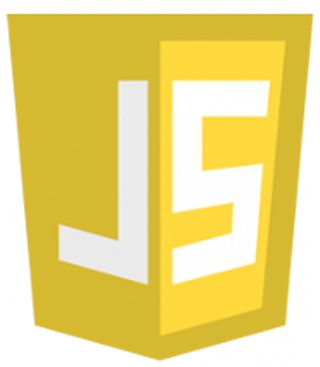
Javascript y jQuery

# Javascript

El lenguaje



PAV2 - UTN FRC

Lo más difícil de usar JavaScript

* El problema puede ser la implementación de DOM del browser
* Los motores de JavaScript son diferentes Estrategias
* Escribir código compatible con los estándares
* Escribir código que tenga en cuenta el entorno (complicado → mucho código)

JavaScript puede no estar disponible

* Deshabilitado o no disponible en el browser PAV2 - UTN FRC
* Google Chrome DevTools / Firebug
* Visual estudio Code
* Extensiones: prettier, esLint , Quokka, liveServer
* Mozilla Developer Network: <https://developer.mozilla.org/es/docs/Web/JavaScript>
* Es clave a la hora de usar JavaScript en la página
* Dos formas de usarlo:
  + Inline

**<script>**

**console.log('Hola pagina');**

**</script>**

* + Archivo JavaScript externo

**<script src="jquery-1.7.1.js" />**

* De forma opcional se puede especificar el contenido
  + text/javascript
* Se pueden incluir tantos scripts como sea necesario

### write()

**<script language=javascript> document.write("¡Hola Mundo!");**

**</script>**

alert()

**<script language=javascript> alert("¡Hola Mundo!");**

**</script>**

Hola Mundo utilizando el elemento script

JavaScript No es un lenguaje fuertemente tipado (strongly typed)

Javascript soporta los siguientes tipos de datos core:

* string
* number (integer y float)
* boolean
* null
* undefined
* object

#### Diferencias entre Null y Undefined

Var x; //esto devolverá undefined , esta declarada pero no inicializada.

Var x=null; //devuelve null , esa inicializada.

Amas están vacías, pero undefined no tiene asignado nada

Hay otros tipos de datos, pero no pertenecen al core sino que son especializaciones del tipo object, se los denomina built-in objects:

* String
* Boolean
* Number
* Date
* Array
* Object
* Function
* RegEx
* Error

Crear una variable

**var a = "cadena"; var b = 15;**

**var c = 12.3, d = "otra cadena"**

Asignar una variable

**a = "nueva cadena"; b = 2555;**

## Variables

CONTENIDO ADICIONAL

¿Y si no pongo var?

**nueva = 25;**

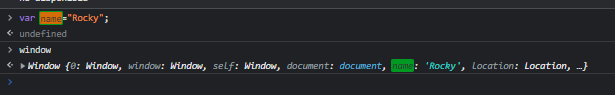
En el ámbito global no hay diferencias

En el ámbito local de una función si ponemos var creará una variable local, de lo contrario buscará en la cadena de ámbitos y cuando la encuentre asignará esa variable. Si no la encuentra crea la variable global.

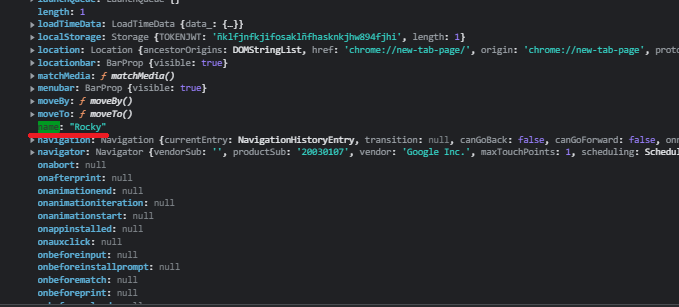
PAV2 - UTN FRC

#### No se recomienda el uso de Var

Cuando una variable es declarada como Var,su ámbito pertenece al global (Scope).

