

Introducción a JavaScript



■ ¿Qué es JavaScript?



Un lenguaje para dominarlos a todos





```
> true + true  
< 2
```







```
> 0.1 + 0.2  
< 0.30000000000000004
```







```
> [] + {}
< "Object Object"
> {} + []
< 0
> |
```







```
> true == []
< false
> true == ![]
< false
```









VOSOTROS

YO



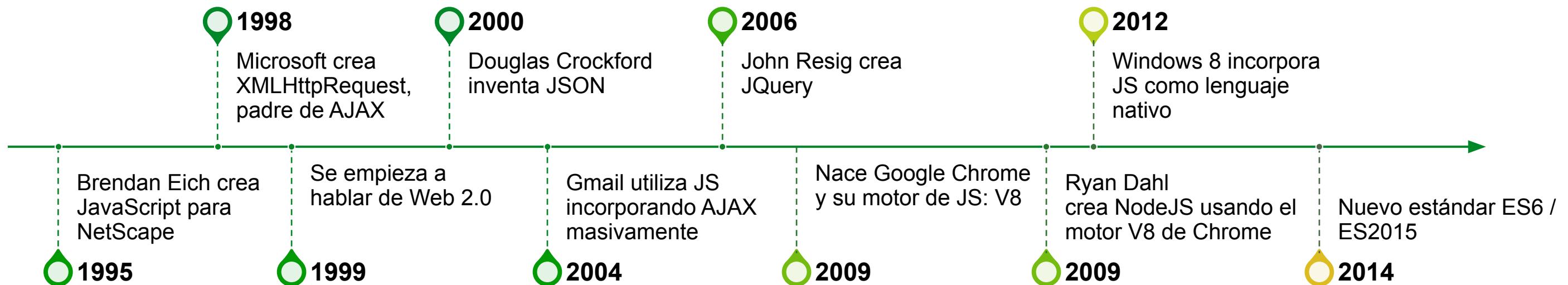
ONE DOES NOT SIMPLY

LEARN JAVASCRIPT

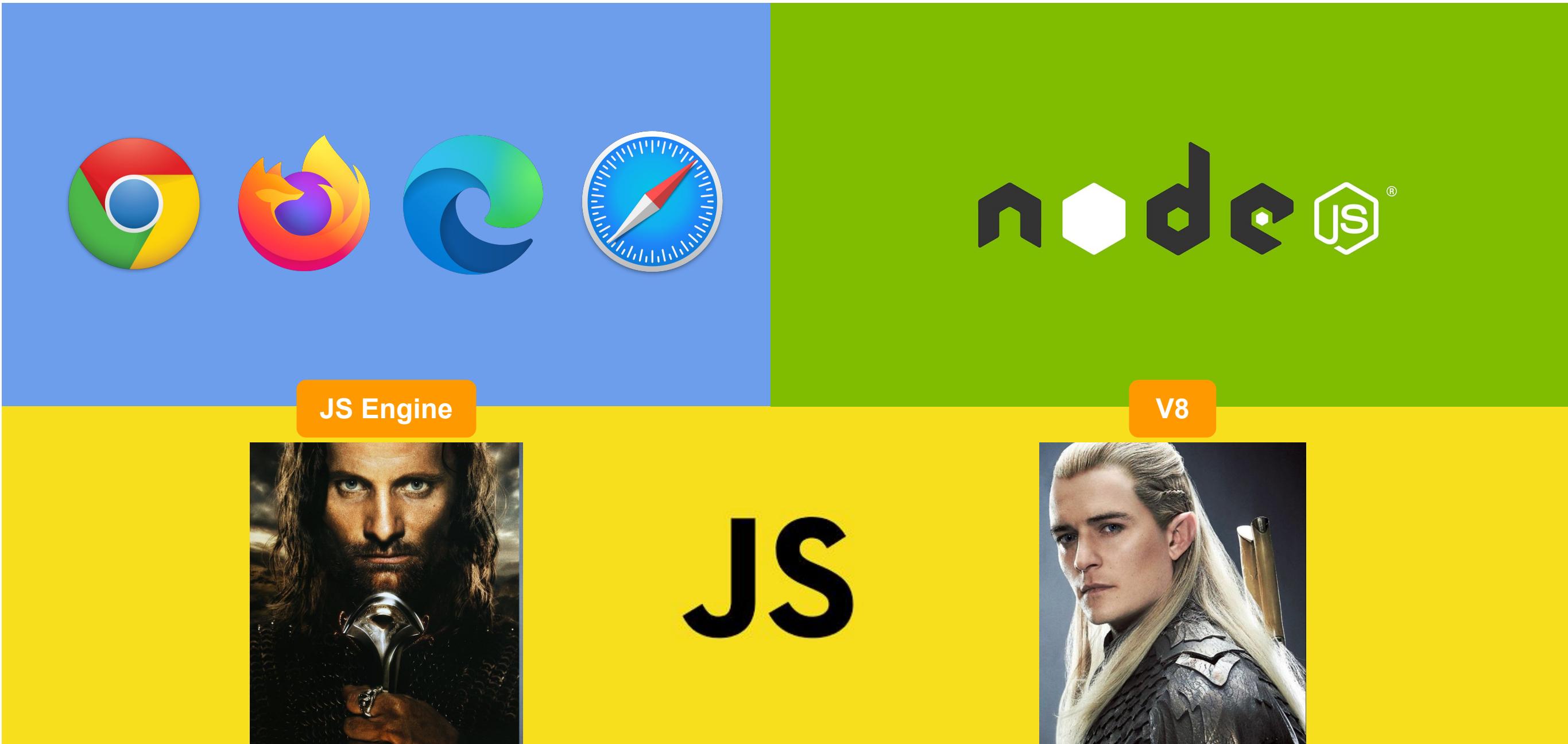




■ Un poco de historia



■ ¿Cómo funciona?



Sintaxis 101

JS



Referencia: <https://developer.mozilla.org/es/docs/Web/JavaScript>

```
1  const name = 'Alberto Casero';
2  var email = "alberto@kasfactory.net";
3
4  let instructor = { ← OBJETO LITERAL
5      name: name,
6      email: email,
7      age: 35, ← VALOR ENTERO
8      languages: ['Spanish', 'English', 'Galician', 'Python', 'JavaScript', 'PHP']
9  }
10
11 let run = true
12 while (run) { ← BUCLE WHILE
13     for (let language of instructor.languages) { ← BUCLE FOR
14         if (language == 'Spanish') {
15             console.log('Hola amigo!')
16         } else { ← IF / ELSE
17             switch (language) {
18                 case "English":
19                     console.log('Hello my friend');
20                     break;
21                 case 'Galician':
22                     console.log("Hola meu amigo")
23                     break;
24                 default:
25                     // Here we are using a template string
26                     console.log(`Hello my friend using ${language}`)
