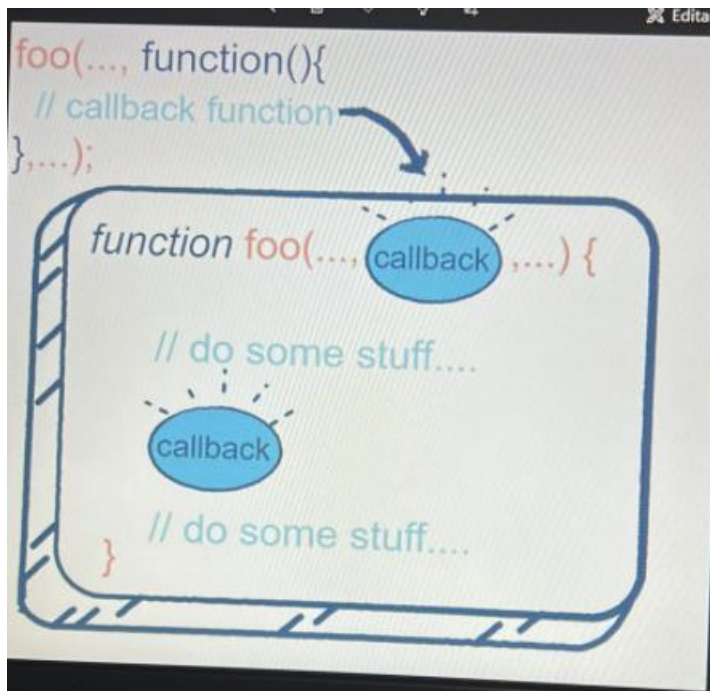
//const funcionConParametros = (mensaje) => console.log( mensaje );
const funcionConParametros = mensaje => console.log( mensaje );
funcionConParametros('saludos con parametros');

//const funcionConVariosParametros = (op1, op2) => op1 + op2;
const funcionConVariosParametros = (op1, op2) => {
  let resultado = op1 + op2;
  return `Resultado: ${resultado}`;
};
console.log( funcionConVariosParametros(3,5));

```

## Funciones tipos callback

Bien, básicamente una función de tipo callback es una función que se pasa como parámetro a otra función y dentro de una función vamos a poder llamar a otra función.



Esto es lo que hace:

```

12 //Función de tipo callback
13 function imprimir(mensaje){
14   console.log(mensaje); Resultado: 8
15 }
16
17 function sumar(op1, op2, funcionCallback){
18   let res = op1 + op2;
19   funcionCallback(`Resultado: ${res}`);
20 }
21
22 sumar(5,3, imprimir);

```

```

function imprimir(mensaje){
  console.log(mensaje); resultado: 8
}

function sumar(op1, op2, imprimir){
  let res = op1 + op2 ;
  imprimir( "resultado: " +res);
}

sumar(5,3, imprimir);

```

```

//Función de tipo callback
function imprimir(mensaje){
  console.log(mensaje); Resultado: 8
}

function sumar(op1, op2, funcionCallback){
  let res = op1 + op2;
  funcionCallback(`Resultado: ${res}`);
}

sumar(5,3, imprimir);

```

callback

La función callback que se podría llamar también imprimir pasa como parámetro que sería el mensaje le manda a mensaje como parámetro el resultado de la suma. Mensaje como es un parámetro de imprimir ejecuta imprimir console log. Y esto se muestra en pantalla.

Lo mejor sería hacer que imprimir haga referencia a todo

### SetTimeout:

```
// llamadas asincronas con uso de setTimeout
function miFuncionCallback (){
  console.log('saludo asincrono despues de 3seg');  saludo asincrono despues de 3seg
}

setTimeout(miFuncionCallback, 3000 ); //no hace falta poner los parentesis
```

Formas de hacer usar los timeout: SIEMPRE DEBE RECIBIR UNA FUNCION

```
// llamadas asincronas con uso de setTimeout
function miFuncionCallback (){
  console.log('saludo asincrono despues de 3seg');
}

setTimeout(miFuncionCallback, 3000 ); //no hace falta poner los parentesis

setTimeout(function(){console.log('saludo asincrono 2')}, 5000);

setTimeout(() => console.log("Saludo Flecha 8 "), 8000);

const tiempo = 3000;
setTimeout(() => {
  console.log("saludo flecha 1")
}, tiempo);
```

Funcion

### SetInterval:

```
let reloj = () => {
  let fecha = new Date();
  console.log(`${fecha.getHours()}: ${fecha.getMinutes()}: ${fecha.getSeconds()}`);
}

setInterval(reloj, 1000);
```

Promesas: Utiliza el objeto promise.