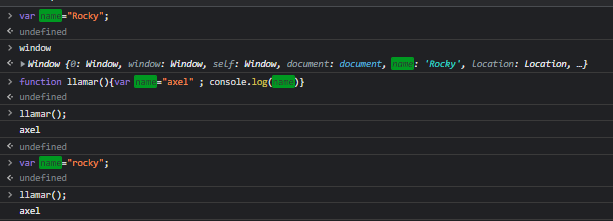


La declaramos y usamos el window y vemos que esta en el ámbito global



*Para que estos problemas no pasen usar let en vez de var*

Puede ocurrir que sea modificada ,por medio de la asignación y que no devuelva el resultado esperado.



typeof es un operador que devuelve el tipo básico de una variable

**var a = 25;**

**typeof 25; /\* devuelve: "number" \*/**

La propiedad constructor devuelve el constructor de la variable y puede servir para tener información más precisa sobre el tipo de variable de que se trata.

**var b = new Date(2017,01,01); typeof b; /\* devuelve: "object" \*/**

**b.constructor; /\* devuelve function Date() { [native code] } \*/**

En JavaScript la conversión de tipos se puede dar por:

* Explicitamente: Usando una función de conversión
* Implicitamente: por JavaScript automáticamente

Se pueden convertir datos a string usando:

* la función global: String()
* el método .toString()

**String(a); String(123); x.toString(); (123).toString();**

**String(false); String(true); false.toString(); true.toString();**

**String(Date());**

**Date().toString();**

Se pueden convertir datos a number usando:

* la función global: Number()
* el método .parseInt()
* el método .parseFloat()
* el operador unario +

**Number("210"); /\* devuelve 210 \*/ Number("21ax"); /\* devuelve NaN \*/**

**parseInt("21ax") /\* devuelve 21 \*/ parseInt("aa") /\* devuelve NaN \*/**

**parseFloat("12.21") parseFloat("15.aa") /\* devuelve 15 \*/**

**+"32";**

**+"21a" /\* devuelve NaN \*/**

**Number(false)**

**Number(true)**

**/\* devuelve 0 \*/**

**/\* devuelve 1 \*/**

**d = new Date();**

**Number(d)**

**/\* devuelve 1404568027739 \*/**

|  |  |
| --- | --- |
| + | suma |
| - | resta |
| \* | producto |
| % | módulo |
| ++ | incremento |
| -- | decremento |

|  |  |
| --- | --- |
| && | and |
| || | or |
| ! | not |
|  |  |
| ?: | operador ternario |

|  |  |
| --- | --- |
| == | igualdad |
| != | desigualdad |
| > | mayor |
| < | menor |
| >= | mayor o igual |
| <= | menor o igual |
| === | estrictamente iguales |
| !== | no estrictamente iguales |

**} else {**

*código que se ejecuta si la condición es falsa*

**}**

**if (***condicion***) {**

*código que se ejecuta si la condición es verdadera*

**}**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| if | (*condicion*) | { |  | | | | | |
|  | *código que* | *se* | *ejecuta* | *si* | *la* | *condición* | *es* | *verdadera* |

if (*condicion1*) {

*código que se ejecuta si la condición1 es verdadera*

} else if (*condicion2*) {

*código que se ejecuta si la condición1 es falsa y la condición2 es verdadera*

#### } else {

*código que se ejecuta si la condición1 es falsa y la condición2 es falsa*

#### }

**switch(***expresión***) { case n:**

*código*

**break; case m:**

*código*

**break; default:**

*código por defecto*

**}**

**for (***sentencia1; sentencia2; sentencia3***) {**

*código*

**}**

**for (var i = 0; i < Cosas.length; i++) { Cosas[i]**

**}**

**/\* Recorre las propiedades de un objeto \*/**

**for (var propiedad in Cosa) { Cosa[propiedad]**

**}**

**while (***condición***) {**

*código*

**}**

**do {**

*código*

**} while (***condición***);**

break: sale de la iteración y continúa ejecutando la sentencia siguiente, si la hay:

**break;**

continue: omite la iteración corriente y continúa la iteración siguiente en el caso de que la condición del bucle se satisfaga.

**continue;**

Array unidimensional let array=[]; Array multidimensional=[[fila] ,[columna]]

Sintaxis:

**var nombre-array = [item1, item2, ...];**

Ejemplo:

**var colores = ["Verde","Rojo","Azul"];**

Acceder a los elementos de un array:

**var color1 = colores[0]; colores[2] = "Amarillo";**

Los arrays son una clase especial de objeto cuyas propiedades son valores numéricos.