27                     break;
28             }
29         }
30     }
31     run = false ← VALOR BOOLEANO
32 }
```

■ Tipos primitivos

- **undefined:** valor indefinido
- **null:** valor nulo
- **Number:** 3, 3.1416, NaN, +/-Infinity, +0
- **String:** “hola amigo”, ‘hola amigo’, `hola \${friend}`
- **Boolean:** true/false
- **Bigint:** para representar números muy grandes
- **Symbol:** para metaprogramar



0



null



undefined



■ Operaciones

```
// Operadores aritméticos
let a = 5;
let b = 2;

console.log('Suma', a + b);
console.log('Resta', a - b);
console.log('Multiplicación', a * b);
console.log('División', a / b);
console.log('Exponente', a ** b); // a elevado a b
console.log('Módulo (resto)', a % b);
console.log('Negación', -a);
console.log('Incremento', a++);
console.log('Decremento', a--);
console.log('Pre-incremento', ++a);
console.log('Pre-decremento', --a);
```

```
// operadores booleanos
console.log('And', a && b)
console.log('Or', a || b)
console.log('Not', !a)

// comparadores de valor
console.log('Igual que', a == b) // '4' == 4 es true
console.log('Distinto que', a != b) // '4' != 4 es false
console.log('Mayor que', a > b) // '3' > 2 es true
console.log('Menor que', a < b) // 3 < 2 es false
console.log('Mayor o igual que', a >= b) // '4' >= 4 es true
console.log('Menor o igual que', a <= b) // '4' <= 4 es true

// comparadores de tipo y valor
console.log('Igual que', a === b) // '4' === 4 es false
console.log('Distinto que', a !== b) // '4' !== 4 es true
```