```
let miPromesa = new Promise((resolver, rechazar) => {
  let expresion = true;
  if(expresion)
    resolver('Resolvió correcto');
  else
    rechazar('Se produjo un error');
});

//TODO Esto puede estar momentaneamente al costado o abajo
//miPromesa.then( valor => console.log(valor), error => console.log(error));
//miPromesa.then(valor => console.log(valor)).catch(error=>console.log(error));

let promesa = new Promise((resolver) => {
  //console.log('inicio promesa');
  setTimeout( ()=> resolver('saludos con promesa y timeout'), 3000);
  //console.log('fin promesa');
});

//promesa.then(valor => console.log(valor));
```

```
//async indica que una función regresa una promesa
async function miFuncionConPromesa(){
  return 'saludos con promesa y async';
}

//miFuncionConPromesa().then(valor => console.log(valor));

//async/await
async function funcionConPromesaYAwait(){
  let miPromesa = new Promise( resolver => {
    resolver('Promesa con await');
  });

  console.log( await miPromesa );
}

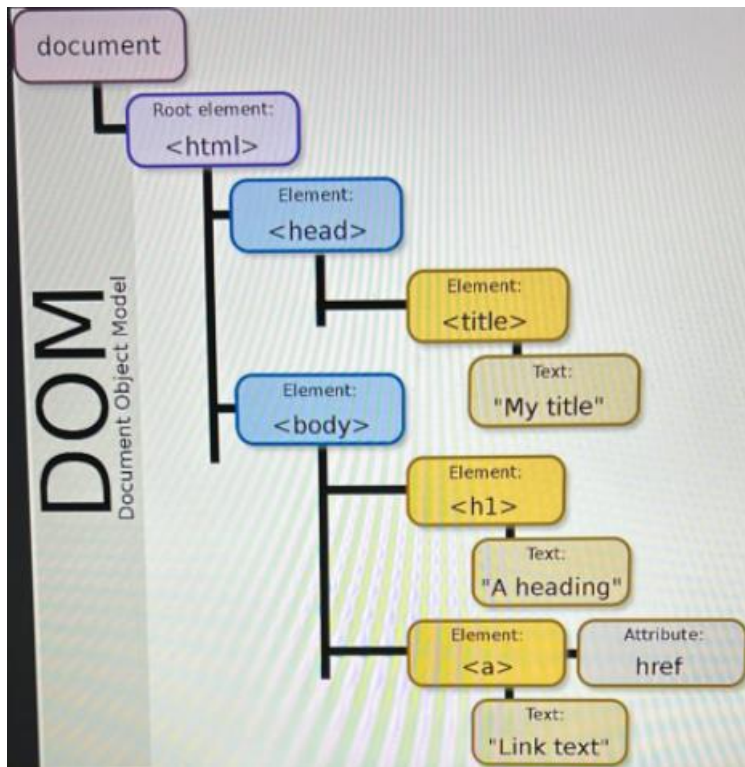
//funcionConPromesaYAwait();
```

**Await** ( Siempre debe estar dentro de la función **async** )

```
//promesas, await, async y setTimeout
async function funcionConPromesaAwaitTimeout(){
  console.log('inicio función');
  let miPromesa = new Promise(resolver => {
    setTimeout(()=> resolver('promesa con await y timeout'), 3000);
  });
  console.log( await miPromesa);
  console.log('fin función');
}

funcionConPromesaAwaitTimeout();
```

Manejo del DOM:



Con document vamos a poder acceder a los elementos de html