Como reconocer un array:

**Array.isArray(colores); // Solo en ECMAScript 5 o sup**

**colores.constructor.toString().indexOf("Array") > -1;**

**colores instanceOf Array;**

length: devuelve el tamaño del array push(): agrega un elemento al final pop(): remueve el último elemento

shift(): remueve el primer elemento y mueve el resto de los elementos

unshift(): agrega un elemento al principio del array

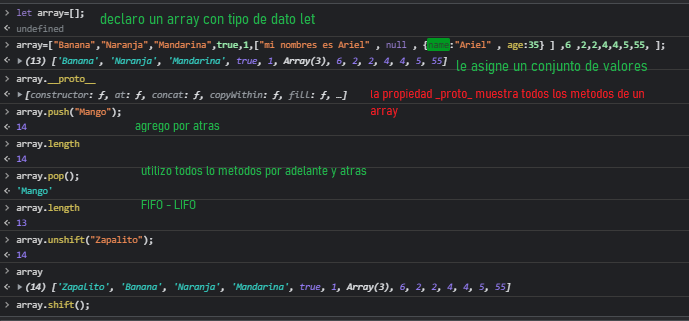
delete: elimina un elemento (dejando un elemento undefined)

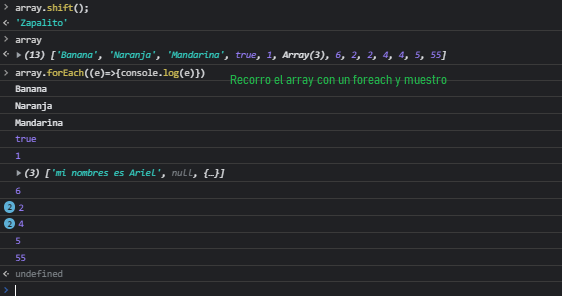
**delete colores[1];**

splice(:posicion, elementos\_a\_remover, nuevo1, nuevo2,...): agrega elementos a un array removiendo aquellos específicados

sort(): ordena el array

reverse(): invierte el orden de los elementos del array



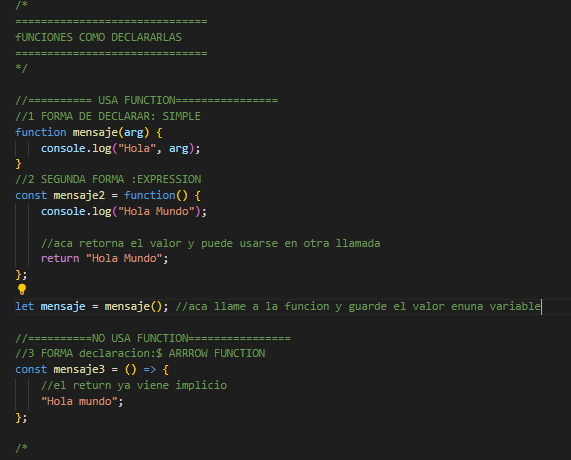


Funciones Anonimas o callBack

Es aquella que no tiene nombre, se utilizan dentro de otra función

Let color=[“rojo”,”verde”,”azul” ];

Color.foreach(function(e)=>console.log(e))



Para invocar una función se utiliza el operador ()

**var a = mensaje2();**

Se puede asignar una función como cualquier objeto

**var c = function(a,b){return a+b};**

**d = c(5,6); /\* d contendrá 11 \*/**

Par clave: valor

Se puede definir un objeto asignando sus propiedades y métodos

**var persona = new Object(); persona.nombre = "Juan"; persona.apellido = "Perez; persona.nombreCompleto = function(){**

**return this.nombre + " " + this.apellido;**

**};**

También se puede declarar y definir mediante literales

**var auto = {marca:"Ford", modelo:"Focus", año:2016};**

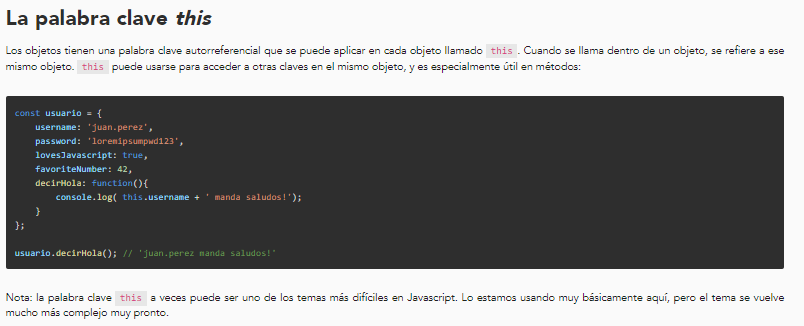
Se puede acceder a las propiedades mediante la notación punto

