JavaScript Puro

Tags apuntes

Como introducir en el código de mi página:

Dentro del html:

```
<script>
    alert('Hello World');
</script>
```

Archivo independiente:

```
<script src="/path/to/script.js"></script>
```

Meterlo en el Head con etiquetas:

defer:

```
<head>
    // resto de la cabecera
    <script src="/path/to/script.js" defer></script>
</head>
```

async:

```
<head>
    // resto de la cabecera
    <script src="/path/to/script.js" async></script>
</head>
```

Añadir imagen defer y async

Interacciones:

1. Alerta:

```
alert("Hola");
```

2. Petición (meter información):

```
let result = prompt("¿Cuántos años?", "")
///prompt(title, [default]);
```

3. confirm:

```
let result = confirm("Es cierto???")
// result == true or false
```

4. console.log

```
//imprimir en la consola -> usamos para debug
console.log("hola")
```

Statements

Declaraciones, a veces podemos saltarnos el punto y coma, recomendado no hacer porque a veces lo pone solo:

Ejemplos:

```
//funciona
alert('hola'); alert('Mundo');

alert('Hola')
alert('Mundo')

alert('Hola');
alert('Mundo');

console.log( 3+
1 +
2);

console.log( 3 +
1);

//no funciona
console.log( 3 +
1
    //da error aquí
+ 2 );
```

Reglas que siguen las ";" automáticas (ASI) → https://262.ecma-international.org/7.0/#sec-automatic-semicolon-insertion

Variables:

las variables son con let → sistema de tipado debil,

```
let message; console.log(typeof(message)) //undefined
message = 'Hello!'; console.log(typeof(message)) //string m
essage = 1234 console.log(typeof(message)) //number
```

No usar var → coñazo

Siempre declarar variables

```
a = 3 //funciona pero puede dar errores posteriores posteriores
//Sería una variable global
//no funcionaria en modo estricto
```

Riesgo de Sobreescribir Variables Globales: Al no declarar una variable con let, const o var, podrías sobrescribir una variable global sin darte cuenta, causando conflictos o errores difíciles de detectar.

2. **Difícil de Depurar y Mantener**: El código se vuelve menos claro porque los lectores del código no sabrán si la variable fue declarada intencionalmente o si es un error. Esto dificulta la lectura y el mantenimiento.

3. **Modo Estricto (**'use strict'**)**: Si tu código está en modo estricto (usando 'use strict'; al comienzo del script o la función), las variables no declaradas generan un error. Este modo evita asignaciones accidentales y hace que JavaScript sea más seguro y predecible.

Constantes:

```
const clave = "hola"
//no s va a poder reasignar valor
```

Tipos de datos:

Number:

- Tanto para integers como coma flotante
- Valores enteros entre –(253-1) y (253-1)
  - BigInt para valores fuera de ese rango (terminado en n)

    ```
    const bigInt = 1234567890123456789012345678901234567890n;
    ```

- Valores numéricos especiales
  - Infinity, -Infinity

    ```
    console.log(1/0); //Infinity
    console.log(-1/0); //-Infinity
    ```

```
let a = Infinity
```

- NaN (Not a number)

```
console.log("texto" / 2); //NaN
```

String:

no existe char

```
let x = "sadfb"
let y = 'sdcfvbgfds'

// no entiendo pero tiene sentido
let phrase = `can embed another ${str}`;
let text = `the result is ${1 + 2}`
```

Métodos:

```
console.log("Texto de prueba".length) // 15
console.log("Texto de prueba".charAt(2)) // x

const result = "Texto de prueba".split(" ")
console.log(result) // ['Texto', 'de', 'prueba']
const [p1, p2, p3] = "Texto de prueba".split(" ")

console.log("Texto de prueba".slice(6)) // de prueba
console.log("Texto de prueba".slice(6,8)) // de
console.log("Texto de prueba".slice(6,7)) // d
```

Más operaciones: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Boolean

```
//Declaración
let nameFieldChecked = true;
let ageFieldChecked = false;

//Pueden ser el resultado de una comparación
let isGreater = 4 > 1;
```

null

```
//Declaración
let age = null;
```

- No es una referencia a un objeto que no existe
- **Valor especial que representa nada, vacío, valor desconocido**

typeof devuelve object por compatibilidad histórica, aunque no lo sea:

```
typeof(null); //object
```

undefined

Representa que no se ha asignado ningún valor

```
let age; //declaración
console.log(age); //undefined
let nombre = undefined; //No deberíamos hacer esto
```

NaN

Si haces operaciones no permitidas con numeros etc en vez de error devuelve obj NotaNumber

```
console.log('a'*3) //NaN
```

Conversiones:

Usando String:

```
console.log(String("23")); //'23'
console.log(String(true)); //'true'
console.log(String(false)); //'false'
console.log(String(undefined)); //'undefined'
console.log(String(null)); //'null'
```

