Bloque 1

Programación con JavaScript

Fundamentos de JavaScript. Colecciones.

Programación con JavaScript
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

Index

Introducción

En esta primera semana, vamos a aprender los fundamentos de **JavaScript** en cuanto a sintaxis se refiere. Este documento será especialmente importante para quién no tenga base previa del lenguaje (pero sí de programación con otros lenguajes, ya que este no es un curso sobre fundamentos de programación). La versión de JavaScript utilizada durante el curso será ES2015 (ES6).

JavaScript es un lenguaje interpretado, ejecutado por un intérprete normalmente integrado en un navegador web (pero no sólo en ese contexto). Lo que se conoce como JavaScript es en realidad una implementación de ECMAScript, el estándar que define las características de dicho lenguaje. La versión más reciente de ECMAScript specification es **ES2017** (Junio del 2017) el cual es también nuevo y no está todavía implementado en todos los navegadores. En la versión **ES2015** (Junio del 2015, también conocida como ES6) se introdujeron numerosos cambios en el lenguaje y una modernización necesaria después de pasar muchos años sin apenas cambios desde la primera versión del lenguaje en 1997. La versión anterior fue **ES5** (Diciembre. 2009), y **ES5.1** (Junio 2011). Aquí puedes ver la compatibilidad de los distintos navegadores con las diferentes versiones.

JavaScript se verá durante los 3 primeros bloques del curso. En el cuarto bloque veremos herramientas muy útiles para gestionar proyectos JavaScript como NPM (Node Package Manager) y WebPack. Finalmente estaremos preparados para conocer TypeScript en el quinto módulo, que se puede definir como un superconjunto de JavaScript que añade tipado a las variables y métodos (number, string, float,…) y algunas características extra.

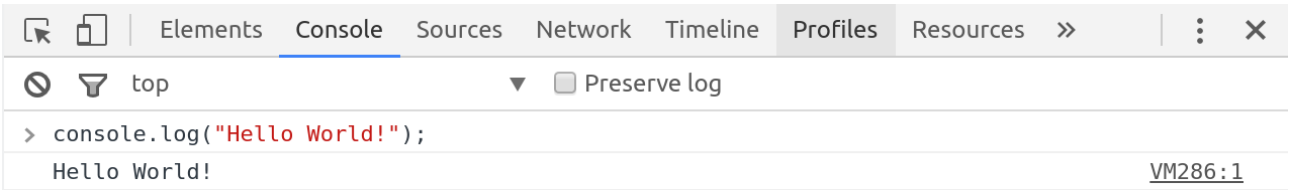
TypeScript está siendo cada vez más popular, y cada vez lo será más ya que se utiliza por defecto en frameworks tan populares como Angular. Cada vez más gente está optando por trabajar con esta variante del lenguaje por las ventajas de desarrollo en cuestión de programación orientada a objetos, autocompletado y detección temprana de errores que veremos a final del curso.

Editores y herramientas para programar

Para realizar este curso, puedes elegir cualquiera de los editores o IDE que prefieras. Sin embargo, lo idóneo sería que usaras el editor usado para los ejemplos (Visual Studio Code), porque todas las instrucciones serán dadas teniendo en cuenta este editor. Hay muchas opciones posibles, y al principio no es demasiado importante qué editor elijas.

La consola del navegador

Abre tu navegador (Te recomiendo Chrome or Firefox) y pulsa F12 para mostrar las herramientas de desarrollador que vienen integradas. Ve a la pestaña de Consola, y desde ahí puedes ejecutar instrucciones en JavaScript y visualizar el resultado de forma inmediata. Esta opción es una buena opción para *testear* pequeñas partes de código.



Por curiosidad, prueba este trozo de código donde podemos ver uno de los famosos bugs de JavaScript (la precisión decimal en algunas operaciones matemáticas)

```
> console.log(5.1 + 3.3);
  8.399999999999999
```

Editores de escritorio

Muchos editores de código e IDEs soportan las últimas versiones de la sintaxis de JavaScript, mientras que algunos de ellos soportan el código completo, integrándolo con jQuery, Angular, etc. Pudes usar aque con el que más cómodo te sientas (Visual Studio, Netbeans, Webstorm, Atom, Sublime, Kate, Notepad++, …), pero todos los ejemplos que ponga serán en Visual Studio Code, porque se integra muy bien con JavaScript, Node, Typescript, y Angular.

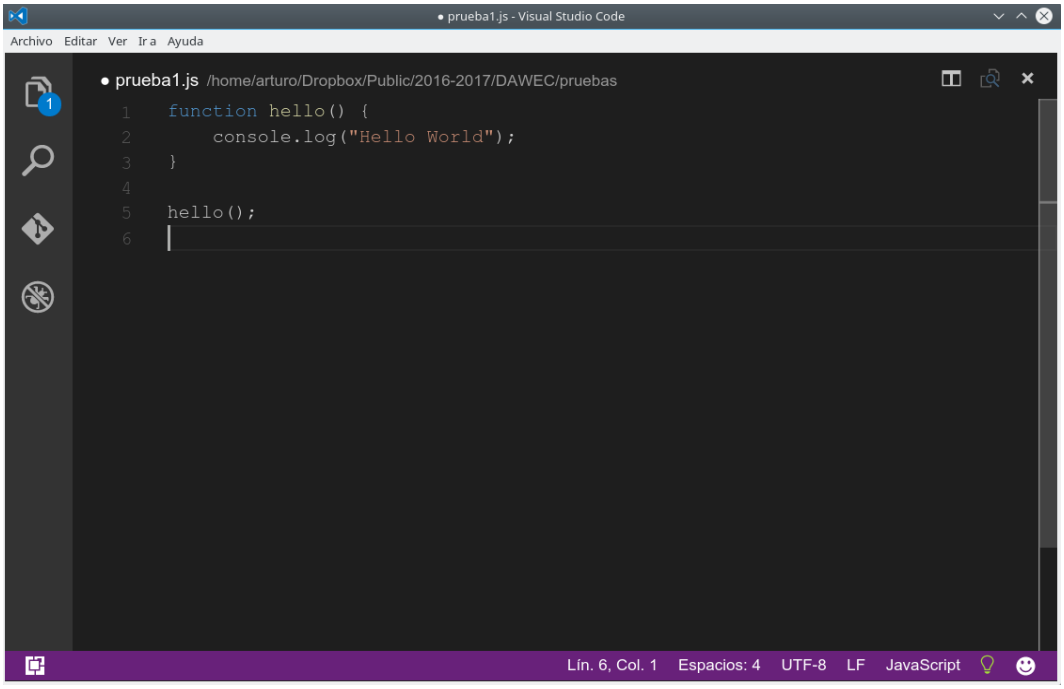
Visual Studio Code

Este editor (con algunas características de IDE) es desarrollado por Microsoft y de código abierto. Se encuentra disponible en Windows, Mac y GNU/Linux. Ha sido desarrollado utilizando Electron, que es un framework que usa **Chromium** y **Node.js** para desarrollar aplicaciones de escritorio utilizando HTML, CSS y JavaScript. Es decir, es una aplicación web de escritorio.