**var m = auto.marca; auto.marca = "Fiat";**

O usando corchetes [ ]

*Nota: el uso por punto solo es válido cuando conoces el nombre en forma literal de la propiedad*

**var m = auto["marca"]; auto["marca"] = "Fiat";**

****

Cuando se crea el execution context, el interprete reserva el espacio de memoria para las variables (hoisting), guarda la referencia al outer enviroment y además “setea” la variable this. Esta variable va a apuntar a distintos objetos dependiendo en cómo fue invocada la función.

* **GLOBAL CONTEXT**

Este es el caso cuando ejecutamos código en el contexto global (afuera de cualquier función). En este caso this hace referencia al objeto global, en el caso del browser hace referencia a window.

* **FUNTION CONTEXT**

Cuando estamos dentro de una función, el valor de this va a depender de cómo sea invocada la función.

* **SIMPLE CALL**

En este caso, el interprete le da a this una referencia al objeto global.

function a() {

function b() {

return this;

}

return b();

}

console.log(a());

* **MÉTODO DE UN OBJETO**

Si tenemos una función como propiedad de un objeto, la palabra clave **this** va a hacer referencia al objeto.

//--------------EJEMPLO 1---------------

var o = { prop: 37, f: function() {return this.prop;} };

console.log(o.f()); // logs 37

// this hace referencia a `o`

//--------------EJEMPLO 2---------------

var o = {prop: 37};

// declaramos la función

function loguea() {return this.prop;}

//agregamos la función como método del objeto `o`

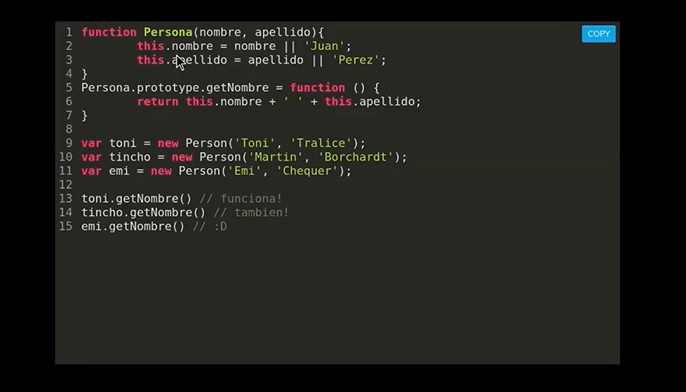
o.f = loguea;

console.log(o.f()); // logs 37

// el resultado es le mismo!

***El this depende mucho del contexto donde se este ejecutando , puede ser un método de un objeto , en el ámbito global o el de una función local .***

**Prototype**

****

## **Throw, and Try...Catch...Finally**

La try declaración define un bloque de código para ejecutar (para probar).

La catch declaración define un bloque de código para manejar cualquier error.

La finally declaración define un bloque de código para ejecutar independientemente del resultado.

La throw instrucción define un error personalizado.

**function myFunction() {  
  const message = document.getElementById("p01");  
  message.innerHTML = "";  
  let x = document.getElementById("demo").value;  
  try {  
    if(x == "") throw "is empty";  
    if(isNaN(x)) throw "is not a number";  
    x = Number(x);  
    if(x > 10) throw "is too high";  
    if(x < 5) throw "is too low";  
  }  
  catch(err) {  
    message.innerHTML = "Error: " + err + ".";  
  }  
  finally {  
    document.getElementById("demo").value = "";  
  }  
}**

JSON (se pronuncia Jason) viene de JavaScript Object Notation

* Es un formato de intercambio de datos independiente del lenguaje
* Es auto-descripto
* Es facil de entender