■ El bucle do...while

```
let i = 1;
do {
    console.log('Hello!', i)
    i++
} while(i < 10)
```



■ El bucle for

```
const fighters = ["Bud Spencer", "Chuck Norris", "Van Damme"];  
  
for (let i = 0; i < fighters.length; i++) {  
  const fighter = fighters[i];  
  console.log(fighter);  
}  
  
for (let i in fighters) {  
  const fighter = fighters[i]  
  console.log(fighter);  
}  
  
for (let fighter of fighters) {  
  console.log(fighter);  
}
```



■ Variables y bloques



■ let, var y const

```
var x = 1;
let y = 1;

if (true) {
  var x = 2;
  let y = 2;
}

console.log(x); // devuelve 2
console.log(y); // devuelve 1
```

var es *function scoped*
let es *block scoped*
const es *block scoped*



Bloques

En JavaScript, un bloque es cualquier conjunto instrucciones entre llaves `{ }`.

Las variables declaradas con **var**, son accesibles y se pueden modificar desde otro bloque.

Las declaradas con **let**, no.



```
1 const name = 'Alberto Casero';
2 var email = "alberto@kasfactory.net";
3
4 let instructor = {
5   name: name,
6   email: email,
7   age: 35,
8   languages: ['Spanish', 'English', 'Galician', 'Python', 'JavaScript', 'PHP']
9 }
10
11 let run = true
12 while (run) {
13   for (let language of instructor.languages) {
14     if (language == 'Spanish') {
15       console.log('Hola amigo!')
16     } else {
17       switch (language) {
18         case "English":
19           console.log('Hello my friend');
20           break;
21         case 'Galician':
22           console.log("Hola meu amigo")
23           break;
24         default:
25           // Here we are using a template string
26           console.log(`Hello my friend using ${language}`)
27           break;
28     }
29   }
30 }
31 run = false
32 }
```





Funciones



■ Funciones

```
function hello(name, surname = 'Skywalker') {  
    return `Hello ${name} ${surname}!`;  
}
```

```
const hello2 = function(name, surname = 'Skywalker') {  
    return `Hello ${name} ${surname}!`;  
}
```

Arrow => functions

```
const hello3 = (name, surname = 'Skywalker') => {  
    return `Hello ${name} ${surname}!`;  
}
```

```
const hello4 = (name, surname = 'Skywalker') => `Hello ${name} ${surname}!`
```

```
const hello5 = name => `Hello ${name}!`
```

⚠ Las arrow functions mantienen el scope del bloque anterior ⚠



Callbacks

Callbacks con papas
Callbacks en la sopa
Estofado de callbacks



```
function caminar() {
    console.log('caminando');
}

function comer(hora, comida) {
    comida(hora);
}

function desayuno(hora) {
    console.log(` ${hora}: Desayuno`);
}

function desayuno2(hora) {
    console.log(` ${hora}: Segundo Desayuno`);
}

function elevenses(hora) {
    console.log(` ${hora}: Desayuno`);
}

function almuerzo(hora) {
    console.log(` ${hora}: Almuerzo!`);
}

function serHobbit(){
    comer('7AM', desayuno);
    caminar();
    comer('10AM', desayuno2);
    caminar();
    comer('11AM', elevenses);
    caminar();
    caminar();
    comer('1PM', almuerzo);
}
```



Arrays y programación funcional



■ Arrays



- Los arrays en JavaScript son objetos.
- Al ser objetos, tienen prototipo y ofrecen una serie de métodos
- Muchos de éstos métodos son comunes en programación funcional

<https://wweb.dev/resources/js-array-functions-cheatsheet>



■ forEach

```
[ 1, 2, 3 ].forEach(val => console.log(val))  
// 1  
// 2  
// 3
```