Usando Number

```
console.log(Number("23") + 1); //24
console.log(Number(" 23 ")); //23
console.log(Number("")); //0
console.log(Number("trece")); //NaN
console.log(Number(true)); //1
console.log(Number(false)); //0
console.log(Number(undefined)); //NaN
console.log(Number(null)); //0
```

también se puede hacer con el '+':

```
let apples = "2";
let oranges = "3";
console.log(+apples + +oranges); //5
```

Usando Boolean:

```
console.log(Boolean(0)); //false
console.log(Boolean(1)); //true
console.log(Boolean(11234)); //true
console.log(Boolean("")); //false
console.log(Boolean("Hola")); //true
console.log(Boolean("0")); //true
console.log(Boolean(" ")); //true
console.log(Boolean(null)); //false
console.log(Boolean(undefined)); //false
```

Conversiones explicitas y cosas raras:

```
console.log("2" + 2); //22
console.log(4 + 5 + "px"); //9px
console.log("$" + 4 + 5); //$45
console.log("2" / "5"); //0.4
console.log(6 - "2"); //4
console.log(false + 34); //34
console.log(true + 34); //35
console.log(true + "34"); //true34
```

Operadores:

- Resta -
- Suma +
- Multiplicación *
- División /
- Resto %

  ```
  console.log(5 % 2); //1
  ```

- Exponente **

  ```
  console.log(2 ** 3); //8
  ```

- Incremento/Decremento ++/—

Reglas de orden, **Muy importante!!!!** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_precedence

Comparaciones:

- Mayor que >
- Menor que <
- Mayor o igual que >=
- Menor o igual que <=
- Igual ==
- Distinto !=
- Igualdad estricta ===

Cosas raras

```
//Strings -> Letra a letra según orden unicode

console.log('Z' > 'A'); // true
console.log('Glow' > 'Glee'); // true
console.log('Gloa' > 'Glee'); //true
console.log('Bee' > 'Be'); // true
console.log('a' > 'A'); // true
console.log('Á' > 'a'); // true

// Tipos distintos -> se convierten a números:
console.log('2' > 1); // true
console.log('01' == 1); // true
console.log(true == 1); // true
console.log(false == 0); // true
console.log('0' == 0); // true
console.log(Boolean('0') == Boolean(0)); // false
console.log(null == undefined); //true

//Si no puede convertir a número devuelve false
console.log('Hola' > 34); // false
console.log('Hola' < 34); // false
console.log('Hola' == 34); // false
```

===

Super importante → para que no haya conversión de datos → si los tipos son distintos devuelve false

```
console.log('01' === 1); // false
console.log(true === 1); // false
```

```
console.log(false === 0); // false
console.log('0' === 0); // false
console.log(null === undefined) //false
```

Casos extraños con null y undefined comparado con 0

```
console.log(null > 0); // false
console.log(null < 0); // false
console.log(null == 0); // false
console.log(null >= 0); // true
console.log(null >= 0); //true

//con undefined siempre false
console.log(undefined > 0); // false
console.log(undefined < 0); // false
console.log(undefined == 0); // false
```

Condicionales If

```
//lo normal
if (year == 2015) {
console.log('That's correct!');
console.log('You're so smart!');
} else if (year == 2016){
console.log();
} else {
console.log();
}
```

Condicional ternario

Otra forma de expresar los condicionales:

```
//...?...:...
let accessAllowed = (age > 18) ? true : false;
let accessAllowed2 = age > 18 ? true : false; //No recomendado -> usar parent

let message = (age < 3) ? 'Hi, baby!' :
    (age < 18) ? 'Hello!' :
    (age < 100) ? 'Greetings!' :
    'What an unusual age!';

(company == 'Netscape') ?
```

```
alert('Right!') : alert('Wrong.'); // No recomendado, no pensado para mostrar
//alertas, usar para dar valor a una variable como antes
```

Operadores:

Orden:

```
const a = 4 ** 3 ** 2; // Same as 4 ** (3 ** 2); evaluates to 262144
const b = 4 / 3 / 2; // Same as (4 / 3) / 2; evaluates to 0.6666...
```

Valores “Truthy” y “Falsy”:

un valor **“truthy”** es cualquier valor que se evalúa como true cuando se utiliza en un contexto booleano, mientras que un valor **“falsy”** es cualquier valor que se evalúa como false.

Operadores “Falsy”:

- false
- 0
- “”
- ‘’
- null
- undefined
- NaN

Operadores “truthy”:

- Números distintos de 0 (tanto positivos como negativos)
- Cadenas no vacías ("hello", "0", "false")
- Objetos y arrays (incluso los vacíos: {}, [])
- Funciones

&&:

En JavaScript, el operador && devuelve el primer valor “falsy” que encuentra, o el último valor si todos son “truthy”.

```
alert(2 && 3) // 3
alert(3 && 0) // 0
```

||:

El operador || devuelve el primer valor “truthy” que encuentre, o el último valor si son todos “falsy”.

```
alert(2 || 3) // 2
alert(0 || undefined) // undefined

let fistname = ''
let seconName = 'perez'
alert(fisrtname || secondName || 0) // perez
```

Ejercicio

```
alert( 1 && null && 2 ); // null
alert( null || 2 && 3 || 4 ); // 3
if (-1 || 0) alert( 'first' ); // first
if (-1 && 0) alert( 'second' ); // nada
if (null || -1 && 1) alert( 'third' ); // third
alert( alert(1) && alert(2) ); // 1
```

NOT !:

```
alert( !true ); // false
alert( !0 ); // true
//A veces se usa doble para convertir a booleano en vez de Boolean()
let test = "non-empty string";
let test2 = "";
alert( !!test ); // true
alert( !!test2 ); // false
alert( !!null ); // false
```

Nullish coalescing ??