**{**

**"clientes":[**

**{"nombre":"Juan", "apellido":"Perez"},**

**{"nombre":"Pedro", "apellido":"Garcia"},**

**{"nombre":"Maria", "apellido":"Sanchez"}**

**]**

**}**

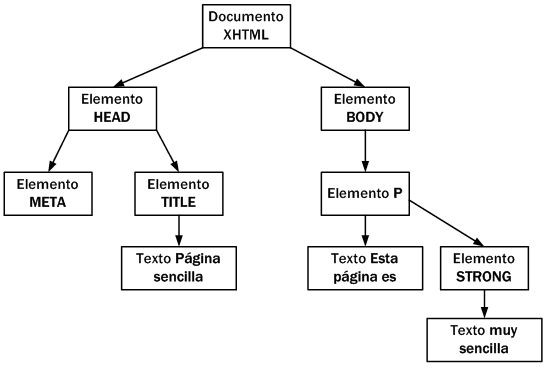
Sintaxis:

* Los datos están como pares nombre/valor
* Los datos se separan por coma
* Las llaves contienen objetos
* Los corchetes contienen arrays

Convertir texto JSON a un objeto JavaScript

**var objeto = JSON.parse(texto);**

DOM (Document Object Model) Modelo de Objetos del Documento

* Es una API para documentos validos HTML y bien construidos XML
* Permite el acceso a la estructura de una página HTML mediante el mapeo de los elementos de esta página en un árbol de nodos.
* define:
  + Los elementps HTML como objetos
  + Las propiedades de todos los elementos HTML
  + Los métodos para acceder a todos los elementos HTML
  + Los eventos para todos los elementos HTML

La especificación DOM define 12 tipos de nodo, pero los de uso más habitual son:

* Document: nodo raíz del que derivan todos los demás nodos del árbol
* Element: representa cada una de las etiquetas HTML
* Attr: representa cada uno de los atributos de los elementos HTML
* Text: contiene el texto encerrado por una etiqueta HTML.
* Comment: representa cada comentario en el documento.

getElementById()

**var cabecera = document.getElementById("cabecera");**

getElementsByTagName()

**var parrafos = document.getElementsByTagName("p"); var primerParrafo = parrafos[0];**

getElementsByClassName()

**var parrafos = document.getElementsByClassName("resaltado");**

innerHTML

**document.getElementById("cabecera").innerHTML = "Cabecera";**

Atributos

**document.getElementById("foto").src = "caripela.jpg"; document.getElementById("btnOk").value = "Aceptar";**

Estilos CSS

**document.getElementById("p2").style.color = "blue"; document.getElementById("demo").style.textAlign = "center";**

**document.getElementById("btnFecha").onclick = mostrarFecha;**

**function mostrarFechaDate() { document.getElementById("txtFecha").innerHTML = Date();**

**}**

**<div onmouseover="mOver(this)" onmouseout="mOut(this)" style="background-color:red">**

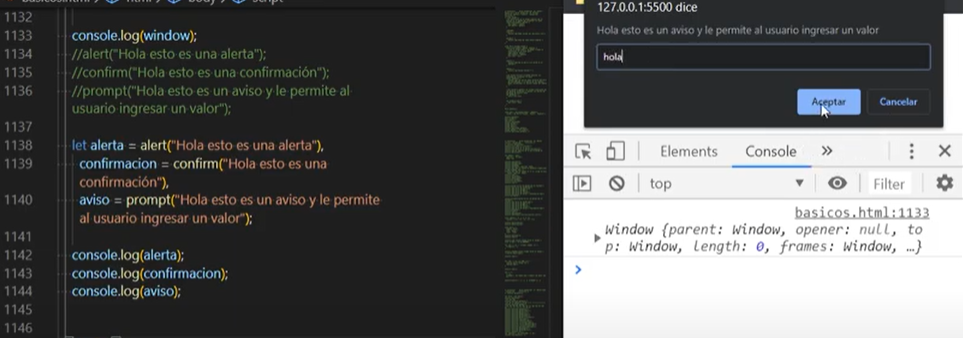
**Pasar el mouse</div>**

**<script>**

**function mOver(obj) {obj.innerHTML = "Excelente!"} function mOut(obj) {obj.innerHTML = "Pasar el mouse"}**

**</script>**

**Clase window alert y Confirm**

****

Cargar un listado de artículos contenidos en un JSON al presionar un botón en el documento HTML