- Sirve para recorrer los elementos de un array de una forma “funcional”
- No devuelve ningún valor



map

```
[ 2, 3, 4 ].map(val => val * 2)  
// [ 4, 6, 8 ]
```

- Sirve para “mapear” un array
- Es decir, convertir un array en otro array con el mismo número de elementos pero diferente contenido



■ find

```
[1, 10, 5, 6].find(val => val > 5)  
// 10
```

- Sirve para encontrar un elemento de un array
- Devuelve el primer elemento que cumpla la condición
- Si no lo encuentra, devuelve undefined



filter

```
[1, 10, 5, 6].filter(val => val > 5)  
// [ 10, 6 ]
```

- Sirve para “filtrar” un array
- Devuelve una array sólo con los elementos que cumplen la condición



sort

```
const a = [1,3,2,4].sort(); // [1,2,3,4] => por defecto, ascendiente
const b = [1,3,2,4].sort((a,b) => b-a); // [4,3,2,1]
const c = [1,3,2,4].sort((a,b) => a-b); // [1,2,3,4]
```

- Sirve para ordenar un array según un criterio (callback)

```
function compare(a, b) {
  if (a es menor que b según criterio de ordenamiento) {
    return -1;
  }
  if (a es mayor que b según criterio de ordenamiento) {
    return 1;
  }
  // a debe ser igual b
  return 0;
}
```



reduce

```
[ 5, 10, 15 ].reduce((accumulated, current) => accumulated + current, 0)  
// 30
```

- Sirve para “reducir” un array a un valor
- Típico cuando queremos recorrer un array para hacer una suma del total





```
[cow, potato, chicken, corn].map(cook) => [burger, fries, chickenLeg, soda]
[burger, fries, chickenLeg, soda].filter(isVegetarian) => [fries, soda]
[burger, fries, chickenLeg, soda].reduce(eat) => 💩
```



■ Clases, objetos, prototipos...





■ Tipos no primitivos

- **Objetos:** permiten estructurar o modelar estructuras de datos
- **Arrays:** para hacer colecciones de variables
- **Clases:** permiten crear estructuras de datos (objetos). Serían lo equivalente a los moldes

JS provee muchos objetos globales

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



■ Objetos literales

```
const goat = {  
    name: 'Michael Jordan',  
    team: 'Chicago Bulls',  
    number: 23,  
    shoot: function() {  
        return 3 // always in  
    }  
}
```



```
const goat = new Object();  
goat.name = 'Michael Jordan';  
goat.team = 'Chicago Bulls';  
goat.number = 23;  
goat.shoot = function() {  
    return 3; // always in  
}
```

Son instancias de la clase Object, que se crean con una sintaxis especial.

Vienen a ser los “diccionarios” en JavaScript.



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

Clases

```
function Team(name) {  
    this.name = name;  
    this.play = function() {  
        console.log(`${this.name} is playing!`);  
    }  
}  
  
Team.prototype.train = function () {  
    console.log(`${this.name} is training!`);  
}  
  
const bulls = new Team('Chicago Bulls');  
bulls.play()  
bulls.train()
```

Esto no cambia

```
class Team {  
    constructor(name) {  
        this.name = name;  
        this.players = []  
    }  
  
    play() {  
        console.log(`${this.name} is playing!`);  
    }  
  
    train () {  
        console.log(`${this.name} is training!`);  
    }  
  
    static info() {  
        console.log('This is an static method')  
    }  
  
    get squad() {  
        return this.players.join(',')  
    }  
  
    set squad(players) {  
        this.players = []  
        for (const player of players) {  
            this.players.push(player.name)  
        }  
    }  
}
```