Comprobar si algo esta definido → no es ni null ni undefined

```
a ?? b //-> devuelve si a esta definido y sino b

// uso para darle valor a una variable
result = (a !== null && a !== undefined) ? a : b; //de forma literal
result = a ?? b

//No se puede usar sin parentesis con && o ||
let x = 1 && 2 ?? 3; // SyntaxError: Unexpected token '??'
```

Switch

```
// a = 4
switch (a){
```

```
    case 4:
        console.log('solo se ejecuta el 4');
        break;
    case 5:
        console.log('solo se ejecuta el 5 y el default');
    default:
        console.log('Default')
    }
// salida -> solo se ejecuta el 4
//a = 5
//salida -> solo se ejecuta el 5 y el default, Default
```

```
// a = 4
switch (a){
    case 4:
        console.log('solo se ejecuta el 4');
    case 5:
        console.log('solo se ejecuta el 5 y el default');
    default:
        console.log('Default')
    }
    // salida -> solo se ejecuta el 4, solo se ejecuta el 5 y el Default, Def
```

```
// a = 4
switch (a){
    case 4:
        console.log('solo se ejecuta el 4');
    case 5:
        console.log('solo se ejecuta el 5 y el default');
        break;
    default:
        console.log('Default')
    }
    // Solo se ejecuta el 4, solo se ejecuta el 5 y el default
```

Bucles:

```
//while
let i = 3;
while (i) {
    alert( i );
    i--;
}

//do...while
```

```
let i = 0;
do {
    console.log( i );
    i++;
} while (i < 3);

//for (1ª forma)
for (let i = 0; i < 3; i++) {
    console.log(i);
}

//for (2ª forma)
let i = 0;
for (; i < 3;) {
    alert( i++ );
}
```

Tenemos break:

```
let sum = 0;
while (true) {
    let value = +prompt("Enter a number", '');
    if (!value) break;
    sum += value;
}
console.log( 'Sum: ' + sum );

//+algo intenta convertir la cadena en un número

let apples = "2";
let oranges = "3";
console.log(+apples + +oranges); //5
```

Tenemos continue:

```
for (let i = 0; i < 10; i++) {
    if (i % 2 == 0) continue;
    console.log(i); // 1, then 3, 5, 7, 9
}
```

Etiquetas:

```
outer: // identifica el bucle de fuera, pudiendo volver a el con un continue/
for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {
        let input = prompt(`Value at coords (${i},${j})`, '');
```

```
        if (!input) break outer;
        // do something with the value...
    }
}
alert('Done!');
//${i} es la manera de meter el valor de i en el String
```

Funciones:

```
function showMessage() {
let mensaje = 'Hello everyone!';
console.log(mensaje);
}
showMessage();

//Las variables declaradas dentro no son accesibles fuera
console.log(mensaje); //ReferenceError: mensaje is not defined
```

```
//las variables globales oscurecen a las globales:
let userName = 'John';
function showMessage() {
    let userName = "Bob"; // como si nunca se hiciera
    let message = 'Hello, ' + userName;
    console.log(message);
}
console.log( userName ); // John
showMessage();
console.log( userName ); // John

// Siempre podemos modificar la variable
function showMessage() {
    userName = "Bob";
    let message = 'Hello, ' + userName;
    console.log(message);
}
console.log( userName ); // John
showMessage();
console.log( userName ); // Bob
```

Se puede llamar a funciones sin usar todos los parámetros:

```
function showMessage(from, text) {
console.log(from + ': ' + text);
}
```

```
showMessage('Ann', 'Hello!'); //Ann: Hello!
showMessage('Ann'); //Ann: undefined
```

Se puede incluir valor por defecto:

```
function showMessage(from, text = 'no hay texto') {
alert( from + ': ' + text );
}
showMessage('Ann', 'Hello!'); //Ann: Hello!
showMessage('Ann'); // Ann: no hay texto
```

En java no se podía hacer esto, se usaba la sobrecarga:

JAVASCRIPT NO SOPORTA LA SOBRECARGA

```
////// JAVA ////// -> RECORDATORIO
public class Main {
    public static void main(String[] args) {
        showMessage("Ann", "Hello!"); // Ann: Hello!
        showMessage("Ann");           // Ann: no hay texto
    }

    // Método con dos parámetros
    public static void showMessage(String from, String text) {
        System.out.println(from + ": " + text);
    }

    // Sobrecarga del método para proporcionar un valor predeterminado
    public static void showMessage(String from) {
        showMessage(from, "no hay texto");
    }
}
```

el número de variables puede ser variable:

```
function foo() {
    for (let i = 0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
foo('Hola'); //Hola
foo(1,2,3,4,5,6,7,8,9); // 1,2,3,4,5,6,7,8,9

// Otra forma rara de declarar las un número variable de variables (lol)
function my_log(x, ...args) { //
console.log(x, args, ...args);
}
my_log('Hola', 'qué', 'tal'); //Hola ['qué', 'tal'] qué tal
```

```
function my_log(x, ...args) {
console.log(x, args);
}
my_log('Hola', 'qué', 'tal'); //Hola ['qué', 'tal']

function my_log(x, ...args) {
console.log(x, ...args);
}
my_log('Hola', 'qué', 'tal'); //Hola qué tal
```

Return

```
// Si no existe (no hay return), la función devuelve undefined
//Se puede usar vacío (return;) para terminar la ejecución de la función
function checkAge(age) {
    if (age >= 18) {
        return true;
    }
    return confirm('¿Tienes permiso?');
}
let age = prompt('How old are you?', 18); //predeterminado 18 como respuesta
let valido = checkAge(age);
```

Expresiones:

Variables a las que son asignadas una función

```
let sayHi = function() {
    console.log( "Hello" );
}; // '}' por acabar la función y ';' por acabar la declaración de la variabl
console.log(sayHi); //se imprime el código de la función
console.log(sayHi());// Hello (se ejecuta el codigo) \nundefined (la propia e

//se puede llamar directamente a la función
sayHi() //Hello
```

ORDEN PARA USARLO:

Una declaración de función se puede usar antes, una expresión, no !!!!

```
sayHi("John"); // Hello, John
saludar("John"); // ReferenceError: saludar is not defined
function sayHi(name) {
    alert( `Hello, ${name}` );
}
let saludar = function(name) {
```

```
    alert( `Hello, ${name}` );
};
```

Arrows

- Equivalente a funciones lambda
- Funciones anónimas
- forma concisa de declarar una función

```
let sum = (a, b) => a + b;
// casi lo mismo que:
let sum = function(a, b) {
return a + b;
};
```

Si es de varios niveles, necesita {} y tiene que tener un return:

```
let sum = (a, b) => {
    let result = a + b;
    return result;
};
console.log( sum(1, 2) ); // 3
```

Diferencias:

Característica	Arrow	Expression
Sintaxis	Más concisa	Más tradicional
Contexto this	Hereda el this léxico	Tiene su propio this
Uso como constructor	No se puede usar con new	Se puede usar con new
Acceso a arguments	No tiene	Tiene

Ejemplo this:

```
let obj = {
    value: 42,
    regularFunction: function() {
        console.log(this.value); // Se refiere a `obj`
    },
    arrowFunction: () => {
        console.log(this.value); // Se refiere al contexto léxico, probableme
    }
};

obj.regularFunction(); // 42
obj.arrowFunction();   // undefined (o el valor global si no está en modo est
```

Ejemplo constructor:

```
let Sum = (a, b) => a + b;
let result = new Sum(1, 2); // Error: Sum is not a constructor
```

```
let Sum = function(a, b) {
    this.result = a + b;
};
let instance = new Sum(1, 2);
console.log(instance.result); // 3
```

Ejemplo arguments:

```
let sum = function() {
    console.log(arguments); // Muestra todos los argumentos pasados
    let total = 0;
    for (let i = 0; i < arguments.length; i++) {
        total += arguments[i];
    }
    return total;
};

console.log(sum(1, 2, 3, 4)); // [1, 2, 3, 4] y devuelve 10
```

```
let sum = () => {
    console.log(arguments); // Error: `arguments` no está definido
    let total = 0;
    for (let i = 0; i < arguments.length; i++) {
        total += arguments[i];
    }
    return total;
};

console.log(sum(1, 2, 3, 4)); // Lanza un error
```

Arrays

Crear

```
let fruits = ['Apple', 'Banana'];
```

Acceder a un elemento

```
console.log(fruits[0]); //Apple
```

Longitud

```
console.log(fruits.length);
```

Recorrerlo

```
fruits.forEach(function(item, index, array) {
    console.log(item, index);
});
```

Añadir un elemento

```
let newLength = fruits.push('Orange');
```

Eliminar el último elemento

```
let last = fruits.pop();
```

Buscar un índice de un elemento

```
let pos = fruits.indexOf('Banana');
```

Eliminar un elemento

```
let removedItem = fruits.splice(pos, 1);
```

Objetos

Crear objeto:

```
let user_vacio_1 = new Object(); // "object constructor" syntax
let user_vacio_2 = {}; // usando sintaxis literal
//ámbas formas son equivalentes

let user = {
    name: "John",
    age: 30
};
//accedemos a la info (da igual cual de las 2 formas):
```

```
console.log(user.name);
console.log(user["name"]);
```

Añadir y eliminar nuevas propiedades:

```
let user = {
    name: "John",
    age: 30
};
//Añadir
user.isAdmin = true;

//Eliminar
delete user.age;
```