## **ES6-ES7**

**KEYWORD *LET* Y *CONST***

**function f() {**

**{**

**let x;**

**{ // okay, block scoped name**

**const x = "sneaky";**

**x = "foo";// error, const**

**}**

**let x = "inner"; // error, already declared in block**

**}**

Las dos palabras ***let*** y ***const*** NO tienen *hoisting* y tienen *scope* de bloque. Esto último quiere decir que cuando declaramos en un mismo bloque un *let* y queremos modificar su contenido, la consola tirará error. Los mismo ocurre con las *constantes*

***LET* Y *VAR* EN LOS CICLOS FOR**

**function crearFuncion() {**

**var arreglo = [];**

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

**arreglo.push(function(){console.log(i);})**

**}**

**return arreglo;**

**}**

var arr = crearFuncion();

var let

arr[0] /\*3\*/ arr[0]//1

arr[1] /\*3\*/ arr[1]//2

arr[2] /\*3\*/ arr[2]//3

|  |
| --- |
| Esta declaración permite otorgarle un valor a cada variable de forma ordenada entre corchetes.  Para saltearnos algunos valores debemos pasar ***espacios vacíos***.  var obj = {  name: "Alejo",  lastname: "Bengoechea",  job: {  daytime: "alumno",  nighttime: "batman"  }  }  var {name, job:{daytime}} = obj  //Alejo, alumno |

**DESTRUCTURING (ES6)**

var [a, b, c] = [1, 2, 3]

//a = 1

//b = 2

//c = 3

var [d, , f] = [4, 5, 6, 7, 8]

//d = 4

//f = 6

**OBJECTOS LITERALES (ES6)**

var apellido = "Bengoechea";

var objetito = {

saludo: "hola",

saludar() {console.log("buen dia"),

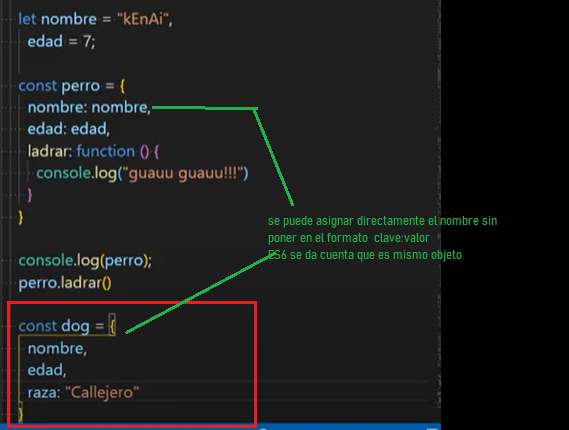
apellido,

["prop" + "dinamica"]: 10

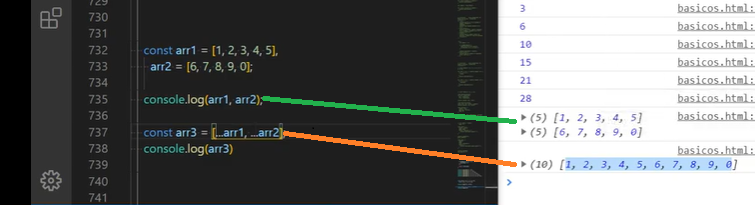
}

Los objetos tienen una estructura de clave-valor. Cuando queremos que una de las propiedades sea un método, no hace falta pasarle una clave.

A su vez, cuando queremos insertar una variable en el objeto, solo basta pasarle como parámetro el nombre de la variable

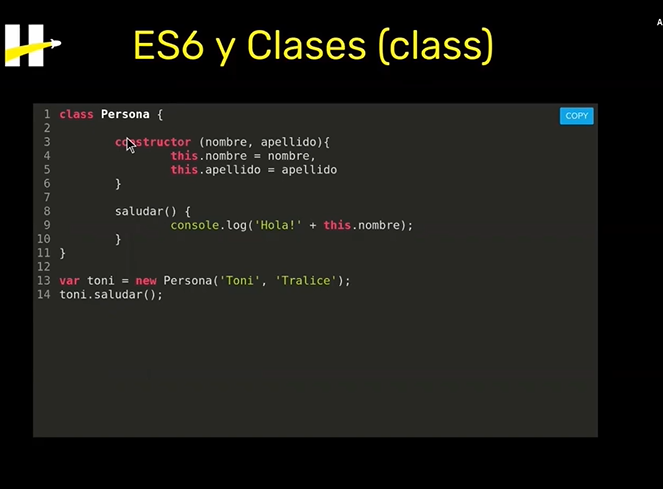


**Operador Spreed**



**Arraw Function**

const sumar=(a,b)=>a+b; //viene implícito el return y no lleva la palabra function



class Persona {

    constructor(nombre, apellido) {

        (this.nombre = nombre), (this.apellido = apellido);