Visual Studio Code es un editor ligero, soporta resaltado de sintaxis, y autocompletado de código para **JavaScript** y **TypeScript** entre otros. También se integra muy bien con Angular y otros frameworks. Otros editores que integran TypeScript son: Visual Studio, Atom, Sublime Text, Webstorm, Emacs o Vim (muchos

de ellos mediante plugins). Visual Studio Code, es el editor que os recomiendo para este curso.

Puedes descargarlo de: https://code.visualstudio.com/Download



Editores web

Podemos usar editores web, al menos para probar código que no sea muy complejo o muy grande. Esta opción es bastante buena para probar código fácilmente sin necesidad de tener instalado un editor, y también para compartir código de forma rápida con otras personas. La parte mala es que no tienen muchas de las ventajas que presentan los editores de escritorio como autocompletado, o plugins, por ejemplo.

Dos famosos editores web son Fiddle (https://jsfiddle.net/) y Plunker (https://plnkr.co/). Puedes guardar tus proyectos y continuar el desarrollo más tarde en cualquier dispositivo.

Javascript, primeros pasos

Es probable que tengas ciertos conocimientos de Javascript, o incluso experiencia en el desarrollo de páginas web. Sin embargo, no es una mala opción hacer una revisión sobre algunos aspectos del lenguaje, esto ayudará a consolidar los conocimientos que tengas e incluso puedes aprender algo que no conocías, como las mejoras introducidas en la versión ES2015.

Integrando JavaScript con HTML

Para integrar el código JavaScript en nuestro HTML, necesitamos usar la etiqueta **<script>**. El sitio recomendado para poner la etiqueta es justo antes de cerrar la etiqueta **</html>**, para que algunos navegadores puedan cargar y construir el DOM antes de procesar el código JavaScript, de otro modo, el navegador se bloqueará el renderizado de la página hasta que se haya procesado todo el código JS.

¿Dónde podemos poner el código JS?:

Dentro de la etiqueta <script> (no recomendado)

```
<!DOCTYPE>
<html>
    <head>
        <title>Ejemplo JS</title>
    </head>
    <body>
         <p>Hola Mundo!</p>
         <script>
             console.log("Hola Mundo!");
         </script>
    </body>
</html>
```

En un archivo separado (recomendado)

```
Archivo: ejemplo1.html

<!DOCTYPE>
<html>
    <head>
        <title>Ejemplo JS</title>
    </head>
    <body>
         <p>Hola Mundo!</p>
         <script src="ejemplo1.js"></script>
    </body>
</html>

Archivo: ejemplo1.js

console.log("Hola Mundo!");
```

console.log() se utiliza a escribir por la consola del navegador aquello que nosotros le pasamos (F12 para abrir las herramientas de desarrollador y ver el resultado). Puedes usar el método **console.error()** para mostrar los errores.

La etiqueta <noscript> se utiliza para poner código HTML que será renderizado sólo cuando el navegador no soporte JavaScript o cuando haya sido desactivado. Esta etiqueta es muy útil para decirle al usuario que la web necesita tener JavaScript activado para funcionar correctamente, por ejemplo.

```
<!DOCTYPE>
<html>
    <head>
        <title>Ejemplo JS</title>
    </head>
    <body>
         <p>Hola Mundo!</p>
         <noscript>
            <h1>JavaScript no está activado. Por favor, actívelo o la aplicación
web no funcionará correctamente.</h1>
         </script>
    </body>
</html>
```

Variables

Puedes declarar una variable usando la palabra reservada **let** (también puedes usar **var**, pero no se recomienda desde la versión ES2015). El nombre de la variable deberá ser con el formato **CamelCase**, es decir, la primera letra en minúsculas, también puede empezar por subrayado (_nombre) o por dólar ($nombre). En JS las variables no tienen un tipo de dato explícito. El tipo puede cambiar internamente dependiendo de cual sea el valor que se le asigne. Esto significa que puedes asignar un string, y posteriormente un número.

```
let v1 = "Hola Mundo!";
console.log(typeof v1); // Imprime -> string

v1 = 123;
console.log(typeof v1); // Imprime -> number
```

¿Qué pasa si declaramos una variable pero no le asignamos un valor?. Hasta que no se le asigne un valor, tendrá un tipo especial conocido como **undefined**. Nota: Este valor es diferente de **null** (el cual se considera un valor).

```
let v1;
console.log(typeof v1); // Imprime -> undefined
if (v1 === undefined) { // (!v1) or (typeof v1 === "undefined") también funciona
  console.log("Has olvidado darle valor a v1");
}
```

¿Qué ocurre si se nos olvida poner **let** o **var**?. JavaScript declarará esa variable como **global**. Esto **NO** es lo recomendado porque las variables globales son peligrosas. Por tanto, es recomendable que usemos siempre **let** la primera vez que vayamos a necesitar una variable local.

Para evitar que se nos olvide declarar con **let**, podemos usar una declaración especial: '**use strict**' al comienzo de nuestro archivo JS. De este modo, no vamos a poder declarar variables globales omitiendo la palabra reservada **let**.

```
'use strict';
v1 = "Hola Mundo";
```

```
▶ Uncaught ReferenceError: v1 is not defined   example1.js:2
>
```

Constantes

Cuando a lo largo de una función o bloque, una variable no va a cambiar de valor, o cuando queremos definir un valor global inmutable (por ejemplo el número PI), se recomienda declararla como constante con la palabra reservada **const** en lugar de usar **let**. En el caso de las constantes globales se recomienda usar mayúsculas.

```
'use strict';
const MY_CONST=10;
MY_CONST=200;  → Uncaught TypeError: Assignment to constant variable.
```

Funciones

En JavaScript, declaramos funciones usando la palabra reservada **function** antes del nombre de la función. Los argumentos que le pasaremos a la función van dentro del paréntesis tras el nombre de la función (recuerda que no hay tipos de variable en JS). Una vez definida la función, entre llaves declaramos el cuerpo de la misma. El nombre de las funciones (como el de las variables) debe escribirse en formato **CamelCase** con la primera letra en minúsculas.

```
function sayHello(name) {
   console.log("Hello " + name);
}

sayHello("Tom"); // Imprime "Hello Tom"
```

No necesitas tener la función declarada antes de llamarla, esto es debido a que el intérprete de JavaScript primero procesa las declaraciones de variables y funciones y después ejecuta el resto del código.