```
<body>
  <h1 id="cabecero">Ejemplo DOM con JS</h1>
  <p id="parrafo">Saludos!</p>
</body>
</html>

<script>
  let cabecero = document.getElementById('cabecero');
  console.log('Saludo desde JS');
</script>
```

Gracias al ID cabecero vamos a poder obtenerlo desde el script de java script y con el método document.getElementById y luego indicando el elemento que queremos obtener vamos a poder acceder al html. Básicamente con document hablamos del dom y con get element by id. Es que estamos accediendo al archivo a través del id que asignamos en el html. Esto nos trae todo el objeto o sea todo el h1

```
let cabecero = document.getElementById('cabecero').innerHTML;
```

Con innerhtml accederemos al valor del documento. En resumen es que

Con `document.getElementById` accedemos al h1 id cabecero y con `innerHTML` accedemos al ejemplo dom con js

Con `innerHTML` también vamos a poder modificar los valores que accedamos a través del id

```
console.log(`valor del parrafo: ${parrafo.innerHTML}`);
parrafo.innerHTML = 'Nuevo valor del parrafo';
```

Aca podemos ver que primero lo obtengo en consola con `parrafo.innerHTML` no obstante este traerá a consola el valor anterior y luego será modificado. Pero esto que modifiquemos se modificara en el html y no en la consola.

```
<body>
  <h1 id="cabecero">Ejemplo DOM con JS</h1>
  <p id="parrafo">Saludos!</p>

  <script>
    let cabecero = document.getElementById('cabecero');
    console.log('valor cabecero: ' + cabecero.innerHTML);

    let parrafo = document.getElementById('parrafo');//obtengo el elemento
    console.log(`valor del parrafo: ${parrafo.innerHTML}`);
    parrafo.innerHTML = 'Nuevo valor del parrafo';

    cabecero.innerHTML = "Nuevo valor del cabecero";

    // console.log('Saludo desde JS');
  </script>
```

El máximo objetivo del `getElementById` es obtener el elemento que indiquemos dentro de los paréntesis que tienen correlación con id del html. Lo que esta antes del igual puede tener cualquier variable ya que esto será parte del js. Por ultimo cuando lo obtenemos podemos ejercer cambios con consola o desde el html en si. Con el `innerHTML` siempre indicamos el valor de la variable relacionada.

**Ahora vamos a ver como obtener mas elementos del mismo tipo:**

**ES IMPORTANTE SABER QUE TAGNAME SIEMPRE SIEMPRE ME DEVUELVE ARREGLOS**  
**ENTONCES ASI LOS PUEDES TRATAR COMO TALES**

A través del tag o de la etiqueta en lugar del id

```
let parrafos = document.getElementsByTagName('p');
// los trata como un array
console.log(`Numeros de parrafos: ${parrafos.length}`);
```

Le indicamos que tag queremos en este caso p. los mismos son tratados como un array de elementos



De este modo podremos mostrar los elementos:

```
for (let i = 0; i < parrafos.length; i++){  
  console.log(parrafos[i].innerHTML);  
}  
  
console.log(`i: ${i} - ${parrafos[i].innerHTML}`)
```

Get element by class name. Esto claramente funciona igual que los demás pero se le pasa el parámetro como clase

```
<script>  
  let elementos = document.getElementsByClassName('azul');  
  // imprimir los elementos  
  console.log(`Cantidad: ${parrafos.length}`);  
  for (let i = 0; i < elementos.length; i++) {  
    // console.log("indice: "+ [i]+ " " + parrafos[i].innerHTML);  
    console.log(`indice: ${[i]} - ${parrafos[i].innerHTML}`);  
  }  
</script>
```

Como obtener elementos de una clase.

