

¿Qué es ECMASCIPT?

ECMA es el nombre de la asociación europea de fabricantes de computadoras (*European Computer Manufacturer Association*). Se trata de una organización sin ánimo de lucro que se encarga, entre otras cosas, de regular el funcionamiento de muchos estándares de la industria mundial.

Netscape envió el borrador de JavaScript a [Ecma International](#) para su estandarización y para que trabajasen sobre su especificación ECMA-262, que comenzó en noviembre de 1996.⁷

JavaScript es una de las implementaciones del estándar ECMA-262, en concreto la que se usa en los navegadores.

Por eso, cuando en la web se habla de ECMASCIPT y JavaScript en realidad se está hablando de la misma cosa. El primero es el estándar que define cómo debe funcionar el lenguaje, y el segundo es una implementación concreta de lo que indica la especificación estándar.

Además cada navegador tiene su propia implementación de ECMASCIPT, es decir, su propio motor de JavaScript.

<http://es6-features.org/>

Navegador	Motor/Implementación	Versión de ECMASCIPT
Google	V8	6
Chrome		
Firefox	SpiderMonkey	5.1 con muchas cosas de 6 y 7
Edge	Chakra	5.1 con muchas cosas de 6 y 7
Safari	JavaScriptCore - Webkit	5.1 con muchas cosas de 6 y 7
Internet	Jscript 9.0	5.1
Explorer		

<https://kangax.github.io/compat-table/es6/>

Funciones

Funciones arrow

La **expresión de función flecha** tiene una sintaxis más corta que una expresión de función convencional y no vincula sus propios `this`, `arguments`, `super`, o `new.target`. Las funciones flecha siempre son **anónimas**. Estas funciones son funciones no relacionadas con métodos y no pueden ser usadas como constructores.

```
<script>
  const fn = () => {
    console.log("arrow function");
  };

  const fn1 = () => console.log("ola");
</script>
```

Funciones

Parámetros por defecto

Podremos definir parámetros por defecto a nuestros argumentos.

```
<script>
  function makeRequest(url, timeout = 2000, callback = function() {}) {
    // the rest of the function
  }
</script>
```

Funciones

Parametros rest

La sintaxis de los **parámetros rest** nos permiten representar un número indefinido de argumentos como un array.

```
<script>
    function restParam(a, b, ...rest) {
        console.log(a); // 1
        console.log(b); // 2
        console.log(rest); // [3, 4, 5, 6]
    }

    restParam(1, 2, 3, 4, 5, 6);
</script>
```

Funciones

Spread operator

La sintaxis extendida o spread syntax permite a un elemento iterable tal como un arreglo o cadena ser expandido, sería lo contrario a los parámetros rest, ya que de un array te genera una lista de parámetros. También crea copias entre arrays y también entre objetos.

```
function spreadOper(a, b, c) {  
  console.log(a, b, c);  
}  
  
const numbers = [1, 2, 3];  
spreadOper(...numbers); // 1, 2, 3  
  
const arr1 = [1, 2, 3];  
const arr2 = [4, 5, 6];  
  
const arr3 = [...arr1, ...arr2];  
console.log(arr3); // [1, 2, 3, 4, 5, 6]  
  
const obj1 = { name: "sebastian" };  
const obj2 = { age: 32 };  
const obj3 = { ...obj1, ...obj2 };  
console.log(obj3); // {name: 'sebastian', age: 32}
```

Objetos

Definicion corta

Podremos definir los objetos de una manera corta, cuando el key y el value sean iguales.

```
// ES5
function createPerson(name, age) {
  return {
    name: name,
    age: age
  };
}

// ES6
function createPerson(name, age) {
  return {
    name,
    age
  };
}
```

Objetos

Métodos concisos

ECMAScript 6 también mejora la sintaxis para asignar métodos a literales de objetos.

```
// ES5
var person = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name);
  }
};

// ES6
var person = {
  name: "Nicholas",
  sayName() {
    console.log(this.name);
  }
};
```

Object

Object.assign()

El método **Object.assign()** se utiliza para copiar los valores de todas la propiedades enumerables de uno o más objetos fuente a un objeto destino. Retorna el objeto destino.

Sintaxis

```
Object.assign(objetivo, ...fuentes)
```

Parámetros

objetivo

El objeto destino.

fuentes

Los objetos origen.