Podemos llamar a una función enviándole más o menos parámetros de los establecidos en la declaración. Si le enviamos más parámetros, los sobrantes serán ignorados y si le enviamos menos, a los no recibidos se les asignará **undefined**.

```
sayHello(); // Imprime "Hello undefined"
```

Retorno de valores

Podemos usar la palabra reservada **return** para devolver un valor en una función. Si intentamos obtener un valor de una función que no devuelve nada, obtendremos **undefined**.

```
function totalPrice(priceUnit, units) {
   return priceUnit * units;
}

let total = totalPrice(5.95, 6);
console.log(total);  // Imprime 35.7
```

Funciones anónimas

La forma de declarar una función anónima es no asignarle ningún nombre. Podemos asignar dicha función como valor a una variable, ya que es un tipo de valor (como puede ser un *string* o número), por tanto, puede ser asignada a (o referenciada

desde) múltiples variables. Se utiliza igual que una función clásica.

```
let totalPrice = function(priceUnit, units) {
   return priceUnit * units;
}

console.log(typeof totalPrice); // Imprime "function" (tipo de la variable totalPrice)

console.log(totalPrice(5.95, 6)); // Imprime 35.7
let getTotal = totalPrice; // Referenciamos a la misma función desde la variable getTotal
console.log(getTotal(5.95, 6));  // Imprime 35.7. También funciona
```

Funciones lambda (o arrow functions)

Una de las funcionalidades más importantes que se añadió en ES2015 fue la posibilidad de usar las funciones lambda (o flecha). Otros lenguajes como C#, Java, etc. también las soportan. Estas expresiones ofrecen la posibilidad de crear funciones anónimas pero con algunas ventajas.

Vamos a ver las diferencias que tiene por creando dos funciones equivalentes (una anónima y otra lambda que hacen lo mismo):

```
let sum = function(num1, num2) {
   return num1 + num2;
}
console.log(sum(12,5)); // Imprime 17

let sum = (num1, num2) => num1 + num2;
console.log(sum(12,5)); // Imprime 17
```

Cuando declaramos una función lambda, la palabra reservada **function** no se usa. Si sólo se recibe un parámetro, los paréntesis pueden ser omitidos. Después de los parámetros debe ir una flecha (**=>**), y el contenido de la función.

```
let square = num => num * num;
console.log(square(3)); // Imprime 9
```

Si sólo hay una instrucción dentro de la función lambda, podemos omitir las llaves '{}', y **debemos omitir** la palabra reservada **return** ya que lo hace de forma implícita (devuelve el resultado de esa instrucción). Si hay más de una instrucción, usamos las llaves y se comporta como una función normal y por tanto, si devuelve algo, debemos usar la palabra reservada return.

```
let sumInterest = (price, percentage) => {
   let interest = price * percentage / 100;
   return price + interest;
}
console.log(sumInterest(200,15)); // Imprime 230
```

La diferencia más importante entre ambos tipos de funciones es el comportamiento de la palabra reservada **this**. Pero eso lo veremos en el tercer bloque del curso (Programación orientada a objetos).

Parámetros por defecto

Si un parámetro se declara en una función y no se pasa cuando la llamamos, se establece su valor como **undefined**.

```
function Persona(nombre) {
   this.nombre = nombre;

   this.diHola = function() {
      console.log("Hola! Soy " + this.nombre);
   }
}
let p = new Persona();
p.diHola (); // Imprime "Hola! Soy undefined"
```

Una solución usada para establecer un valor por defecto era usar el operador '||' (or), de forma que si se evalúa como undefined (false), se le asigna otro valor.

```
function Persona(nombre) {
   this.nombre = nombre || "Anónimo";
   ...
}
```

Sin embargo, en **ES2015** tenemos la opción de establecer un valor por defecto.

```
function Persona( nombre = "Anónimo") {
   this.nombre = nombre;
   ...
}
```

También podemos asignarle un valor por defecto basado en una expresión.

```
function getPrecioTotal(precio, impuesto = precio * 0.07) {
   return precio + impuesto;
}
console.log(getPrecioTotal(100)); // Imprime 107
```

Estructuras condicionales

La estructura **if** se comporta como en la mayoría de los lenguajes de programación. Lo que hace es evaluar una condición lógica, devolviendo un booleano como resultado y si es cierta, ejecuta el código que se encuentra dentro del bloque **if**. De forma optativa podemos añadir el bloque **else if**, y un bloque **else**.

```
let price = 65;

if(price < 50) {
   console.log("Esto es barato!");
} else if (price < 100) {
   console.log("Esto no es barato...");
} else {
   console.log("Esto es caro!");
}
```

La estructura **switch** tiene un comportamiento similar al de otros lenguajes de programación. Como sabemos, se evalúa una variable y se ejecuta el bloque correspondiente al valor que tiene (puede ser número, string,..). Normalmente, se

necesita poner la instrucción **break** al final de cada bloque, ya que de no ponerlo continuaría ejecutando las instrucciones que haya en el siguiente bloque. Un ejemplo donde dos valores ejecutarían el mismo bloque de código es el siguiente:

```
let userType = 1;

switch(userType) {
   case 1:
   case 2: // Tipos 1 y 2 entran aquí
      console.log("Puedes acceder a esta zona");
      break;
   case 3:
      console.log("No tienes permisos para acceder aquí");
      break;
   default: // Ninguno de los anteriores
      console.error("Tipo de usuario erróneo!");
}
```

En JavaScript (a diferencia de muchos otros lenguajes), puedes hacer que el **switch** se comporte como un **if**. Esto se hace evaluando un booleano (normalmente true) en lugar de otro tipo de valor, de forma que el **case** se evalúan condiciones:

```
let age = 12;

switch(true) {
   case age < 18:
      console.log("Eres muy joven para entrar");
      break;
   case age < 65:
      console.log("Puedes entrar");
      break;
   default:
      console.log("Eres muy mayor para entrar");
}
```

Bucles

Tenemos el típico bucle **while** que evalúa una condición y se repite una y otra vez hasta que la condición sea falsa (o si la condición es falsa desde un primer momento, no entra a realizar el bloque de instrucciones que contiene).

```
let value = 1;

while (value <= 5) { // Imprime 1 2 3 4 5
   console.log(value++);
}
```

Además de **while**, podemos usar **do..while**. La comprobación de la condición se realiza al final del bloque de instrucciones, por lo tanto, siempre se va a ejecutar el código al menos una vez.

```
let value = 1;

do { // Imprime 1 2 3 4 5
   console.log(value++);
} while (value <= 5);
```

El bucle **for** funciona igual que en otros lenguajes de programación. Inicializamos uno o más valores, establecemos la condición de finalización y el tercer apartado es para establecer el incremento o decremento.

```
let limit = 5;

for (let i = 1; i <= limit; i++) { // Imprime 1 2 3 4 5
   console.log(i);
}
```

Como sabrás, puedes inicializar una o más variables y también ejecutar varias instrucciones en cada iteración separándolas con comas.

```
let limit = 5;

for (let i = 1, j = limit; i <= limit && j > 0; i++, j--) {
   console.log(i + " - " + j);
}
/* Imprime
1 – 5
2 – 4
3 – 3
4 – 2
5 – 1
*/
```

Dentro de un bucle, podemos usar las instrucciones de **break** y **continue**. La primera de ellas saldrá del bucle de forma inmediata tras ejecutarse, y la segunda, irá a la siguiente iteración saltándose el resto de instrucciones de la iteración actual (ejecuta el correspondiente incremento si estamos dentro de un bucle **for**).