Método
estático

Getters
y Setters



■ Herencia

```
class BasketballTeam extends Team {  
    constructor(name) {  
        super(name)  
        this.sport = 'Basketball'  
    }  
  
    getInitialSquad() {  
        this.players.shuffle() // shuffle no existe  
        return this.players.slice(0, 4)  
    }  
  
    play() {  
        const initialSquad = this.getInitialSquad().join(',')  
        console.log(`${this.name} is playing with ${initialSquad}`)  
    }  
}
```

```
function ...BasketballTeam (name, players) {  
    this.name = name;  
    this.players = players;  
  
    this.getInitialSquad = function() {  
        this.players.shuffle(); // shuffle no existe  
        return this.players.slice(0, 4);  
    }  
  
    this.play = function() {  
        const initialSquad = this.getInitialSquad().join(',')  
        console.log(`${this.name} is playing with ${initialSquad}`)  
    }  
}  
  
BasketballTeam.prototype = new Team();
```

Herencia clásica



Herencia “prototipada”



■ JavaScript y sus objetos

En Javascript todo son objetos (y si no, se comportan como tal).

Cada objeto tiene una propiedad **prototype** que apunta su prototipo.

El prototipo tiene a su vez su propiedad **prototype** que apunta a otro prototipo, y así sucesivamente hasta llegar al objeto Object (prototipo padre), cuyo prototipo es null.

A esto se le llama cadena de prototipos.



■ ...y objetos de tipos primitivos...



Objetos de tipos primitivos

En JavaScript definen objetos para cada uno de los tipos de datos primitivos:

objetos [String](#), objetos [Number](#), objetos [Boolean](#), objetos [Array](#)

Éstos objetos almacenan los mismos valores de los tipos de datos primitivos y añaden propiedades y métodos para manipular sus valores.

Todos éstos objetos, heredan de (o su prototipo es) [Object](#).



Las variables en la memoria



Valores de variables en memoria

Para los tipos primitivos se almacena el contenido en la memoria RAM

```
const figure = "✌";
```

```
let discovered = false;
```

```
let timesTurned = 0;
```

figure

“✌”

#2001

discovered

false

#2002

timesTurned

0

#2003

#2004

#2005



Valores de variables en memoria

Para los tipos no primitivos funciona diferente

```
const card = {figure, discovered}
```

```
const board = [card]
```

```
const logBoard = [board]
```

```
logBoard[0][0].timesTurned = 1
```

card.timesTurned === 1 // OJO CON ESTO!!!



RAM ASIGNADA AL PROGRAMA

card	=> #2302	#2301
------	----------	-------

	'figure': "✌", 'discovered': false	#2302
	, 'timesTurned': 1	

board	=> #2304	#2303
-------	----------	-------

	=> #2301	#2304
--	----------	-------

logBoard	=> #2306	#2305
----------	----------	-------

	=> #2303	#2306
--	----------	-------

		#2307
--	--	-------

		#2308
--	--	-------

■ Valor vs referencia

Por referencia, lo que “ocurre” dentro de la función “repercute” afuera.

Por valor, lo que “ocurre” dentro de la función, “se queda” en la función.

pass by reference

cup = 

fillCup()

pass by value

cup = 

fillCup()

www.penjee.com





■ Modules

“Los buenos escritores, dividen sus libros en capítulos.

Los buenos programadores, dividen su código en módulos.”





A dramatic scene from The Lord of the Rings: The Return of the King showing Aragorn, Legolas, and Gimli in a dark, smoky environment. Aragorn is in the foreground, looking intensely at the viewer. Legolas is behind him, and Gimli is on the right.

CommonJS
Modules

ES6
Modules

AMD
Modules

CommonJS Modules

```
// square.js
module.exports = class Square {
  ...
  constructor(width) {
    this.width = width;
  }
  area() {
    return this.width ** 2;
  }
};
```

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

```
// circle.js
const { PI } = Math;
exports.area = (r) => PI * r ** 2;
exports.circumference = (r) => 2 * PI * r;
```

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```



■ AMD Modules

- Iguales que los módulos CommonJS pero de naturaleza asíncrona.
- Pensados para ser utilizados en los navegadores web.



■ ES6 Modules

Los módulos JavaScript permiten partir nuestro código en diferentes archivos para poder re-utilizarlo en diferentes sitios.

Los módulos ES6 forman parte de la propia especificación del lenguaje.

Su carga es asíncrona.

Todo lo que hay en su interior es privado, salvo que se marque como exportable.