```
// del mismo modo. así los modifico  
elementos[i].innerHTML = "NUEVO ELEMENTO ";
```

Como obtener los elementos de una clase:

```
for (const elemento of elementos) {  
  console.log(`${elemento.innerHTML}`);  
}
```

Usando el for of que se encarga de recorrer los elementos y luego ir imprimiéndolos en una variable.

Otro método para recuperar métodos de con una clase de css

Este es el query selector all

En esta clase lo que se hace es pasar como identificador la clase y donde se aplica dicha clase por ejemplo p.azul la clase azul en el tag de párrafo

```

<script>
  let elementos = document.querySelectorAll('p.azul');
  console.log(`cantidad de elementos: ${elementos.length}`);
  for (const elemento of elementos) {
    console.log(`${elemento.innerHTML}`);
  }
</script>

```

Manejo de formularios:

```

<body>
  <form id="formulario">
    Nombre: <input type="text" name="nombre" value="Juan" /><br>
    Apellido: <input type="text" name="apellido" value="Perez"><br>
  </form>
  <button onclick="mostrarValores()">Mostrar</button> <hr>
  <div id="valores"></div>
  <script>
    function mostrarValores(){
      // funcion que manda a llamar al formulario para obtener los datos
      let formularioJS = document.forms['formulario'];
      // recorrer los elementos;
      let texto = '';
      for (let elementos of formularioJS){
        texto += elementos.value + '<br/>'; //Acceder a los elementos y asi lo vamos accediendo
      }
      console.log(texto);
      document.getElementById('valores').innerHTML = texto;
    }
  </script>
</body>
</html>

```

Como acceder a los valores. Como obtenerlos y como mostrarlos tanto en consola y como mostrarlos en el documento

AHORA COMO ACCEDER A LOS ELEMENTOS DE MANERA INDIVIDUAL

```

<body>
  <form id="formulario">
    Nombre: <input type="text" name="nombre" value="Juan"/> <br>
    Apellido: <input type="text" name="apellido" value="Perez"><br>
  </form>
  <button onclick="mostrarValores()">Mostrar</button> <hr>
  <div id="valores"></div>

  <script>
    function mostrarValores(){
      let formularioJS = document.forms['formulario'];
      let texto = '';
      let nombre = formularioJS['nombre'];//a travez del value
      let apellido = formularioJS['apellido'];// de esta manera recuperamos todo el objeto
      texto = nombre.value + '<br/>' + apellido.value;
      //Aca debemos pasar el value, para indicar que queremos obtener en este caso el nombre y el apellido
      document.getElementById('valores').innerHTML = texto;
    }
  </script>
</body>

```

Aca vemos como rescatamos los valores con value y a su vez los mostramos en la pagina web con el parámetro del div

Uso de documentWrite:

Hay que tener cuidado porque document.write puede sobrescribir todo el archivo

```
<h1>Manejo DOM con JavaScript</h1>
<button onclick="mostrar()">Mostrar</button>
<br/>
<script>
  document.write('Saludos desde JavaScript');

  function mostrar(){
    document.write('Adios');
  }
</script>
```

Este método sobrescribe todo lo que ve. Hay que tener cuidado

Mas usos de getElementById con este elemento podemos modificar lo que vemos en el html

Por ejemplo al hacer click se llama a la función mostrar que a su vez es un botón el llamado de la función hace un document.getElementById con titulo o sea que agarra el titulo del h1 y lo accede para luego modificarlo si en lugar del += hubiese un = . pero el += lo que hace es que nos concatena con el nuevo titulo. Ya que sino le estamos diciendo una igualdad en lugar de una concatenacion

```
<body>
  <h1 id='titulo'>Manejo DOM con JavaScript</h1>
  <button onclick="mostrar()">Mostrar</button>
  <br/>
  <script>
    function mostrar(){
      document.getElementById('titulo').innerHTML += '<br/>Nuevo Título';
    }
  </script>
```

Como modificar elementos del html

## Modificar una imagen en HTML