Tipos de datos básicos

Números

En JS no hay diferencia entre números enteros y decimales (float, double). El tipo de dato para cualquier número es **number**.

```
console.log(typeof 3); // Imprime number
console.log(typeof 3.56); // Imprime number
```

Además puedes imprimir números en notación científica (exponencial):

```
let num = 3.2e-3; // 3.2*(10^-3)
console.log(num); // Imprime 0.0032
```

Los números son objetos

En JavaScript todo es un **objeto**, incluso los valores primitivos que hay en otros lenguajes (Java, C++, etc.). Por ejemplo, si ponemos un punto después de escribir un número, podemos acceder a algunos métodos o propiedades.

```
console.log(3.32924325.toFixed(2)); // Imrpime 3.33
console.log(5435.45.toExponential()); // Imrpime 5.43545e+3
console.log((3).toFixed(2)); // Imrpime 3.00 (Un entero necesita estar dentro de un paréntesis para poder
```

acceder a sus propiedades)

Existe también un objeto global del lenguage llamado **Number**, donde podemos acceder a otras propiedades bastante útiles para trabajar con números.

```
console.log(Number.MIN_VALUE); // Imprime 5e-324 (El número más pequeño)
console.log(Number.MAX_VALUE); // Imprime 1.7976931348623157e+308 (El número más grande)
```

Hay también valores especiales para los números fuera de rango (**Infinito** y **-Infinito**). Podemos comparar si un número tiene uno de estos valores directamente usando === Infinity ó valor === -Infinity por ejemplo. Sin embargo, hay una función llamada **isFinite(value)** que nos devuelve falso cuando el valor es Infinity ó -Infinity.

```
console.log(Number.MAX_VALUE * 2); // Imprime Infinity
console.log(Number.POSITIVE_INFINITY); // Imprime Infinity
console.log(Number.NEGATIVE_INFINITY); // Imprime -Infinity
console.log(typeof Number.POSITIVE_INFINITY); // Imprime number

let number = Number.POSITIVE_INFINITY / 2; // Sigue siendo todavía infinito!!
if(isFinite(number)) { // Es igual que (number !== Infinity && number !== -Infinity)
   console.log("El número es " + number);
} else { // Enters here
   console.log("El número no es finito");
}
```

Operaciones con números

Podemos realizar las operaciones típicas (+,-,*,/,%,…). Pero, ¿Qué ocurre si uno de los operandos no es un número?. Por ejemplo, si un número se pone entre comillas, es considerado un string. Cuando hacemos una operación numérica con valores que no son números, intenta transformar esos valores a números (*cast* implícito). Si no puede, nos devuelve un valor especial llamado NaN (Not a Number).

```
let a = 3;
let b = "asdf";
let r1 = a * b; // b es “asdf”, y no será transformado a número
console.log(r1); // Imprime NaN

let c;
let r3 = a + c; // c es undefined, no será transformado a número
console.log(r3); // Imprime NaN

let d = "12";
console.log(a * d); // Imprime 36. d puede ser transformado al  número 12
console.log(a + d); // Imprime 312. El operador + concatena si hay un string
console.log(a + +d); // Imprime 15.  El operador ‘+’ delante de un valor lo transforma en numérico
```

Para comprobar si un número es NaN, se puede utilizar el método que devuelve un booleano (true si es NaN): **Number.isNaN(valor)**

undefined y null

En JavaScript cuando una variable (o parámetros de una función) han sido definida sin asignarle valor, se inicializa con el valor especial undefined.

No deberíamos confundir **undefined** con **null**. El segundo es un tipo de valor,

que explícitamente asignas a una variable. Vamos a verlo mediante un ejemplo.

```
let value; // Value no ha sido asignada (undefined)
console.log(typeof value); // Imprime undefined

value = null;
console.log(typeof value); // Imprime object
```

Como puedes ver, **null** es consderado un tipo de objeto (una referencia vacía). **undefined** es un valor especial que nos avisa que la variable no ha sido inicializada.

Boolean

En JS, los booleanos se representan en minúsculas (**true**, **false**). Puedes negarlos usando el operador **!** antes del valor.

Strings

Los valores string se representan dentro de 'comillas simples' o "comillas dobles". Podemos utilizar el operador + para concatenar cadenas.

```
let s1 = "Esto es un string";
let s2 = 'Esto es otro string';

console.log(s1 + " - " + s2); // Imrpime: Esto es un string – Esto es otro string
```

Cuando el string se encuentra dentro de comillas dobles, podemos usar comillas simples dentro y viceversa. Sin embargo, si quieres poner comillas dobles dentro de una cadena declarada a su vez dentro de comillas dobles, necesitas *escaparlas* para no cerrar el string previo, ocurriría lo mismo si fueran comillas simples.

```
console.log("Hello 'World'"); // Imprime: Hello 'World'
console.log('Hello \'World\''); // Imprime: Hello 'World'

console.log("Hello \"World\""); // Imprime: Hello "World"
console.log('Hello "World"'); // Imprime: Hello "World"
```

Como en el caso de los números, los strings son objetos y tienen algunos métodos útiles que podemos utilizar. Todos estos métodos no modifican el valor de la variable original a menos que la reasignes.

```
let s1 = "Esto es un string";
// Obtener la longitud del string
console.log(s1.length); // Imprime 17

// Obtener el carácter de una cierta posición del string (Empieza en 0)
console.log(s1.charAt(0)); // Imprime "E"

// Obtiene el índice de la primera ocurrencia
console.log(s1.indexOf("s")); // Imprime 1

// Obtiene el índice de su última ocurrencia
console.log(s1.lastIndexOf("s")); // Imprime 11

// Devuelve un array con todas las coincidencias en de una expresión regular
console.log(s1.match(/.s/g)); // Imprime ["Es", "es", " s"]
```

```
// Obtiene la posición de la primera ocurrencia de una expresión regular
console.log(s1.search(/[aeiou]/)); // Imprime 3

// Reemplaza la coincidencia de una expresión regular (o string) con un string (/g opcionalmente reemplaza todas)
console.log(s1.replace(/i/g, "e")); // Imprime "Esto es un streng"

// Devuelve un substring (posición inicial: incluida, posición final: no incluida)
console.log(s1.slice(5, 7)); // Imprime "es"

// Igual que slice
console.log(s1.substring(5, 7)); // Imprime "es"

// Como substring pero con una diferencia (posición inicial, número de caracteres desde la posición inicial)
console.log(s1.substr(5, 7)); // Imprime "es un s"

// Transforma en minúsculas, toLowerCase no funciona con caracteres especiales (ñ, á, é, ...)
console.log(s1.toLocaleLowerCase()); // Imprime "esto es un string"

// Transforma a mayúsculas
console.log(s1.toLocaleUpperCase()); // Imprime "ESTO ES UN STRING"

// Devuelve un string eliminando espacios, tabulaciones y saltos de línea del principio y final
console.log("   String con espacios   ".trim()); // Imprime "String con espacios"

// Devuelve si una cadena empieza por una determinada subcadena
console.log(str.startsWith("This")); // Imprime true

// Devuelve si la cadena acaba en la subcadena recibida
console.log(str.endsWith("string")); // Imprime true

// Devuelve si la cadena contiene la subcadena recibida
console.log(str.includes("is")); // Imprime true

// Genera una nueva cadena resultado de repetir la cadena actual N veces
console.log("la".repeat(6)); // Imprime "lalalalalala"
```