    }

    saludar() {

        console.log("Hola!" + this.nombre);

    }

}

//var toni = new Persona("Toni", "Tralice");

//toni.saludar();

//ahora aplicamos herencia

class Empleado extends Persona {

    constructor(nombre, apellido, empleo, sueldo) {

        super(nombre, apellido);

        this.empleo = empleo;

        this.sueldo = sueldo;

    }

}

var toni = new Empleado("Toni", "Tralice", "Profesor", 100);

toni.saludar();

function crearClasePersona() {

    class Persona {

        constructor(nombre, edad, hobbies, amigos) {

            // El constructor de la clase Persona recibe nombre (string), edad (integer), hobbies (array de strings), amigos (array de objetos)

            // Inicializar las propiedades de la persona con los valores recibidos como argumento

            this.nombre = nombre;

            this.edad = edad;

            this.hobbies = hobbies;

            this.amigos = amigos;

            // Tu código aca:

        }

addFriend(nombre, edad) {

// El método 'addFriend' recibe un string 'nombre' y un entero 'edad' y debe agregar un objeto:

            // { nombre: nombre, edad: edad} al arreglo de amigos de la persona.

            // No debe retornar nada.

            // Tu código aca:

            const obj = { nombre: nombre, edad: edad };

            this.amigos.push(obj);

        }

        addHobby(hobby) {

   // El método 'addHobby' recibe un string 'hobby' y debe agregarlo al arreglo de hobbies de la persona.

            // No debe retornar nada.

            // Tu código aca:

            this.hobbies.push(hobby);

        }

   getFriends() {

// El método 'getFriends' debe retornar un arreglo con sólo los nombres del arreglo de amigos

            // de la persona.

            // Ej:

            // Suponiendo que la persona tiene estos amigos: [{nombre: 'martin', edad: 31},{nombre: 'toni', edad: 33}]

            // persona.getFriends() debería devolver ['martin', 'toni']

            // Tu código aca:

            var arr2 = [];

            arr2 = this.amigos.map(function(a) {

                return a["nombre"];

            });

            return arr2;

        }

getHobbies() {

// El método 'getHobbies' debe retornar un arreglo con los hobbies de la persona

            // Ej:

            // persona.getHobbies() debe devolver ['correr', 'dormir', 'nadar']

            // Tu código aca :

            return this.hobbies;

        }

getPromedioEdad() {

  // El método 'getPromedioEdad' debe retornar el promedio de edad de los amigos de una persona

            // Ej:

            // Si la persona tuviera estos amigos:

            // {

            //   amigos: [{

            //     nombre: 'toni',

            //     edad: 33,

            //   }, {

            //     nombre: 'Emi',

            //     edad: 25

            //   }]

            // }

            // persona.getPromedioEdad() debería devolver 29 ya que (33 + 25) / 2 = 29

            // Tu código aca:

            var suma = 0;

            var cont = 0;

            for (let a of this.amigos) {

                suma += a.edad;

                cont++;

            }

            return suma / cont;

        }

    }

    return Persona;

}

**TEMPLATE STRINGS**

**var b = "hola**

**como estas"**

**var a = "Hola \*\*\\n\*\* como estas"**

Existen algunas restricciones en JavaScript. Por ejemplo, so se puede hacer un new line en el medio de un string. **\n** sirve para hacer este cometido.

**var z = `Hola como estas`**

**var q = `Hola ${name} como estas? Hoy es ${(() => 10 + 6)()} de febrero`**

**//Hola Juan como estas? Hoy es 16 de febrero.**

Las comillas reversas (alt + 96) son ***literal strings***.