```
<body>
  <h1>MODIFICAR IMAGEN</h1>
  <hr/>
  
  <br>
  <button onclick='modificador()'>Cambiar Imagen</button>
  <script>
    function modificador(){
      document.getElementById('imagen').src = 'html5.png';
    }
  </script>
</body>
</html>
```

Podemos ver que ahora es .src ya que este contiene el atributo que nosotros queremos acceder y no podemos aplicar al innerhtml porque no hay nada dentro de la etiqueta ya que es algo vacío

Y así se puede modificar un escrito con CSS

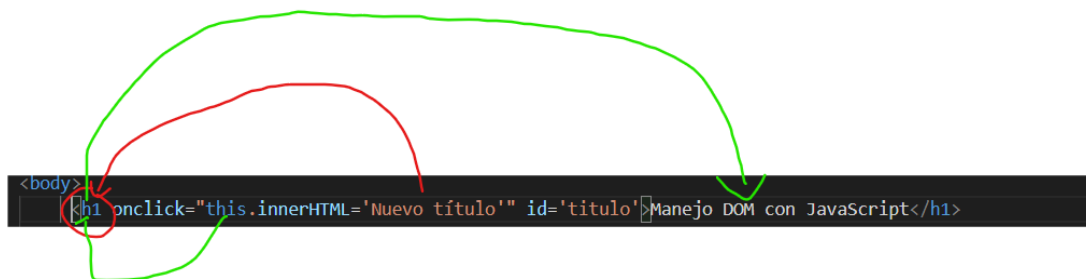
Modificar elemento de CSS con JS

```
<body>
  <h1 id="titulo" style="color: aqua;">MODIFICAR IMAGEN</h1>
  <hr/>
  
  <br>
  <button onclick='modificador()'>Cambiar Imagen</button>
  <!-- <script>
    function modificador(){
      document.getElementById('imagen').src = 'html5.png';
    }
  </script> -->
  <script>
    function modificador(){
      document.getElementById('titulo').style.color = 'red';
    }
  </script>
</body>
```

Como capturar el elemento directamente: Como modificar en la misma línea un elemento

Al hacer click on click on especifica que es un método nada mas que eso. Por ultimo vemos que al hacer clic en menjo de dom con js este clic va ejecutar la función this que this apunta a la etiqueta h1 y h1 hace referencia al tituo entonces accedemos con el innerhtml y le indicamos el valor que va a tener ahora . como puede ser esto también se lo podría cambiar de color con

This.style.color= red



```
<body>
<h1 onclick="this.innerHTML='Nuevo título' id='titulo'" Manejo DOM con JavaScript</h1>
```



```
<h1 onclick="cambiarTexto(this)">Manejo DOM con JavaScript</h1>
<br/>
<script>
  function cambiarTexto(titulo){
    console.log(titulo);
    console.log(titulo.innerHTML);
    titulo.innerHTML = 'Cambiamos el título';
    console.log(titulo.innerHTML);
  }
</script>
```

Una forma mas segura y también eficiente seria esta.

Funcion onload = “se aplica una función ()” y esto se aplica en el body

```

<body onload="entrar()">
  <h1 id='titulo'>Manejo DOM con JavaScript</h1>
  <br/>
  <div id='mostrar'></div>
  <script>
    function entrar(){
      alert('Entrando a la página web');
      let texto = '';
      if(navigator.cookieEnabled){
        texto = 'Cookies están habilitadas';
      }
      else{
        texto = 'Cookies están inhabilitadas';
      }
      document.getElementById('mostrar').innerHTML = texto;
    }
  </script>

```

La función se carga al inicio de la pagina y luego de esto se realiza el análisis si tiene las cookies almacenadas.

Onchange esto sirve para cambiar un formato en una propiedad

```

<body>
  <h1 id='titulo'>Manejo DOM con JavaScript</h1>
  <br/>
  Nombre: <input type="text" onchange="convertir(this)"/>
  <script>
    function convertir(nombreInput){
      nombreInput.value = nombreInput.value.toUpperCase()
    }
  </script>
</body>
</html>

```

De este modo no hace falta usar documentbyid sino que le paso con el this los datos del elemento en este caso nombreinput o nombre solamente seria suficiente y luego hago una igualdad para cambiar el valor dps del cambio que se detecta por la función onchange

Función onmouseover y función onmouseout. Sirve para que genere efectos cuando nos posamos arriba del elemento y cuando nos salimos de el