También, ahora los string soportan caracteres con más de dos bytes (4 caracteres hexadecimales), a veces llamados caracteres de plano astral en unicode (*astral plane)*, se deben escribir entre llaves \u{}. Ejemplo:

```
let uString = "Unicode astral plane: \u{1f3c4}";
console.log(uString); // Imprime "Unicode astral plane: 🏄" (icono del surfista)
```

Estos caracteres especiales devuelven un valor de dos caracteres cuando se mide la longitud de un string:

```
let surfer = "\u{1f3c4}"; // Un carácter: 🏄
console.log(surfer.length); // Imprime 2
```

Sin embargo, si transformamos un string en un array (de caracteres), este será dividido correctamente (cada carácter en una posición del array):

```
let surfer2 = "\u{1f30a}\u{1f3c4}\u{1f40b}"; // TRES caracteres: 🌊🏄🐋
console.log(surfer2.length); // Imprime 6
console.log(Array.from(surfer2).length); // Imprime 3 (convertido en array correctamente)
```

Template literals (sustitución de variables y multilínea)

Desde ES2015, JavaScript soporta string **multilínea** con **sustitución de variables**. Ponemos el string entre caracteres ` (backquote) en lugar de entre comillas simples o dobles. La variable (o cualquier expresión que devuelva un valor) va dentro de **${}** si se quiere sustituir por su valor.

```
let num = 13;

console.log(`Example of multi-line string
the value of num is ${num}`);
```

```
Example of multi-line string
the value of num is 13
```

Conversión de tipos

Puedes convertir un dato a **string** usando la función **String(value)**. Otra opción es concatenarlo con una cadena vacía, de forma que se fuerce la conversión

```
let num1 = 32;
let num2 = 14;

// Cuando concatenamos un string, el otro operando es convertido a string
console.log(String(32) + 14); // Imprime 3214
console.log("" + 32 + 14); // Imprime 3214
```

Puedes convertir un dato en **number** usando la función Number(value). Puedes también añadir el prefijo '+' antes de la variable para conseguir el mismo resultado.

```
let s1 = "32";
let s2 = "14";

console.log(Number(s1) + Number(s2)); // Imprime 46
console.log(+s1 + +s2); // Imprime 46
```

La conversión de un dato a **booleano** se hace usando la función Boolean(value).Puedes añadir **!!** (doble negación), antes del valor para forzar la conversión. Estos valores equivalen a false: **string vacío** (""), **null**, **undefined**, **0**. Cualquier otro valor debería devolver true.

```
let  v = null;
let  s = "Hello";

console.log(Boolean(v)); // Imprime false
console.log(!!s); // Imprime true
```

Ámbito de las variables

Variables globales

Cuando se declara una variable en el bloque principal (fuera de cualquier función), ésta es creada como global. Las variables que no son declaradas con la palabra reservada **let** son también globales (a menos que estemos usando el **strict mode**, que no permite hacer esto). Cuando ejecutamos JavaScript en un navegador, el objeto global **window** mantiene todas las variables globales (de hecho es implícito por defecto**: window.variable → variable**).

```
let  global = "Hello";
```

```
function cambiaGlobal() {
   global = "GoodBye";
}

cambiaGlobal();
console.log(global); // Imprime "GoodBye"
console.log(window.global); // Imprime "GoodBye"
```

Intenta declarar una variable global dentro de una función usando strict mode. Si no usas strict (Es recomendable que lo uses), podrías crear una variable global así.

```
'use strict';

function changeGlobal() {
   global = "GoodBye";
}

changeGlobal(); // Error → Uncaught ReferenceError: global is not defined
```

Variables definidas en funciones

Todas las variables que se declaran dentro de una función son locales.

```
function setPerson() {
   let  person = "Peter";
}

setPerson();
console.log(person); // Error → Uncaught ReferenceError: person is not defined
```

Si una variable global con el mismo nombre existe, la variable local no actualizará el valor de la variable global.

```
function setPerson() {
   let person = "Peter";
}

let person = "John";
setPerson();
console.log(person); // Imprime John
```

Operadores

Suma '+'

Este operador puede usarse para sumar números o concatenar cadenas. Pero, ¿Qué ocurre si intentamos sumar un número con un string, o algo que no sea un número o string?. Vamos a ver qué pasa:

```
console.log(4 + 6); // Imprime 10
console.log("Hello " + "world!"); // Imprime "Hello world!"
console.log("23" + 12); // Imprime "2312"
console.log("42" + true); // Imprime "42true"
console.log("42" + undefined); // Imprime "42undefined"
console.log("42" + null); // Imprime "42null"
console.log(42 + "hello"); // Imprime "42hello"
console.log(42 + true); // Imprime 43 (true => 1)
```

```
console.log(42 + false); // Imprime 42 (false => 0)
console.log(42 + undefined); // Imprime NaN (undefined no puede ser convertido a number)
console.log(42 + null); // Imprime 42 (null => 0)
console.log(13 + 10 + "12"); // Imprime "2312" (13 + 10 = 23, 23 + "12" = "2312")
```

Cuando un operando es string, siempre se realizará una **concatenación**, por tanto se intentará transformar el otro valor en un string (si no lo es). En caso contrario, intentará hacer una suma (convirtiendo valores no numéricos a número). Si la conversión del valor a número falla, devolverá **NaN** (Not a Number).