Valor devuelto

El objeto destino.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }
```

Destructuring

Es bastante común definir objetos y arrays, y luego extraer sistemáticamente información relevante de esas estructuras. ECMAScript 6 simplifica esta tarea al agregar la desestructuración, que es el proceso de descomponer una estructura de datos en partes más pequeñas.

Object

```
// OBJECT DESTRUCTURING
let node1 = {
  type1: "Identifier",
  name1: "foo"
};

let { type1, name1 } = node1;

// OBJECT DESTRUCTURING - Definition
let node2 = {
  type2: "Identifier",
  name2: "foo"
};

let { type2, name2 = "luis", value = true } = node2;

// OBJECT DESTRUCTURING - Rename
let node3 = {
  type3: "Identifier",
  name3: "foo"
};
```

```
// OBJECT DESTRUCTURING - Nested
let node = {
  type: "Identifier",
  name: "foo",
  loc: {
    start: {
      line: 1,
      column: 1
    },
    end: {
      line: 1,
      column: 4
    }
  }
};

let {
  loc: { start }
} = node;

console.log(start.line); // 1
console.log(start.column); // 1
```

Deconstructing array

```
// ARRAY DESTRUCTURING
let colors = ["red", "green", "blue"];
let [firstColor, secondColor] = colors;

console.log(firstColor); // "red"
console.log(secondColor); // "green"

// ARRAY DESTRUCTURING - ESPACIO
let colors = ["red", "green", "blue"];
let [, , thirdColor] = colors;

console.log(thirdColor); // "blue"

// ARRAY DESTRUCTURING - ASIGNACION
let colors = ["red", "green", "blue"],
  firstColor = "black",
  secondColor = "purple";

[firstColor, secondColor] = colors;

console.log(firstColor); // "red"
console.log(secondColor); // "green"
```

```
// ARRAY DESTRUCTURING - INTERCAMBIO
let a = 1,
  b = 2;

[a, b] = [b, a];

console.log(a); // 2
console.log(b); // 1

// ARRAY DESTRUCTURING - DEFAULT VALUES
let colors = ["red"];

let [firstColor, secondColor = "green"] = colors;

console.log(firstColor); // "red"
console.log(secondColor); // "green"

// ARRAY DESTRUCTURING - NESTED
let colors = ["red", ["green", "lightgreen"], "blue"];
let [firstColor, [secondColor]] = colors;

console.log(firstColor); // "red"
console.log(secondColor); // "green"
```

```
// ARRAY DESTRUCTURING - REST ITEMS
let colors = ["red", "green", "blue"];

let [firstColor, ...restColors] = colors;

console.log(firstColor); // "red"
console.log(restColors.length); // 2
console.log(restColors[0]); // "green"
console.log(restColors[1]); // "blue"
```

Destructuring mixing and parameters

```
// MIXING DESTRUCTURING
let node = {
  type: "Identifier",
  name: "foo",
  loc: {
    start: {
      line: 1,
      column: 1
    },
    end: {
      line: 1,
      column: 4
    }
  },
  range: [0, 3]
};

let {
  loc: { start },
  range: [startIndex]
} = node;

console.log(start.line); // 1
console.log(start.column); // 1
console.log(startIndex); // 0
```

```
// DESTRUCTURING – PARAMETERS WITH DEFAULT VALUE
function setCookie(
  name,
  value,
  {
    secure = false,
    path = "/",
    domain = "example.com",
    expires = new Date(Date.now() + 360000000)
  } = {}
) {
  // code to set the cookie
}

setCookie("type", "js", {
  secure: true,
  expires: 60000
});
```

Classes

ES6 nos trae el soporte de clases a través el paradigma de clases

```
class CustomHTMLElement {
  constructor(element) {
    this.element = document.querySelector(element);
  }

  get html() {
    return this.element.innerHTML;
  }

  set html(value) {
    this.element.innerHTML = value;
  }

let p = new CustomHTMLElement('p');
p.html();
p.html('hola');
```

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

  sayName() {
    console.log(this.name);
  }
}

let person = new PersonClass("Nicholas");
person.sayName(); // "Nicholas"
```

let y const

Let y const son tipos de variables, y estas nos traen el soporte de bloque y no tienen elevación.
Cons a diferencia de let es un tipo de variable que no permite mutar el valor de la variable.

```
let name = 'sebastian';
const PI = 3.5;
```

Template string

Las plantillas de cadena de texto (template strings) son literales de texto que habilitan el uso de expresiones incrustadas. Es posible utilizar cadenas de texto de más de una línea, y funcionalidades de interpolación de cadenas de texto con ellas. También podemos ejecutar funciones entre el template string.

```
const name = 'sebastian';
const age = 33;

function fn() {
  return `y esto es dentro de una funcion ${2+2}`
}

let template = `Hola mi nombre es ${name}
y tengo ${age} anios ${fn()}`;
```

Módulos

En ECMAScript 6, cuando se usa la sintaxis del módulo (importar / exportar), cada archivo se convierte en su propio módulo con un espacio de nombres privado. Las funciones y variables de nivel superior no contaminan el espacio de nombres global. Para exponer funciones, clases y variables para que otros módulos importen, puede usar la palabra clave de exportación.

Mediante las sentencias import y export , podremos importar y exportar módulos.

```
// module "my-module.js"
function cube(x) {
| return x * x * x;
}

const foo = Math.PI + Math.SQRT2;
var graph = {
  options: {
    color: "white",
    thickness: "2px"
  },
  draw: function() {
    console.log("From graph draw function");
  }
};
export { cube, foo, graph };

import { cube, foo, graph } from "my-module";
graph.options = {
  color: "blue",
  thickness: "3px"
};
graph.draw();
console.log(cube(3)); // 27
console.log(foo); // 4.555806215962888
```