

## Declaración de variables: var, let, const

	scope	reassignable	redeclarable	mutable	hoisting	TDZ
// var	function	sí	sí	sí	sí	no
// let	block	sí	no	sí	sí	sí
// const	block	no	no	sí	sí	sí

```
function foo() { // comienzo scope de function y de block (A)
  var a = 1; let b = 2; const c = 3;
  if (true) { // comienzo nuevo scope de block (B)
    var a = 10; // scope: function, redeclara a
    let b = 20; // scope: block, no redeclara pq scope es bloque (llaves)
    const c = 30; // scope: block, no redeclara pq scope es bloque (llaves)
    console.log(a, b, c); // 10 20 30
  } // fin scope de block (B)
  console.log(a); // 10 - scope: function. Valor en condición if
  console.log(b); // 2 - scope: block. Valor 1ª línea function.
  console.log(c); // 3 - scope: block. Valor 1ª línea function.
} // fin scope de function y de block (A)
```

```
console.log(a); // undefined - hoisting
var a = 1; let b = 2;
let b = 200; // SyntaxError - no redeclarable
```

```
const a = {prop:3}; a.prop = 30; // {c:30} - mutable.
const b = [1, 2, 3]; b.push(4); // [1, 2, 3, 4] - mutable.
const c = 3;
c = 30; // TypeError - no reassignable
```

let a = 1; <pre>{   console.log(a); // 1 }</pre>	let a = 1; <pre>{   console.log(a); // ReferenceError - no inicializada   let a = 2; // hoisting y TDZ }</pre>
--	--

## Object property shorthand

// Valor de propiedad de objeto igual a valor de variable declarada anteriormente

```
const name = 'John'; const surname = 'Snow'; const age = 29;
const obj = {name, surname, age};
console.log(obj.name, obj.surname, obj.age) // John Snow 29
function getKing(name, surname, age) { return {name, surname, age}; }
getKing('Joffrey', 'Baratheon', 16); // {name: "Joffrey", surname: "Baratheon", age: 16}
```

## Spread operator

// Expande los elementos de un iterable (ejem. array) en un array

```
const arrA = ['a', 'b', 'c'];
const arrB = [...arrA, 'd', 'e', 'f']; // ['a', 'b', 'c', 'd', 'e', 'f']
```

```
let str = 'Hello'; // String, Array, TypedArray, Map y Set son objetos iterables
console.log( [...str] ); // ['H', 'e', 'l', 'l', 'o']
```

```
const arrA = [{a:true}, [1, 2, 3]];
const arrB = [...arrA]; // copia de array
arrB[0].a = true;
arrB[1].push(1); // objetos dentro del array mantienen las referencias
arrA[0].a === arrB[0].a && arrA[1].length === arrB[1].length // true
```

## Arrow Functions =>

// Arrow functions son funciones anónimas. No tienen su propio arguments.  
 // Usaría arguments.length de su función contenedora (si la tuviera)  
 // No tienen su propio this, usan el this del contexto (lexical this).  
 // No puede ser usado como constructor.

```
const double = x => x * 2; // implicit return (quitar llaves y return, una sola línea). Si hay solo un parámetro se pueden quitar los paréntesis
const doublePi = () => {
  let pi = 3.1416;
  return pi * 2;
}; // Si no hay parámetros los paréntesis son obligatorios
const foo = (x, y) => ({ a: x, b: y }); // para retornar objetos literales usando la sintaxis simplificada es necesario encerrar el objeto literal entre paréntesis
```

## Default parameters

// Posibilidad de dar valores por defecto a los parámetros de una función

```
function foo(a = 1, b = 2*a) {
  console.log(a, b);
}
foo(4); // 4 8 - valor por defecto usado cuando hay omisión o undefined
foo(4, 0); // 4 0
foo(undefined, 3); // 1 3;
```

## Rest parameters

// Posibilidad de expresar un número no definido de parámetros como un array

```
function foo(...a) {
  console.log(a);
}
foo('a', 'b', 'c'); // ['a', 'b', 'c']
```

## Destructuring objects and arrays

// Destructuring permite extraer valores de arrays y objetos (o cualquier iterable)

```
const arr = [1, 2]; // asignación de variables
let [a, b] = arr; // a = 1; b = 2;
```

```
const arr = [1, 2, 3, 4]; // se puede usar el rest operator
let [a, b, ...c] = arr; // a = 1; b = 2; c = [3, 4];
```

```
const arr = [1, 2]; // valores por defecto
let [a = 3, b = 4, c = 5] = arr; // a = 1; b = 2; c = 5;
```

```
let a = 1, b = 2; // intercambiar (swap) valores
[b, a] = [a, b] // a = 2; b = 1;
```

```
function getKing () {
  return {kingName: 'John', kingSurname: 'Snow'}
}
let {kingName = 'Robb', kingSurname = 'Stark'} = getKing(); // valores por defecto
const queen = {name: 'Daenerys', surname: 'Targaryen'};
let {name: queenName, surname: queenSurname} = queen; // renombrado de variables
// queenName = 'Daenerys'; queenSurname = 'Targaryen'; // name-ReferenceError
```