Operadores aritméticos

Otros operadores aritméticos son: **resta** (-), **multiplicación** (*), **división** (/), y **resto** (%),. Estos operadores operan siempre con números, por tanto, cada operando debe ser convertido a número (si no lo era previamente).

```
console.log(4 * 6); // Imprime 24
console.log("Hello " * "world!"); // Imprime NaN
console.log("24" / 12); // Imprime 2 (24 / 12)
console.log("42" * true); // Imprime 42 (42 * 1)
console.log("42" * false); // Imprime 0 (42 * 0)
console.log("42" * undefined); // Imprime NaN
console.log("42" - null); // Imprime 42 (42 - 0)
console.log(12 * "hello"); // Imprime NaN ("hello" no puede ser convertido a número)
console.log(13 * 10 - "12"); // Imprime 118 ((13 * 10) - 12)
```

Operadores unarios

En JavaScript podemos preincrementar (++variable), postincrementar (variable++), predecrementar (--variable) y postdecrementar (variable--).

```
let  a = 1;
let  b = 5;
console.log(a++); // Imprime 1 y incrementa a (2)
console.log(++a); // Incrementa a (3), e imprime 3
console.log(++a + ++b); // Incrementa a (4) y b (6). Suma (4+6), e imprime 10
console.log(a-- + --b); // Decrementa b (5). Suma (4+5). Imprime 9. Decrementa a (3)
```

También, podemos usar los signos – y + delante de un número para cambiar o mantener el signo del número. Si aplicamos estos operadores con un dato que no es un número, éste será convertido a número primero. Por eso, es una buena opción usar **+value** para convertir a número, lo cual equivale a usar **Number(value)**.

```
let  a = "12";
let  b = "13";
let  c = true;
console.log(a + b); // Imprime "1213"
console.log(+a + +b); // Imprime 25 (12 + 13)
console.log(+b + +c); // Imprime 14 (13 + 1). True -> 1
```

Operadores relacionales

El operador de comparación, compara dos valores y devuelve un booleano (true o false) Estos operadores son prácticamente los mismos que en la mayoría de lenguajes de programación, a excepción de algunos, que veremos a continuación.

Podemos usar == o === para comparar la igualdad (o lo contrario !=, !==). La principal diferencia es que el primero, no tiene en cuenta los tipos de datos que están siendo comparados, compara si los valores son equivalentes. Cuando usamos ===, los valores además deben ser del mismo tipo. Si el tipo de valor es diferente (o si es el mismo tipo de dato pero diferente valor) devolverá falso. Devuelve true cuando ambos valores son idénticos y del mismo tipo.

```
console.log(3 == "3"); // true
console.log(3 === "3"); // false
console.log(3 != "3"); // false
console.log(3 !== "3"); // true
// Equivalente a falso (todo lo demás es equivalente a cierto)
console.log("" == false); // true
console.log(false == null); // false (null no es equivalente a cualquier boolean).
console.log(false == undefined); // false (undefined no es equivalente a cualquier boolean).
console.log(null == undefined); // true (regla especial de JavaScript)
console.log(0 == false); // true
console.log({} >= false); // Object vacío -> false
console.log([] >= false); // Array vacío -> true
```

Otros operadores relaciones para números o strings son: menor que (<), mayor que (>), menor o igual que (<=), y mayor o igual que (>=). Cuando comparamos un string con estos operadores, se va comparando carácter a carácter y se compara su posición en la codificación Unicode para determinar si es menor (situado antes) o mayor (situado después). A diferencia del operador de suma (+), cuando uno de los dos operandos es un número, el otro será transformado en número para comparar. Para poder comparar como string, ambos operandos deben ser string.

```
console.log(6 >= 6); // true
console.log(3 < "5"); // true ("5" → 5)
console.log("adiós" < "bye"); // true
console.log("Bye" > "Adiós"); // true
console.log("Bye" > "adiós"); // false. Las letras mayúsculas van siempre antes
console.log("ad" < "adiós"); // true
```

Operadores booleanos

Los operadores booleanos son **negación** (!), **y** (&&), **o** (||). Estos operadores normalmente son usados de forma combinada con los operadores relacionales formando una condición más compleja, la cual devuelve true o false.

```
console.log(!true); // Imprime false
console.log(!(5 < 3)); // Imprime true (!false)

console.log(4 < 5 && 4 < 2); // Imprime false (ambas condiciones deben ser ciertas)
console.log(4 < 5 || 4 < 2); // Imprime true (en cuanto una condición sea cierta, devuelve cierta y deja de comparar)
```

Se puede usar el operador **y**, o el operador **o** con valores que no son booleanos, pero se puede establecer equivalencia (explicada anteriormente). Con el operador **or,** en encontrarse un true o equivalente, lo devolverá sin seguir evaluando el resto. El operador **and** al evaluar las condiciones, si alguna de ellas es falsa o equivalente no sigue evaluando. Siempre se devuelve la última expresión evaluada.

```
console.log(0 || "Hello"); // Imprime "Hello"
console.log(45 || "Hello"); // Imprime 45
```

```
console.log(undefined && 145); // Imprime undefined
console.log(null || 145); // Imprime 145
console.log("" || "Default"); // Imprime "Default"
```

Usamos la **doble negación !!** para transformar cualquier valor a booleano. La primera negación fuerza el casting a boolean y niega el valor. La segunda negación, vuelve a negar el valor dejándolo en su valor equivalente original.

```
console.log(!!null); // Imprime false
console.log(!!undefined); // Imprime false
console.log(!!undefined === false); // Imprime true
console.log(!!""); // Imprime false
console.log(!!0); // Imprime false
console.log(!!"Hello"); // Imprime true
```

El operador boolean || (or) puede ser usado la para simular valores por defecto en una función. Por ejemplo:

```
function sayHello(name) {
   // Si nombre es undefined o vacío (""), Se le asignará "Anonymous" por defecto
   let sayName = name || "Anonymous";
   console.log("Hello " + sayName);
}

sayHello("Peter"); // Imprime "Hello Peter"
sayHello(); // Imprime "Hello Anonymous"
```

Colecciones

Arrays

En JavaScript, los arrays son un tipo de objetos. Podemos crear un array con la instancia de un objeto de clase **Array**. Estos no tienen un tamaño fijo, por tanto, podemos inicializarlo con un tamaño y luego añadirle más elementos.