```

<body>
  <h1 id='titulo' onmouseover="rojo(this)" onmouseout="azul(this)">
    Manejo DOM con JavaScript
  </h1>
  <br/>
  <script>
    function rojo(titulo){
      titulo.style.color = 'red';
    }
    function azul(titulo){
      titulo.style.color = 'blue';
    }
  </script>

```

Algo parecido sucede con onmousedown y con onmouseup. Esto sería cuando hago clic sobre el título y cuando suelto el botón cambia a azul.

También se puede aplicar al evento onclick siempre se pasa como elemento el this. El this indica el valor de la etiqueta. Si decimos this nos referimos a la clase en la que está dicho dis en este caso titulo o h1

Onfocus y onblur esto sirve para editar el fondo donde trabajamos.

```

Nombre: <input type="text" onfocus="cambiar(this)" onblur="regresar(this)"/>
<br/><br/>
Apellido: <input type="text" onfocus="cambiar(this)" onblur="regresar(this)"/>
<script>
  function cambiar(elementoInput){
    elementoInput.style.background = 'yellow';
  }
  function regresar(elementoInput){
    elementoInput.style.background = 'white';
  }
</script>

```

Addeventlistener:

```

<body>
  <h1 id='titulo'>Manejo DOM con JavaScript</h1>
  <br/>
  Nombre: <input type="text" id='nombre' />
  <br/><br/>
  Apellido: <input type="text" id='apellido' />
  <script>
    document.getElementById('nombre').addEventListener('focus', cambiar);
    document.getElementById('nombre').addEventListener('blur', regresar);

    document.getElementById('apellido').addEventListener('focus', cambiar);
    document.getElementById('apellido').addEventListener('blur', regresar);

    function cambiar(evento){
      let componente = evento.target;
      componente.style.background = 'yellow'; //formas de asociar el elemento que queremos cambiar y el evento que va a generar
    }
    function regresar(evento){
      evento.target.style.background = 'white'; //segunda forma
    }
  </script>

```

Esto sirve para cambiar elementos como antes pero desde js y solo teniendo el id. Y agregando un evento. Luego pasamos dentro del eventlistener el valor de lo que deseamos hacer sin el on. Por ejemplo focus para cambiar el color interno cuando estamos sobre la caja de texto y

luego lo mismo para cuando salimos de la caja. Luego agregaos dichas funciones. Con `evento.target` vamos asociarnos a que elemento estamos editando

La función puede o no recibir un parámetro

Y este parametro puede ser cualquier cosa. Pero siempre va a apuntar a donde agregamos la función por ejemplo definimos el element by id con nombre y luego le decimos que ahaga el evento listener y luego le pasamos sin el on el evento que queremos que realice por ejemplo focus para que pinte el interior cuando estemos sobre esa caja que indicamos con el id y luego le pasamos el parámetro de la función que vamos a definir y sin () y luego definimos dicha función

De este modo, también se le puede aplicar una función flecha

```
document.getElementById('nombre').addEventListener('focus', (evento)=>{
    evento.target.style.background = 'pink';
});
document.getElementById('nombre').addEventListener('blur', (evento)=>{
    evento.target.style.background = '';
});

document.getElementById('apellido').addEventListener('focus', cambiar);
document.getElementById('apellido').addEventListener('blur', regresar);
```

Aca vemos para nombre y no para apellido la utilizacion de función flecha el único problema de esto es que no es reutilizable como el anterior ya que se aplica directamente sobre dicha elemento .

Las otras son funciones callback para apellido ya que llaman a una función



## Delegacion de eventos

Utilizando la función flecha y trabajando dentro de un form para que todo lo que este allí dentro sea aplicado todo junto y no por separado.

Por ultimo también debemos pasar dps de la función flecha el parámetro de true. Para que esta se aplique a los sub elementos del form o sea para el input text de apellido y nombre

```
<body>
  <h1 id='titulo'>Manejo DOM con JavaScript</h1>
  <br/>
  <form id='forma'>
    Nombre: <input type="text" id='nombre' />
    <br/><br/>
    Apellido: <input type="text" id='apellido' />
  </form>