Module Cheatsheet

Name Export

```
export const name = 'value'
```

Name Import

```
import { name } from '....'
```

Default Export

```
export default 'value'
```

Default Import

```
import anyName from '....'
```

Rename Export

```
export { name as newName }
```

Name Import

```
import { newName } from '....'
```

Export List + Rename

```
export {  
    name1,  
    name2 as newName2  
}
```

Import List + Rename

```
import {  
    name1 as newName1,  
    newName2  
} from '....'
```

■ Algunos objetos globales de JS



■ Math

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

- Permite hacer operaciones matemáticas comunes, como
 - floor, round, ceil => redondear
 - min, max => encontrar máximos y mínimos en un array
 - log2, log10, log => operar con logaritmos
 - random => generar un número aleatorio de 0 a 1
 - abs, sign, sqrt => calcular
 - sin, cos, asin, tan => trigonometría



Date

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

- Permite hacer operaciones con fechas
 - get* => getters de días, meses, horas...
 - set* => setters de días, meses, horas...
 - to* => devuelve un string



■ JSON

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

- Permite hacer operaciones con objetos json
 - parse => de string a objeto
 - stringify => de objeto a string



■ Set

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

- Permite hacer listas únicas de valores primitivos o referencias
 - add => añadir un elemento
 - delete => eliminar un elemento
 - has => comprobar que existe el elemento en el set
 - ...



Error

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

- Permite gestionar errores en tiempo de ejecución. Para poder usarlo correctamente necesitamos la construcción try & catch

```
try {
  throw new Error("Whoops!");
} catch (e) {
  console.error(`s{e.name}: ${e.message}`);
}
```



Sintaxis más avanzada



Spread operator

```
// Spread Operator ...

//Combine arrays
let mid = [3, 4];
let arr = [1, 2, ...mid, 5, 6]; // [1, 2, 3, 4, 5, 6]

//Copy arrays
let arr = [1,2,3];
let arr2 = [...arr]; // like arr.slice()
arr2.push(4) // [1, 2, 3, 4]

//Calling functions Apply
const arr = [2, 4, 8, 6, 0];
const max = Math.max(...arr);

console.log(max); // 8
```

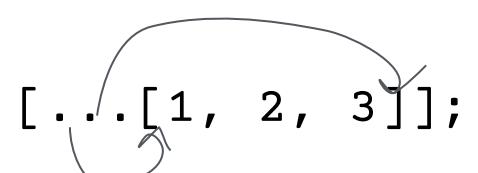
<https://medium.com/@serbanmihai/javascript-es6-cheatsheet-spread-operator-d39089ae24a1>



Spread operator

```
let arr = [1, 2, 3];

let newArr = [...arr]; // newArr === [1, 2, 3]

// paso a paso sería algo así
let newArr = [...[1, 2, 3]]; // internal step 1

let newArr = [...[1, 2, 3]]; // internal step 2
let newArr = [...[1, 2, 3]]; // internal step 2
let newArr = [1, 2, 3]; // internal step 3
```

ESTO OCURRE TANTO PARA **ARRAYS** COMO PARA **OBJETOS**! POR LO TANTO **NO LOS PODEMOS MEZCLAR!!**



■ Destructuring

```
// Destructuring objects
let person = {first_name: 'Joe', last_name: 'Appleseed'};
let {first_name, last_name} = person; //first_name = 'Joe', last_name = 'Appleseed'

// Destructuring arrays
const iterable = ['a', 'b'];
const [x, y] = iterable; // x = 'a'; y = 'b'

// Destructuring for-of loop
const arr = ['a', 'b'];
for (const [index, element] of arr.entries()) {
  console.log(index, element);
  // 0 a
  // 1 b
}
```

<https://medium.com/@serbanmihai/javascript-es6-cheatsheet-spread-operator-d39089ae24a1>



■ Operador ternario

```
let age = 18;  
let message;  
  
if (age >= 16) {  
    message = 'You can drive.';  
} else {  
    message = 'You cannot drive.';  
}  
  
console.log(message);
```

Opcion A

```
let age = 18;  
let message;  
  
age >= 16 ? (message = 'You can drive.') : (message = 'You cannot drive.');//  
  
console.log(message);
```

Opcion B

```
let age = 18;  
let message;  
  
message = age >= 16 ? 'You can drive.' : 'You cannot drive.';  
  
console.log(message);
```



■ Y más...