La estructura **&{*algo*}** sólo sirve cuando se lo utiliza entre comillas invertidas.

|  |  |
| --- | --- |
| [**Babel**](https://babeljs.io/)  es una plataforma en la que se hace una transcompilación (o traducción) de JavaScript gratuito y de código abierto que se utiliza principalmente para convertir el código ECMAScript 2015+ en una versión de JavaScript compatible con versiones anteriores que pueden ejecutar motores de JavaScript más antiguos. |  |

**Tema Adicional Clase Date**

//creo un objeto

miFecha = new Date()

Para crear un objeto fecha con un día y hora distintos de los actuales tenemos que indicar entre paréntesis el momento con que inicializar el objeto. Hay varias maneras de expresar un día y hora válidas, por eso podemos construir una fecha guiándonos por varios esquemas. Estos son dos de ellos, suficientes para crear todo tipo de fechas y horas.

miFecha = new Date(año,mes,dia,hora,minutos,segundos)

miFecha = new Date(año,mes,dia)

Los objetos de la clase Date no tienen propiedades pero si un montón de métodos, vamos a verlos ahora.

**getDate()** Devuelve el día del mes.

**getDay()** Devuelve el día de la semana.

**getHours()** Retorna la hora.

**getMinutes()** Devuelve los minutos.

**getMonth()** Devuelve el mes (atención al mes que empieza por 0).

**getSeconds()** Devuelve los segundos.

**getTime()**Devuelve los milisegundos transcurridos entre el día 1 de enero de 1970 y la fecha correspondiente al objeto al que se le pasa el mensaje.

**getYear()**

Retorna el año, al que se le ha restado 1900. Por ejemplo, para el 1995 retorna 95, para el 2005 retorna 105. Este método está obsoleto en Netscape a partir de la versión 1.3 de Javascript y ahora se utiliza

**getFullYear()**

Retorna el año con todos los dígitos. Usar este método para estar seguros de que funcionará todo bien en fechas posteriores al año 2000.

**setDate()** Actualiza el día del mes.

**setHours()** Actualiza la hora.

**setMinutes()**Cambia los minutos.

**setMonth()**Cambia el mes (atención al mes que empieza por 0).

**setSeconds()**Cambia los segundos.

**SetTime()**

Actualiza la fecha completa. Recibe un número de milisegundos desde el 1 de enero de 1970.

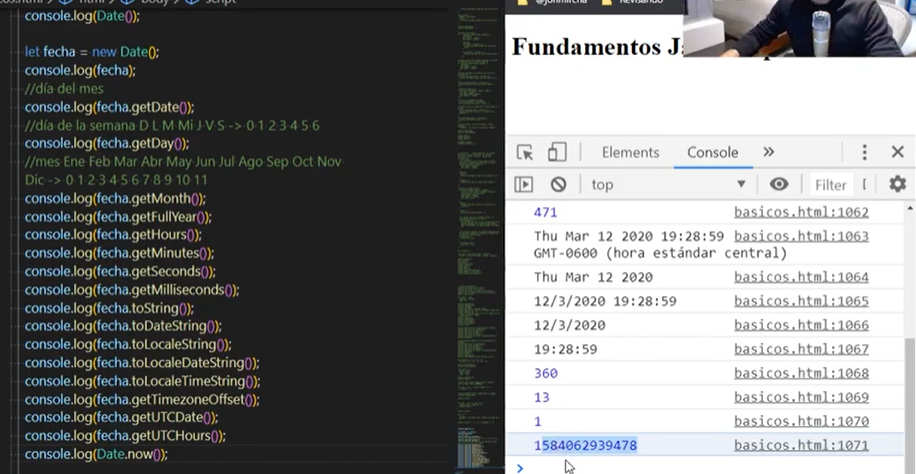
**setYear()**

Cambia el año recibe un número, al que le suma 1900 antes de colocarlo como año de la fecha. Por ejemplo, si recibe 95 colocará el año 1995. Este método está obsoleto a partir de Javascript 1.3 en Netscape. Ahora se utiliza setFullYear(), indicando el año con todos los dígitos.

**setFullYear()**

Cambia el año de la fecha al número que recibe por parámetro. El número se indica completo ej: 2005 o 1995. Utilizar este método para estar seguros que todo funciona para fechas posteriores a 2000.

Como habréis podido apreciar, hay algún método obsoleto por cuestiones relativas al año 2000: setYear() y getYear(). Estos métodos se comportan bien en Internet Explorer y no nos dará ningún problema el utilizarlos. Sin embrago, no funcionarán correctamente en Netscape, por lo que es interesante que utilicemos en su lugar los métodos getFullYear() y setFullYear(), que funcionan bien en Netscape e Internet Explorer.

****