El constructor puede recibir 0 parámetros (array vacío), 1 número (el tamaño del array), o en cualquier otro caso, se creará un array con los elementos recibidos. Debemos tener en cuenta que en JavaScript un array puede contener al mismo tiempo diferentes tipos de datos: number, string, boolean, object, etc.

```
let a = new Array(); // Crea un array vacío
a[0] = 13;
console.log(a.length); // Imprime 1
console.log(a[0]); // Imprime 13
console.log(a[1]); // Imprime undefined
```

Fíjate que cuando accedes a una posición del array que no ha sido definida, devuelve undefined. La longitud de un array depende de las posiciones que han sido asignadas. Vamos a ver un ejemplo de lo que ocurre cuando asignas una posición mayor que la longitud y que no es consecutiva al último valor asignado.

```
let a = new Array(12); // Crea un array de tamaño 12
console.log(a.length); // Imprime 12
a[20] = "Hello";
console.log(a.length); // Ahora imprime 21 (0-20). Las posiciones 0-19 tendrán el valor undefined
```

Podemos reducir la longitud del array modificando directamente la propiedad de la longitud del array (**length)**. Si reducimos la longitud de un array, las posiciones mayores a la nueva longitud serán consideradas como undefined (borradas).

```
let a = new Array("a", "b", "c", "d", "e"); // Array con 5 valores
console.log(a[3]); // Imprime "d"
a.length = 2; // Posiciones 2-4 serán destruidas
console.log(a[3]); // Imprime undefined
```

Puedes crear un array usando corchetes en lugar de usar **new Array()**. Los elementos que pongamos dentro, separados por coma serán los elementos que inicialmente tendrá el array.

```
let a = ["a", "b", "c", "d", "e"]; // Array de tamaño 5, con 5 valores inicialmente
console.log(typeof a); // Imprime object
console.log(a instanceof Array); // Imprime true. a es una instancia de array
a[a.length] = "f"; // Insertamos in nuevo elemento al final
console.log(a); // Imprime ["a", "b", "c", "d", "e", "f"]
```

Recorriendo arrays

Podemos recorrer un array con los clásicos bucles while y for, creando un contador para el índice que iremos incrementando en cada iteración. Otra versión del

for, es el bucle **for..in**. Con este bucle podemos iterar los índices de un array o las propiedades de un objeto (similar al bucle foreach de otros lenguajes, pero recorriendo los índices en lugar de los valores).

```
let  ar = new Array(4, 21, 33, 24, 8);

let i = 0;
while(i < ar.length) { // Imprime 4 21 33 24 8
   console.log(ar[i]);
   i++;
}

for(let i = 0; i < ar.length; i++) { // Imprime 4 21 33 24 8
   console.log(ar[i]);
}

for (let i in ar) { // Imprime 4 21 33 24 8
   console.log(ar[i]);
}
```

Iterando las propiedades de un objeto (veremos objetos en el bloque 3):

```
let person = {
   nombre : "John",
   edad  : 45,
   telefono: "65-453565"
};

/**
 * Imprimirá:
 * nombre: John
 * edad: 45
 * telefono: 65-453565
 */
for (let field in person) {
   console.log(field + ": " + person[field]);
}
```

Desde ES2015 podemos iterar por los elementos de un array o incluso por los caracteres de una cadena sin utilizar el índice. Para ello se utiliza el bucle **for..of** (que se comporta como un bucle **foreach** de otros lenguajes).

```
let a = ["Item1", "Item2", "Item3", "Item4"];

for(let index in a) {
   console.log(a[index]);
}

for(let item of a) { // Hace lo mismo que el bucle anterior
   console.log(item);
}

let str = "abcdefg";

for(let letter of str) {
   if(letter.match(/^[aeiou]$/)) {
      console.log(letter + " es una vocal");
   } else {
      console.log(letter + " es una consonante");
   }
}
```

Métodos de arrays

Insertar valores al principio de un array (**unshift**) y al final (**push**).

```
let  a = [];
a.push("a"); // Inserta el valor al final del array
a.push("b", "c", "d"); // Inserta estos nuevos valores al final
console.log(a); // Imprime ["a", "b", "c", "d"]
a.unshift("A", "B", "C"); // Inserta nuevos valores al principio del array
console.log(a); // Imprime ["A", "B", "C", "a", "b", "c", "d"]
```

Ahora, vamos a ver la operación opuesta. Eliminar del principio (**shift**) y del final (**pop**) del array. Estas operaciones nos devolverán el valor que ha sido eliminado.

```
console.log(a.pop()); // Imprime y elimina la última posición → "d"
console.log(a.shift()); // Imprime y elimina la primera posición → "A"
console.log(a); // Imprime ["B", "C", "a", "b", "c"]
```

Podemos convertir un array a string usando **join()** en lugar de **toString()**. Por defecto, devuelve un string con todos los elementos separados por coma. Sin embargo, podemos especificar el separador a imprimir.

```
let  a = [3, 21, 15, 61, 9];
console.log(a.join()); // Imprime "3,21,15,61,9""
console.log(a.join(" -#- ")); // Imprime "3 -#- 21 -#- 15 -#- 61 -#- 9"
```

¿Cómo concatenamos dos arrays?. Usando **concat**.

```
let  a = ["a", "b", "c"];
let  b = ["d", "e", "f"];
let  c = a.concat(b);
console.log(c); // Imprime ["a", "b", "c", "d", "e", "f"]
console.log(a); // Imprime ["a", "b", "c"] . El array original no ha sido modificado
```

El método **slice** nos devuelve un nuevo *sub-array*.

```
let a = ["a", "b", "c", "d", "e", "f"];
let b = a.slice(1, 3); // (posición de inicio → incluida, posición final → excluida)
console.log(b); // Imprime ["b", "c"]
console.log(a); // Imprime ["a", "b", "c", "d", "e", "f"]. El array original no es modificado
console.log(a.slice(3)); // Un parámetro. Devuelve desde la posición 3 al final → ["d", "e", "f"]
```

splice elimina elementos del array original y devuelve los elementos eliminados. También permite insertar nuevos valores.

```
let  a = ["a", "b", "c", "d", "e", "f"];
a.splice(1, 3); // Elimina 3 elementos desde la posición 1 ("b", "c", "d")
console.log(a); // Imprime ["a", "e", "f"]
a.splice(1,1, "g", "h"); // Elimina 1 elemento en la posición 1 ("e"), e inserta "g", "h" en esa posición
console.log(a); // Imprime ["a", "g", "h", "f"]
a.splice(3, 0, "i"); // En la posición 3, no elimina nada, e inserta "i"
console.log(a); // Imprime ["a", "g", "h", "i", "f"]
```

Podemos invertir el orden del array usando el método **reverse**.

```
let  a = ["a", "b", "c", "d", "e", "f"];
a.reverse(); // Hace el reverse del array original
```

```
console.log(a); // Imprime ["f", "e", "d", "c", "b", "a"]
```

También, podemos ordenar los elementos de un array usando el método **sort**.

```
let  a = ["Peter", "Anne", "Thomas", "Jen", "Rob", "Alison"];
a.sort(); // Ordena el array original
console.log(a); // Imprime ["Alison", "Anne", "Jen", "Peter", "Rob", "Thomas"]
```

Pero, ¿Qué ocurre si intentamos ordenar elementos que no son string?. Por defecto, lo ordenará por su valor como string (teniendo en cuenta que si son objetos, se intentará llamar al método **toString()** para ordenarlo). Para ello, tendremos que pasar una función (de ordenación), que comparará 2 valores del array y devolverá un valor numérico indicando cual es menor (negativo si el primero es menor, 0 si son iguales y positivo si el primero es mayor).

```
let a = [20, 6, 100, 51, 28, 9];
a.sort(); // Ordena el array original
console.log(a); // Imprime [100, 20, 28, 51, 6, 9]
a.sort((n1, n2) => n1 - n2);
console.log(a); // Imprime [6, 9, 20, 28, 51, 100]
```