<https://www.javascripttutorial.net/es-next/>





■ Asincronía



J.Baños



■ Asincronía

*“JavaScript has certain characteristics that make it very different than other dynamic languages, namely that it has no concept of threads. Its model of concurrency is completely based around **events**. ”*

Ryan Dahl - El de Node





Callbacks

■ Controlando el tiempo y callbacks

```
function callback() {  
  console.log('Termine!');  
}  
  
console.log('Empiezo');  
setTimeout(callback, 2000);
```

setTimeout

Permite ejecutar una función pasados un tiempo que se indica en milisegundos.

- A la función que se pasa como parámetro en ambos casos se le llama callback.
- La idea de **un callback es “dejar programado lo que queremos que pase” cuando suceda algo**: que pase un tiempo, que el usuario haga click en un botón, que la base de datos me responda con los resultados, que me termine de descargar unos datos de internet....



```
let times = 0  
const intervalID = setInterval(() => {  
  console.log('Beeeeeeep! 🎙')  
  times++  
  if (times > 2500) {  
    clearInterval(intervalID) // finaliza el interval  
  }  
}, 1000)
```

setInterval

Permite ejecutar una función periódicamente cada un tiempo que se indica en milisegundos.







Promesas

Promesas / Promises

```
function doYouLoveMe(name) {  
  return new Promise(function(resolve, reject) {  
    console.log('Let me think about it...')  
    setTimeout(() => {  
      if (name = 'Alberto') {  
        resolve('Yes!')  
      } else {  
        reject('Sorry but no...')  
      }  
    }, 1000)  
  })  
}
```

```
const lovePromise = doYouLoveMe('Alberto')  
lovePromise.then(response => {  
  console.log(response)  
}).catch(error => {  
  console.error(error)  
})
```

```
const lovePromise = doYouLoveMe('Alberto')  
lovePromise.then(response => {  
  console.log(response)  
}, error => {  
  console.error(error)  
})
```

- Una promesa es un objeto que representa una operación que aún no se ha completado, pero que se completará más adelante.
- Tiene tres estados posibles:
 - Pendiente (pending)
 - Completada (settled)
 - Satisfactoriamente (fulfilled): cuando ha ido todo bien
 - Rechazada (rejected): cuando algo ha ido mal





ONE DOES NOT SIMPLY

LEARN PROMISES



ASYNC/AWAIT

YOU SHALL NOT SEEM ASYNCHRONOUS



Ignasi Marimon-Clos
@ignasi35

...

Async await
Async await
Async await
Async await
In the jungle
The mighty jungle
The lion sleeps tonight.



■ Async/await

```
function loveChecker() {  
  doYouLoveMe('Alberto').then(response => {  
    console.log('Alberto', response) ←  
  }, error => {  
    console.error('Alberto', error) ←  
  })  
  doYouLoveMe('Frodo').then(response => {  
    console.log('Frodo', response) ←  
  }, error => {  
    console.error('Frodo', error) ←  
  })  
}
```