Las propiedades pueden tener + de una palabra:

```
let user = {
    name: "John",
    age: 30,
    "likes birds": true
};

//pero habría que acceder con []
console.log(user["likes birds"]); // no se puede hacer user.likes birds (obvi
```

Comprobar si existe una propiedad

Con undefined:

```
let user = {};
console.log(user.noSuchProperty === undefined); // true
```

Con “in”:

```
console.log("noSuchProperty" in user); // false
let key = "age";
console.log(key in user); // false
let nombre = name;
console.log(nombre in user); //true
```

Recorrer objetos:

```
let user = {
    name: "John",
```

```
    age: 30,
    isAdmin: true
};
for (let key in user) {
    alert(key); // name, age, isAdmin
    alert(user[key]); // John, 30, true
}
```

Anidar objetos:

```
let user = {
    name: "John",
    birthday: {
        year: 1990,
        month: "November"
    }
};
console.log(user.birthday.year);
console.log(user["birthday"]["month"]);
```

API

Hay un montón de interfaces y Apis predefinidas:

Lista: https://developer.mozilla.org/en-US/docs/Web/API

Window

Variable global disponible en el navegador
– Ofrece muchos métodos y objetos, accesibles sin necesidad de usar el objeto window
– document: el documento HTML
– alert(), prompt()...
– Información sobre la ventana: screenX, screenY, scrollX, scrollY...

– Más info: https://developer.mozilla.org/en-US/docs/Web/API/Window

Propiedad Onload:

Para cuando el recurso se ha cargado

```
window.addEventListener('load', (event) => {
    init();
});
//Alternativamente pero no recomendable
window.onload = function() {
    init();
};
```

Propiedad Onclick:

```
<button id="boton">Click me</button>
----
let element = document.getElementById("boton");
element.addEventListener("click", myScript);
//Alternativamente pero no recomendable
element.onclick = function(){myScript};
```

Acceder a documentos:

```
let markup = document.documentElement.innerHTML;
```

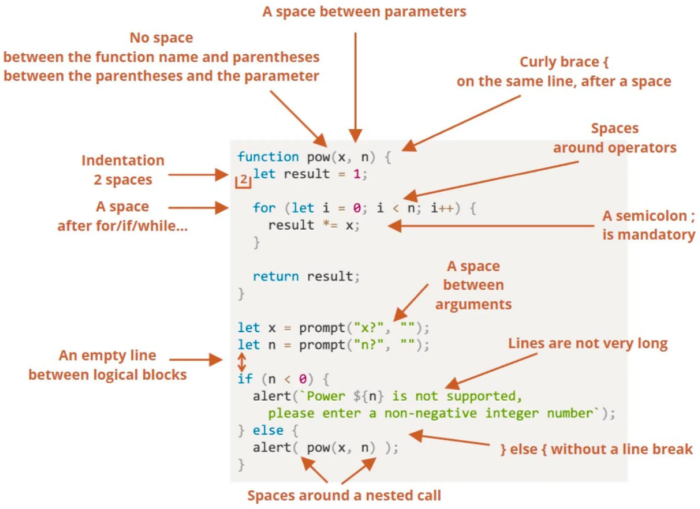
Acceder a elementos:

```
// No entiendo nada xd
let myElement = document.getElementById("intro");
let x = document.getElementsByTagName("p");
let x = document.getElementsByClassName("intro");
let x = document.querySelectorAll("p.intro");
```

Añadir elementos nuevos:

```
let element = document.getElementById("new");
element.appendChild(tag);
```

Estilo:



Callback:

Funciones pasadas como argumento a otra función, que se invocan cuando se completa alguna acción o rutina. Pueden ser síncronas pero normalmente se usan de forma asíncrona

```
let suma = 0;
function callback(val=20){
    suma += val;
    console.log("Dentro del callback:",suma);
}
console.log(suma);
//El método .forEach() llama a la función callback para cada elemento del arr
[1,2,3,4].forEach(callback);
console.log(suma);
```

```
let suma = 0;
function callback(val=20){
    suma += val;
    console.log("Dentro del callback:",suma);
}
console.log(suma);
//SetTimeout no ejecuta la función inmediatamente, sino que programa su ejecu
setTimeout(callback,100);
console.log(suma);

//0
//0
//Dentro del callback: 20
```

Input

Teclado

Eventos:

- keydown: Cuando se pulsa una tecla
- keypress: Mientras se pulsa (múltiples invocaciones). Deprecado
- keyup: cuando se libera la tecla

```
document.addEventListener('keydown', logKey);
function logKey(e) {
console.log(e.code);
}
```

Ratón

Eventos:

- click: cuando se pulsa y luego se libera el botón del ratón

- dblclick: doble click
- mouseup: cuando se libera el botón del ratón
- mousedown: cuando se pulsa el botón del ratón
- mousemove: mientras se mueve el ratón (múltiples invocaciones)

```
document.addEventListener('click', click);
function click(e) {
console.log(e);
}
```