## Template Literals

// Nueva sintaxis para strings, con backticks (`), que permite multilinea  
 // e interpolación mediante \${}

```
const user = {name: 'John', surname: 'Snow'};
let userData = `
  nombre: ${user.name}
  apellidos: ${user.surname}
  fecha actual: ${new Date().toLocaleDateString()}
`;
```

## Tagged template literals

// Funciones usadas como prefijos de un template literal.  
 // El primer parámetro de esta función es un array de los strings que aparecen  
 // entre las variables interpoladas.  
 // Los siguientes parámetros son las variables interpoladas usadas.

```
function creditsFormat(stringsArr, credits) {
  return `${stringsArr[0]}<strong${credits > 0 ? ' ' : ' '}<br>
  class="zero">${credits}</strong>${stringsArr[1]}`;
}
var credits = 5;
var creditsMsg = creditsFormat`Dispones de ${credits} créditos`;
// Dispones de <strong>5</strong> créditos
```

## Array

// Uso de higher-order functions para operaciones normalmente realizadas con bucles.

```
const arr = [1, 2, 3, 11, 20];
// Métodos de iteración (higher-order functions)
// arr.method(callback(currentValue[, index[, array]]), [thisArg])
arr.every(elem => elem > 10); // false - ¿todos los elementos > 10?
arr.some(elem => elem > 10); // true - ¿algún elemento > 10?
arr.filter(elem => elem > 10); // [11, 20] - elementos > 10
arr.find(elem => elem > 10); // 11 - primer elemento > 10 sino undefined
arr.findIndex(elem => elem > 10); // 11 - índice del primer elemento > 10 sino -1
arr.forEach(elem => fn(elem)); // ejecuta función por cada elemento del array
arr.map(elem => elem + 10); // nuevo array con valor de cada elemento igual al
// return de la función aplicada a cada elemento del array original
```

```
// arr.method(callback(accumulator, currentValue[, index[, array]]), [initialValue])
arr.reduce((acc, elem) => elem + acc); // 37 - reduce array a un único valor como
// resultado de ejecutar una función por cada iteración en la que además del
// valor del array que se está procesando se usa el retorno de la iteración
// anterior (acumulador)
arr.reduceRight((acc, elem) => elem + acc); // 37 - igual que reduce pero
cambiando orden
```

```
const arr = [1, 2, 3, 11, 20];
// Otros métodos
arr.includes(11); // true - ¿algún elemento === 2?
arr.isArray(); // true - determina si es un array
arr.indexOf(11); // 3 - índice del primer elemento === 11 sino -1
```

```
const arr = [1, 2, 3, 11, 20];
// Métodos que retornan iteradores
let it = arr.entries(); // retorna iterador con array [clave, valor] como value
let it2 = arr.keys(); // retorna iterador con clave como value
let it3 = arr.values(); // retorna iterador con valor como value
```

## Promises

// Las promesas son objetos que pueden retornar un valor en el futuro  
 // Usados en programación asíncrona. Por ejemplo permite  
 // El primer parámetro del constructor de una promesa es la función callback  
 // usada en then cuando todo esta ok. El segundo es la función usada  
 // en catch cuando hay algún error.

```
function fetchUser(url) { // función que retorna promesa
    return new Promise((resolve, reject) => { // dos funciones callback como
        parámetros para ok y ko
            fetch(url).then(response => response.json()) // algunas apis nativas
            como fetch retornan promesas
                .then(data => resolve(data.results[0])) // ok - los métodos then son
                encadenables
                .catch(error => reject(new Error(error))); // ko
        });
    }

    fetchUser('https://randomuser.me/api/?results=1') // llamada a función que
    retorna promesa
        .then(result => console.log(result)) // si retorno es ok
        .catch(error => console.log('Error fetchUser: ' + error)); // si retorno es ko

    // Métodos estáticos de Promise:
    // Promise.all (se pasa un array de promesas y se ejecuta solo cuando todas
    // se han resuelto),
    // Promise.race (mismo que all pero se ejecuta en cuanto se resuelva una)

    Promise.all([
        fetchUser('https://randomuser.me/api/?results=1&nat=es&gender=female'),
        fetchUser('https://randomuser.me/api/?results=1&nat=es&gender=male')
    ])
        .then(result => console.log(result))
        .catch(error => console.log('Error fetchUser: ' + error));
}
```

## Generators

// Un generator es una función que puede pausarse y reanudarse en cualquier  
 // momento. Se declara con un asterisco \* y las pausas mediante la palabra  
 // yield. Las funciones generadoras retornan un objeto iterable

```
function* vidasExtraFn() {
    yield 2;
    yield 1;
}

function* vidas() {
    yield 3;
    yield 2;
    let vidasExtras = yield 1; // valor true recibido de next
    if (vidasExtras) {
        yield * vidasExtraFn(); // llamada a otra función generadora
    }
}

const play = vidas();
const v = 'Nº Vidas: ';

console.log(v + play.next().value); // Nº vidas: 3
console.log(v + play.next().value); // Nº vidas: 2
console.log(v + play.next().value); // Nº vidas: 1
console.log(v + play.next(true).value + ' extras!'); // Nº vidas: 2 extras!
console.log(v + play.next().value); // Nº vidas: 1
console.log(v + '0. Game over.');
```