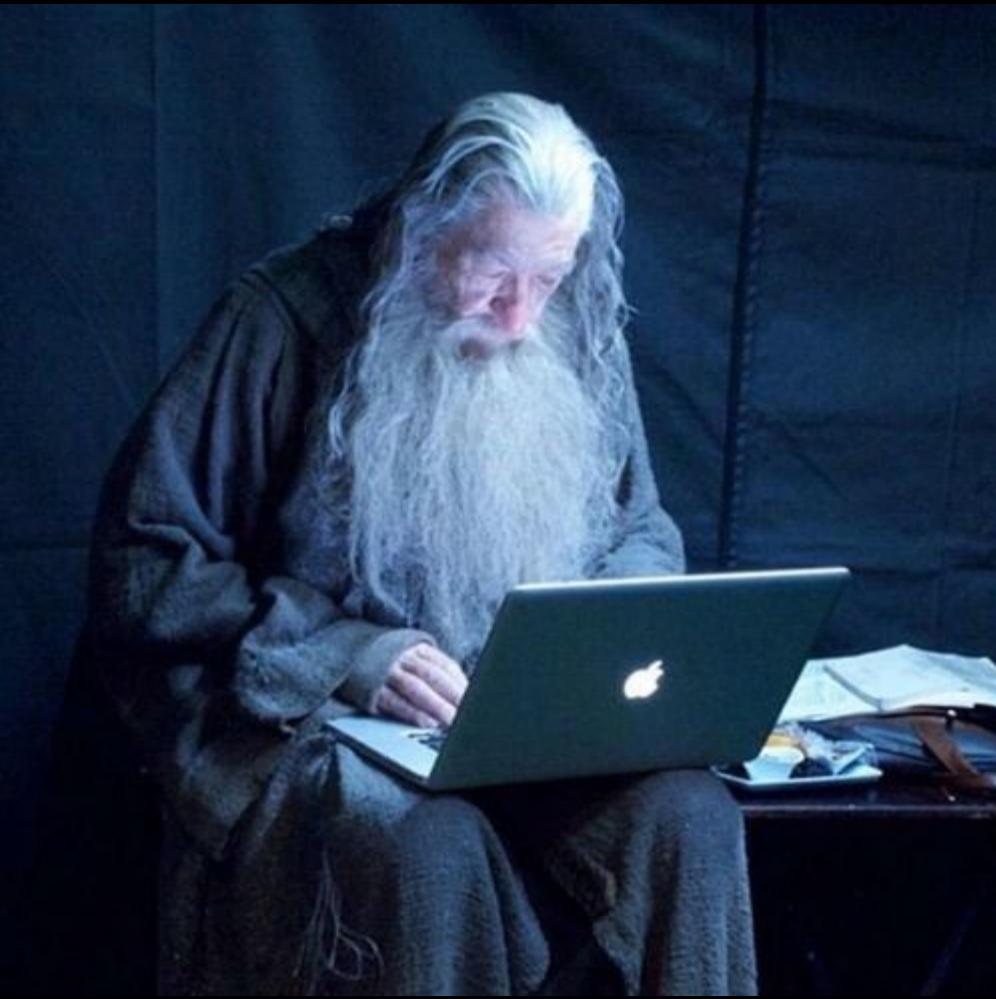
```
async function loveChecker() {  
  try {  
    let response = await doYouLoveMe('Alberto') →  
    console.log('Alberto', response)  
  } catch (error) {  
    console.error('Alberto', error)  
  }  
  try {  
    response = await doYouLoveMe('Frodo') →  
    console.log('Frodo', response)  
  } catch (error) {  
    console.error('Frodo', error)  
  }  
}
```

- Con async/await conseguimos que nuestro código parezca síncrono.
- Al usar **async** en una función, JS **envuelve la función en una promesa**
- Al usar **await**, JS **maneja el callback de la promesa por nosotros**.
- A esto (que el lenguaje haga cosa por nosotros usando ciertas palabras) se le llama sugar syntax.



A close-up portrait of a woman with long, dark, wavy hair. She has a serious, intense expression, looking directly at the viewer. Her skin tone is light, and she has dark eyes. The background is filled with bright, orange and yellow flames, suggesting a fire or explosion. The overall mood is dramatic and intense.

It's done.



A wizard is never late, nor is he early. He arrives precisely when ~~he means to~~

the promise is resolved.

A photograph of five young women of diverse ethnicities standing in a row outdoors. They are all smiling and looking towards the camera. The woman on the far left has curly brown hair and is wearing a white t-shirt with a graphic design. The second woman from the left has dark curly hair and is wearing a light-colored t-shirt. The third woman has straight brown hair and is wearing a light-colored t-shirt. The fourth woman has long dark hair and is wearing a dark green t-shirt. The woman on the far right has short blonde hair and is wearing a dark green t-shirt.

¡GRACIAS!