Canvas

> Me lo salto porque creo que es lo menos importante → de resaca

Mirar si los eventos son solo de jquery o de js también

Strict mode

- Variante restringida de JavaScript
- Alternativa al modo normal o "sloppy mode"
- La semántica es diferente
- Puede funcionar de forma diferente en los navegadores
- Cambios:
  - Elimina fallos silenciosos para que lancen errores
  - Soluciona fallos que dificulta la optimización
  - Prohíbe sintaxis que pueda ser definida en futuras versiones de ECMAScript
- Se usa automáticamente en los módulos y las clases

```
//para usarlo en todo el. script
"use sctrict";
const v = "Hi! I'm a strict mode script!";
```

Para usarlo en una función:

```
function myStrictFunction() {
    "use strict";
    function nested() {
        return "And so am I!";
    }
    return `Hi! I'm a strict mode function! ${nested()}`;
}
```

```
function myNotStrictFunction() {
    return "I'm not strict.";
}
```

No se puede hacer:

No se puede usar en funciones con parámetros por defecto

```
//MAL
function sum(a = 1, b = 2) {
"use strict";
return a + b;
}
//Uncaught SyntaxError: Illegal 'use strict' directive in
//function with non-simple parameter list
```

No se pueden duplicar parametros en funciones:

```
function sum(a, a) {
"use strict";
return a + a;
}
// Uncaught SyntaxError: Duplicate parameter name not allowed in this context
```

No se pueden usar variables sin declarar

```
"use strict";
variable = 12;
// Uncaught ReferenceError: variable is not defined
```

Y más → https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Módulos

Mecanismo para organizar, reutilizar y encapsular código. Cada módulo puede exportar partes especificas de su código y otros módulos pueden importarlas según sea necesario.

Separar el código en bloques que se puedan importar si es necesario

Muchas bibliotecas y frameworks hacían uso de ellos (CommonJS y RequireJS)

Soportado en la nayor parte de navegadores modernos

Usa importantes y export

Archivos con extensión .mjs (aunque puede ser también **.js)**

Automátícamente usan el modo estricto

para usar otros módulos necesito usar un módulo, no se puede usar import en un script

solo se ejecutan una vez aunque se hayan referenciado múltiples veces

las características importadas no están disponibles en el sope global

Las variables globales si están disponibles en los módulos

```
<head>
…
<script type="module" src="main.js"></script>
</head>
<!-- • No hace falta añadir defer porque los módulos lo aplican automáticamen
```

Export

Exportar las características del módulo: funciones, var, let, const, clases

```
export const name = "square";
export function draw(ctx, length, x, y, color) {
    ctx.fillStyle = color;
    ctx.fillRect(x, y, length, length);
    return { length, x, y, color };
}
// Lo mismo pero al final
const name = "square";
function draw(ctx, length, x, y, color) {
    ctx.fillStyle = color;
    ctx.fillRect(x, y, length, length);
    return { length, x, y, color };
}
export {name, draw};
```

Import

```
import {name, draw} from "./modules/square.js";
console.log(name);
// se pueden renombrar
import {name as nombre, draw} from "./modules/square.js";
console.log(nombre);
```

importar todo dentro de un objeto:

```
import * as mates from './math.js';
```

Ejemplo básico módulos:

```
// archivo math.js

export function sumar(a, b) {
    return a + b;
}
export function restar(a, b) {
    return a - b;
}
export const PI = 3.14159;
```

```
//archivo main.js

import { sumar, restar, PI } from './math.js';

console.log(sumar(2, 3)); // 5
console.log(restar(10, 5)); // 5
console.log(PI); // 3.14159
```

Exportación por defecto

No requiere nombre al ser importado {...}

Como sabe cual es puedes importarlo con el nombre que quieras, el resto debes poner {nombre exacto} de la importación

```
//saludo.js
export default function saludar(nombre) {
    return `Hola, ${nombre}!`;
}
```

```
//main.js
import saludo from './saludo.js';
console.log(saludo('Juan')); // Hola, Juan!
```

Tapermonkey

Extensión del navegador

Disponible para múltiples navegadores

Permite ejecutar JS en la web que quieras y hacer modificaciones:

– Eliminar elementos molestos

– Añadir características que queramos

Permite buscar scripts de otros usuarios

Cambiar @match por la URL de la web que queramos

– Añadir * para que sirva en todo el dominio

Opcionalmente cambiar @name para asignarle un nombre

Añadir un console.log con el mensaje que sea dentro de function:

```
(function() {
    'use strict';
    console.log("¡Funciona!");
})();
```

Guardar el fichero (File > Save)

Clases

Se define con Class

y un nuevo objeto con new

```
class Rectangle {
    …
}
const rectangulo = new Rectangle(100,80);
const rectangulo = new Rectangle(); //usa los por defectos si tiene y sino un

//usar siempre const para objetos -> sino problemas, puedes liarla con lo de
let obj = { name: "Juan" };
obj.name = "Ana"; // Mutar la propiedad
console.log(obj.name); // "Ana"

obj = { age: 30 }; // Esto es válido
console.log(obj); // { age: 30 }
```

Método constructor

- constructor
- Tiene que ser único
- Se puede usar super
- Puedes definir propiedades de instancia