## Async Await

// async define una función como asíncrona que retorna una promesa.  
 // En una función asíncrona la expresión await indica que se pause la  
 // ejecución de la función hasta que no se resuelva la promesa asignada

```
// Aquí se usa la función fetchUser del apartado Promises

async function getUsers() {
    const userA =
    fetchUser('https://randomuser.me/api/?results=1&nat=es&gender=female');
    const userB =
    fetchUser('https://randomuser.me/api/?results=1&nat=es&gender=male');
    return [await userA, await userB]; // dos pausas con await
}

getUsers()
    .then(result => console.log(result))
    .catch(error => console.log('Error fetchUser: ' + error));
```

## Object literal

// Mejoras en la creación de objetos con la notación literal

```
const name = 'Bob';
function getday() {
    return (new Date()).toLocaleDateString();
}

let obj = {
    // declaración abreviada de propiedades
    name, // = name : name
    getday, // = getday : getday
    _visits: 0,
    greet(nameParam) { // declaración abreviada de métodos
        return 'Hi ' + nameParam;
    },
    get visits() { // getters
        this._visits++;
        return this._visits;
    },
    set visits(num) { // setters
        console.log('Nº visitas modificado');
        this._visits = num;
    },
    ['ageOf' + name]: 50 // Computed Property Names
}

obj.visits // 1 - llamada a getter
obj.visits // 2 - llamada a getter
obj.visits = 0 // Nº visitas modificado 0 - llamada a setter
obj.ageOfBob // 50 - Computed Property Name
```

## Miscelanea

```
const obj = {a:1, b:2, c:3};
Object.values(obj); // [1, 2, 3] - array con valores de propiedades enumerables
Object.entries(obj); // [['a', 1], ['b', 2], ...] - array de pares clave valor
const obj2 = { ...obj, a: 100 } // spread operator en objetos. Shallow copy
const obj3 = Object.assign({}, obj); // Equivalente a ejemplo anterior
for (let o in obj2) { // bucle sobre propiedades enumerables de un objeto
    console.log(o + ' : ' + obj2[o]);
}
for (let o of 'Hola') {console.log(o);} // bucle sobre objetos iterables
```

## Modules

// Un módulo por archivo. API de módulos son estáticos. Singletons.  
 // Imports son read-only, ¿binding to the variable?.  
 // Las variables expuestas en los módulos son 'bindeadas' cuando se importan.  
 // Si cambian en módulo, ese cambio será visible donde se halla importado.

<pre>// named exports ('x.js') export let a = 1; export let b = () =&gt; {...};  // named exports agrupados let a = 1; let b = () =&gt; {...}; export {a, b}</pre>	<pre>// import named exports import {a} from 'x.js' // or import {a, b as b1} from 'x.js' // alias - or import * as foo from 'x.js' // * todo  console.log(a); console.log(b1); console.log(foo.a); a = 2 // TypeError - read-only</pre>
<pre>// named exports alias const a = 1; let b = function () {...}; export {a as a1, b}</pre>	<pre>import {a1, b} from 'x.js'</pre>

<pre>// default export, uno por archivo let a = function() {...}; export default a;</pre>	<pre>// import default export import m2 from './x.js'; m2();</pre>
<pre>// combinado con named exports let a = function() {...}; export let b = 2; export default a; // or</pre>	<pre>import m2, {b} from './x.js'; m2(); console.log(b);</pre>

```
export * from 'y.js'; // reexportar
```

## Proxies

// Intercepta operaciones sobre datos Object (objetos, funciones, arrays...)  
 // Creación: new Proxy(target, handler). Target es el objeto a interceptar y  
 // handler es el objeto que usaremos para manejar los eventos interceptados  
 // mediante métodos llamados traps (get, set, has, delete, apply, ...)

```
const reverenceHandler = {
    get: function(target, prop, receiver) { // trap get
        if (prop === 'name') {
            return `${target.rol} ${target.name}`;
        } else {
            return Reflect.get(target, prop); // retorna original
        }
    }
};

let fighter = {name: 'John', surname: 'Snow', rol: 'King'};
const signatureProxy = new Proxy(fighter, reverenceHandler);

signatureProxy.name; // "King John" - acceso interceptado y modificada respuesta
fighter.name; // "John" - acceso normal
```

## Referencias

// Mozilla Developer Network: <https://developer.mozilla.org/es/docs/Web/JavaScript>  
 // Dr. Axel Rauschmayer: <http://exploringjs.com/es6/>  
 // Kyle Simpson: <https://github.com/getify/You-Dont-Know-JS>  
 // Manuel Beaudru: <https://mbeaudru.github.io/modern-js-cheatsheet/>  
 // kangax es6 table: <http://kangax.github.io/compat-table/es6/>