  <script>
    const forma = document.getElementById('forma');

    forma.addEventListener('focus', (evento)=>{
      evento.target.style.background = 'yellow';
    }, true);
    forma.addEventListener('blur', (evento)=>{
      evento.target.style.background = '';
    }, true);

  </script>
```

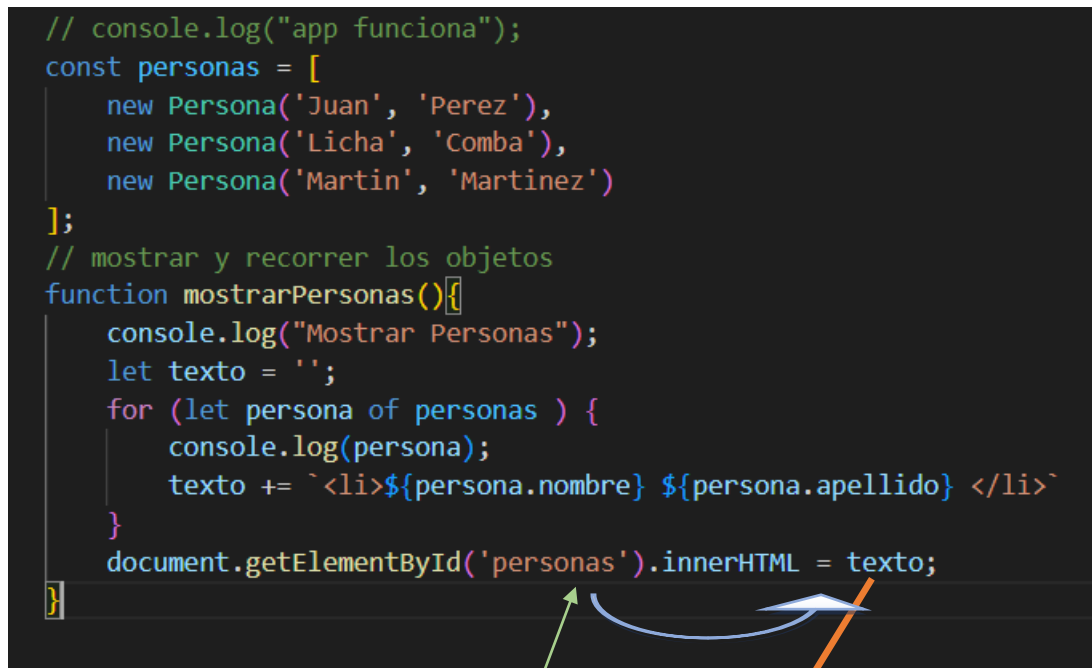
## CALCULADORA EN JS

Div class=container es importante poner todo el documento html dentro de esta

Como agregar elementos a un formulario

El array es porque seria como agregar todas juntas dichas personas y se vayan almacenando dentro de un vector