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

classFields

característica moderna de JavaScript que permite declarar propiedades directamente dentro de la definición de una clase, sin necesidad de inicializarlas en el constructor.

```
//antes
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre; // Propiedad pública
        this.edad = edad;     // Propiedad pública
    }
}
```

```
//ahora
class Persona {
    nombre = "Desconocido";  // Propiedad pública con valor por defecto
    edad = 0;                // Propiedad pública con valor por defecto
}
```

Estas propieades pueden ser:

estáticas

No se usa let ni const

Se les puede asignar un valor por defecto

Si no se les asigna valor serán undefined

Pueden ser privadas con #

```
class Rectangle {
    static total = 0; // Estática
    #id; // Privado
    height = 0; // Le asignamos un valor por defecto
    width;
    constructor(height, width) {
        this.height = height;
        this.width = width;
        this.#id = Math.random();
        Rectangle.total++;
        }
}
```

Herencia

```
class Animal {
    constructor(name) {
        this.name = name;
    }
```

```
}
    class Dog extends Animal {
    constructor(name) {
        super(name);
    }
}
```

métodos

pueden ser estáticos y privados

```
class Rectangle {
    height;
    width;
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    getArea(){
        return this.height * this.width;
    }
}
```

métodos Get y Set

para cambiar u obtener desde fuera e valor de un parámetro

```
class Color {
    constructor(r, g, b) {
        this.values = [r, g, b];
    }
    get red() { return this.values[0]; }
    set red(value) { this.values[0] = value; }
}

const color = new Color(255,0,0);
console.log(color.red);
color.red = 100;
```

AJAX

Asynchronous JavaScript and XML

No es una tecnología, es una integración de diferentes tecnologías: HTML/XHTML, CSS, JS, DOM, XML, XSLT, JSON, XMLHttpRequest

Es una técnica que permite que las aplicaciones web se comuniquen con el servidor de forma asíncrona, sin recargar toda la página. Esto hace que las páginas web sean más dinámicas y

rápidas, mejorando la experiencia del usuario.

XMLHttpRequest

Se usa para obtener datos de una URL, a pesar del nombre, no sólo sirve con XML

Pasos

Generar un objeto

```
const httpRequest = new XMLHttpRequest();
```

Asignarle un callback

```
httpRequest.onreadystatechange = () => {…};
```

Inicializar la petición

```
httpRequest.open("METHOD", "URL", true);
// METHOD: Método HTTP en mayúsculas (GET / POST / PUT …)
// true: Indica que la petición es asíncrona
```

Enviamos la petición, opcionalmente con el body

```
httpRequest.send();
```

Ejemplo

```
const httpRequest = new XMLHttpRequest();
if (!httpRequest) {
    console.err("Giving up :( Cannot create an XMLHTTP instance");
} else {
    httpRequest.onreadystatechange = getData;
    httpRequest.open("GET", "diccionario.txt");
    httpRequest.send();
}
function getData() {
    try {
        if (httpRequest.readyState === XMLHttpRequest.DONE) {
            if (httpRequest.status === 200) {
                let palabras = httpRequest.responseText.split("\n");
                console.log(palabras.length);
            } else { console.log("There was a problem with the request."); }
        }
    } catch (e) {
        alert(`Caught Exception: ${e.description}`);
    }
}
```

XMLHttpRequest.timeout : permite especificar un timeout

XMLHttpRequest.setRequestHeader(): permite especificar las cabeceras

Hay más eventos asociados: abort, error, timeout...

Callback hell

Si queremos una aplicación con lo siguiente:

- Que tenga un botón
- Al hacer click en el botón aparece un mensaje diciendo "¡Hola!"
- Cuando ha pasado 1 segundo el mensaje cambia a "¡Adiós!"
- Un segundo después desaparece el mensaje

El código luciría un poco así:

```
window.addEventListener("load", function() {
    document.getElementById("boton").addEventListener("click", function(){
        document.getElementById("mensaje").innerHTML = "¡Hola!";
        setTimeout(function(){
            document.getElementById("mensaje").innerHTML = "¡Adiós!";
            setTimeout(function(){
                document.getElementById("mensaje").innerHTML = "";
            }, 1000);
        }, 1000);
    });
});
```

Mejor no usar funciones anónimas, queda más claro:

```
window.addEventListener("load", init);
function init(){document.getElementById("boton").addEventListener("click", sa
function saludar(){
    document.getElementById("mensaje").innerHTML = "¡Hola!";
    setTimeout(despedirse, 1000);
}
function despedirse(){
    document.getElementById("mensaje").innerHTML = "¡Adiós!";
    setTimeout(borrarMensaje, 1000);
}
function borrarMensaje(){document.getElementById("mensaje").innerHTML = "";}
```

Pero claro, el manejo de excepciones puede ser horrible, para eso se usan las promesas:

Promesas

Objetos que representan que se completará una operación asíncrona

Le asocias un callback, en vez de pasárselo a una función

Una promesa tiene tres posibles estados:

1. **Pendiente (**pending**)**: La operación aún no se ha completado.
2. **Resuelta (**fulfilled**)**: La operación se completó con éxito y tiene un valor resultante.
3. **Rechazada (**rejected**)**: La operación falló y tiene un motivo de error.

```
const audioSettings = {…};

function successCallback(result) {
    console.log(`Audio file ready at URL: ${result}`);
}

function failureCallback(error) {
    console.error(`Error generating audio file: ${error}`);
}
//Con callbacks tradicionales
//createaudioFileAsync tiene que estar configurado para ver si la creacion,
//es exitosa o no y llamar a una de las dos funciones dependiendo de esto
createAudioFileAsync(audioSettings, successCallback, failureCallback);
//Con promesas -> si promesa resuelta ejecuta ducces..., si rechazada failure
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
```

Otro ejemplo de como las promesas son útiles:

En el **callback hell**, las funciones se anidan profundamente cuando una operación asíncrona depende de los resultados de otras. Esto genera un código difícil de leer y mantener.

```
conectarAServidor((error, conexion) => {
    if (error) {
        console.error("Error al conectar:", error);
        return;
    }
    obtenerDatos(conexion, (error, datos) => {
        if (error) {
            console.error("Error al obtener datos:", error);
            return;
        }
        procesarDatos(datos, (error, resultado) => {
            if (error) {
                console.error("Error al procesar datos:", error);
                return;
            }
            console.log("Resultado final:", resultado);
        });
    });
});
```

```
//Si ocurre un error, este código puede ser difícil de manejar, ya que los
//errores no siempre se propagan correctamente en operaciones asíncronas.
```

Solución con promesas

Las promesas eliminan la necesidad de anidar múltiples callbacks al permitir encadenar operaciones mediante `.then()`. Esto hace que el flujo del código sea más lineal.

Reescritura usando **promesas**:

```
conectarAServidor()
    .then(conexion => obtenerDatos(conexion))
    .then(datos => procesarDatos(datos))
    .then(resultado => {
        console.log("Resultado final:", resultado);
    })
    .catch(error => {
        console.error("Error:", error);
    });
```

- **Encadenamiento**: Cada operación devuelve una nueva promesa, lo que permite encadenar operaciones secuenciales.
- **Manejo centralizado de errores**: El `.catch()` captura cualquier error ocurrido en cualquier parte de la cadena, lo que simplifica la gestión de errores.

Cómo las excepciones influyen en las promesas

Las excepciones son fundamentales en las promesas porque cualquier error o excepción dentro de una promesa se captura automáticamente y rechaza la promesa. Esto unifica el manejo de errores, tanto sincrónicos como asíncronos.

Errores en callbacks tradicionales

En el modelo tradicional de callbacks, necesitas manejar manualmente los errores en cada nivel, lo que aumenta la complejidad

Incluso si lanzas una excepción dentro de una promesa, esta será manejada en el `.catch()`:

```
conectarAServidor()
    .then(conexion => {
        throw new Error("Fallo en la conexión");
    })
    .then(datos => procesarDatos(datos))
    .catch(error => {
        console.error("Error:", error.message); // "Error: Fallo en la conexi
    });
```

Otra forma más simplificada aún es con async/await

Async/await

El uso de `async/await` lleva la legibilidad un paso más allá. Combina las ventajas de las promesas con una sintaxis que parece sincrónica.

- **Ventaja 1**: Evita la necesidad de encadenar `.then()`.
- **Ventaja 2**: El manejo de errores es más intuitivo con `try/catch`.

```
async function ejecutarTareas() {
    try {
        const conexion = await conectarAServidor();
        const datos = await obtenerDatos(conexion);
        const resultado = await procesarDatos(datos);
        console.log("Resultado final:", resultado);
    } catch (error) {
        console.error("Error:", error);
    }
}

ejecutarTareas();
```

`async`

Asocia una función asíncrona a un nombre

```
async function name() {
//
}
```

`await`

- Permite usar código asíncrono con Promesas como si fuera síncrono
- Pausa la ejecución del resto del código hasta que se resuelva la promesa
- Se puede usar try/catch
- Tiene que usarse dentro de una función async o en el top de un módulo

```
async function getWords() {
    const response = await fetch("diccionario.txt");
    const result = await response.text();
    let palabras = result.split("\n");
    console.log(palabras.length);
}
getWords();
```

Fetch

- API para obtener recursos
- Evolución de XMLHttpRequest
- Basado en Promesas
- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

```
//Código ejemplo:
fetch("diccionario.txt")
    .then((result) => result.text())
    .then(processData);

function processData(result){
    let palabras = result.split("\n");
    console.log(palabras.length);
}
```

También permite hacer POST, alternativa a XMLHttpRequest(vimos en AJAX) → más simple

```
const data = {name: "John", surname: "Smith", age: 48};
const options = {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify(data)
    }
const URL = "127.0.0.1:5500";
fetch(URL, options)
    .then((result) => console.log("Enviado con éxito"))
    .catch((e) => console.log("Error:", e));
```

Importante opción Redirect para viajar entre una página y otra