```
// console.log("app funciona");
const personas = [
  new Persona('Juan', 'Perez'),
  new Persona('Licha', 'Comba'),
  new Persona('Martin', 'Martinez')
];
// mostrar y recorrer los objetos
function mostrarPersonas(){
  console.log("Mostrar Personas");
  let texto = '';
  for (let persona of personas ) {
    console.log(persona);
    texto += `<li>${persona.nombre} ${persona.apellido} </li>`
  }
  document.getElementById('personas').innerHTML = texto;
}
```



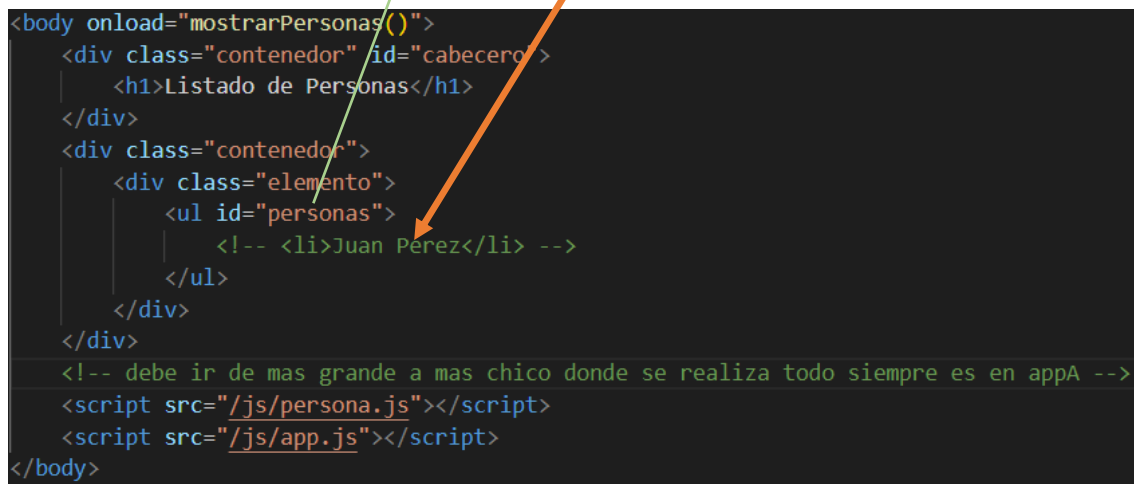
Con función mostrar personas voy a poder cargar el formulario con las personas que están almacenadas en la clase

Persona se encarga de recorrer el arreglo que defini arriba de personas y luego texto que es una variable pre inicializada podre ir concatenando los elementos. Esto seria como

Texto = texto + ( persona. Nombre) + (persona.apellido). entonces por el for va agregando cada una

Fuera del form me encargo de igualar la variable del documento html y accediendo a la instancia interna del html con id personas me encargo de cargarlo

```
<body onload="mostrarPersonas()">
  <div class="contenedor" id="cabecero">
    <h1>Listado de Personas</h1>
  </div>
  <div class="contenedor">
    <div class="elemento">
      <ul id="personas">
        <!-- <li>Juan Perez</li> -->
      </ul>
    </div>
  </div>
  <!-- debe ir de mas grande a mas chico donde se realiza todo siempre es en appA -->
  <script src="/js/persona.js"></script>
  <script src="/js/app.js"></script>
</body>
```



El onload siempre va en el body y este es el que me va a permitir a mi poder trabajarlo con js

Como agregar un elemento o un texto o un botón al centro

```
<!-- cursor: pointer para que se ponga la mano sobre el boton -->
```

Como centrar un elemento div. Ejemplo de clase reloj

```
#contenedor.reloj-contenedor{  
  background-color: rgb(12, 119, 206);  
  padding: 25px;  
  max-width: 350px;  
  text-align: center;  
  border-radius: 5px;  
  margin: 0 auto;  
}
```

Con margin 0 parte superior y también parte inferior y auto para la parte izquierda y derecha.

```
const mostrarReloj = () =>{  
  let fecha = new Date();  
  let hora = formatoHora(fecha.getHours());  
  let minutos = formatoHora(fecha.getMinutes());  
  let seg = formatoHora(fecha.getSeconds());  
  document.getElementById('hora').innerHTML = `${hora}:${minutos}:${seg}`;  
  
  let mes = new Date();  
  let dia = formatoAño(mes.getUTCDay());  
  let meses = formatoAño(mes.getUTCMonth());  
  let año = formatoAño(mes.getUTCFullYear());  
  document.getElementById('fecha').innerHTML = `${dia}/${meses}/${año}`;  
  
  document.getElementById('contenedor').classList.toggle('animar');  
  // classlist trae todas las clases de ese contenedor y una de sus clases es animar  
  // toggle : te desactiva una clase relacionada  
}  
  
const formatoHora = (horas) =>{  
  if(horas <10){  
    horas = '0' + horas;  
  }return horas;  
}  
  
const formatoAño = (años) =>{  
  if(años <10){  
    años = '0'+años;  
  }return años;  
}  
  
setInterval(mostrarReloj,1000);
```

Aca también vemos como obtener de un elemento otros mas