Veamos un ejemplo con objetos (los veremos en el bloque 3), en este caso son personas y las ordenaremos según la edad.

```
function Persona(nombre, edad) { // Constructor de la clase persona
   this.nombre = name;
   this.edad = edad;

   this.toString = function() { // Método toString()
      return this.nombre + " (" + this.edad + ")";
   }
}

let  personas = [];
personas[0] = new Persona("Thomas", 24);
personas[1] = new Persona("Mary", 15);
personas[2] = new Persona("John", 51);
personas[3] = new Persona("Philippa", 9);

personas.sort((p1, p2) => p1.age - p2.age));
console.log(personas.toString()); // Imprime: "Philippa (9),Mary (15),Thomas (24),John (51)"
```

Usando **indexOf**, podemos conocer si el valor que le pasamos se encuentra en el array o no. Si lo encuentra nos devuelve la primera posición donde está, y si no, nos devuelve -1. Usando el método **lastIndexOf** nos devuelve la primera ocurrencia encontrada empezando desde el final.

```
let a = [3, 21, 15, 61, 9, 15];
console.log(a.indexOf(15)); // Imprime 2
console.log(a.indexOf(56)); // Imprime -1. No encontrado
console.log(a.lastIndexOf(15)); // Imprime 5
```

El método **every** devolverá un boolean indicando si todos los elementos del array cumplen cierta condición. Esta función recibirá cualquier elemento, lo testeará, y devolverá cierto o falso dependiendo de si cumple la condición o no.

```
let a = [3, 21, 15, 61, 9, 54];
```

```
console.log(a.every(num =>  num < 100));  // Comprueba si cada número es menor a 100. Imprime true
console.log(a.every(num => num % 2 == 0)); // Comprueba si cada número es par. Imprime false
```

Por otro lado, el método **some** es similar a **every**, pero devuelve cierto en el momento en el que **uno de los elementos** del array cumple la condición.

```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.some(num => num % 2 == 0));  // Comprueba si algún elemento del array es par. Imprime true
```

Podemos iterar por los elementos de un array usando el método **forEach.** De forma opcional, podemos llevar un seguimiento del índice al que está accediendo en cada momento, e incluso recibir el array como tercer parámetro.

```
let a = [3, 21, 15, 61, 9, 54];
let sum = 0;
a.forEach(num => sum += num);
console.log(sum); // Imprime 163

a.forEach((num, indice, array) => { // índice y array son parámetros opcionales
   console.log("Índice " + indice + " en [" + array + "] es " + num);
}); // Imprime -> Índice 0 en [3,21,15,61,9,54] es 3, Índice 1 en [3,21,15,61,9,54] es 21, ...
```

Para modificar todos los elementos de un array, el método **map** recibe una función que transforma cada elemento y lo devuelve. Este método devolverá al final un nuevo array del mismo tamaño conteniendo todos los elementos transformados.

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.map(num => num*2)); // Imprime [8, 42, 66, 24, 18, 108]
```

Para filtrar los elementos de un array, y obtener como resultado un array que contenga sólo los elementos que cumplan cierta condición, usamos el método **filter**.

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.filter(num => num % 2 == 0));  // Imprime [4, 12, 54]
```

El método **reduce** usa una función que acumula un valor, procesando cada elemento (segundo parámetro) con el valor acumulado (primer parámetro). Como segundo parámetro de reduce, deberías pasar un valor inicial. Si no pasas un valor inicial, el primer elemento de un array será usado como tal (si el array está vacío devolvería undefined).

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.reduce((total, num) => total + num, 0));  // Suma todos los elementos del array. Imprime 133
console.log(a.reduce((max, num) => num > max? num : max, 0));  // Número máximo del array. Imprime 54
```

Para hacer lo mismo que **reduce** hace pero al revés, usaremos **reduceRight**.

```
let  a = [4, 21, 33, 12, 9, 154];
// Comienza con el último número y resta todos los otros números
console.log(a.reduceRight((total, num)  => total - num));
// Imprime 75 (Si no queremos enviarle un valor inicial, empezará con el valor de la última posición del array
```

Extensiones de Array en ES2015

Estos son los nuevos métodos que tiene el objeto global Array:

- **Array.of(value)** → Si queremos instanciar un array con un sólo valor, y éste es un número, con new Array() no podemos hacerlo, ya que crea vacío con ese número de posiciones.

```
let array = new Array(10); // Array vacío (longitud 10)
let array = Array(10); // Mismo que arriba: array vacío ( longitud 10)
let array = Array.of(10); // Array con longitud 1 -> [10]
let array = [10]; // // Array con longitud 1 -> [10]
```

- **Array.from(array, func)** → Funciona de forma similar al método **map**, crea un array desde otro array. Se aplica una operación de transformación (función lambda o anónima) para cada ítem.

```
let array = [4, 5, 12, 21, 33];
let array2 = Array.from(array, n => n * 2);
console.log(array2); // [8, 10, 24, 42, 66]
let array3 = array.map(n => n * 2); // Igual que Array.from
console.log(array3); // [8, 10, 24, 42, 66]
```

- **Array.fill(value)** → Este método sobrescribe todas las posiciones de un array con un nuevo valor. Es una buena opción para inicializar un array que se ha creado con N posiciones.

```
let sums = new Array(6); // Array con 6 posiciones
sums.fill(0); // Todas las posiciones se inicializan a 0
console.log(sums); // [0, 0, 0, 0, 0, 0]

let numbers = [2, 4, 6, 9];
numbers.fill(10); // Inicializamos las posiciones  al valor 10
console.log(numbers); // [10, 10, 10, 10]
```

- **Array.fill(value, start, end)** → Este método hace lo mismo que antes pero rellenando el array desde una posición inicial (incluida) hasta una final (excluida). Si no se especifica la última posición, se rellenará hasta el final.

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.fill(10, 2, 5); // Las posiciones 2,3,4 se ponen a 10
console.log(numbers); // [2, 4, 10, 10, 10, 16]

let numbers2 = [2, 4, 6, 9, 14, 16];
numbers2.fill(10, -2); // Las dos últimas posiciones se ponen a 10
console.log(numbers2); // [2, 4, 6, 9, 10, 10]
```

- **Array.find(condition)** → Encuentra y devuelve el primer valor que encuentre que cumple la condición que se establece. Con **findIndex**, devolvemos la posición que ocupa ese valor en el array.

```
let numbers = [2, 4, 6, 9, 14, 16];
console.log(numbers.find(num => num >= 10)); // Imprime 14 (primer valor encontrado >= 10)
console.log(numbers.findIndex(num => num >= 10)); // Imprime 4 (numbers[4] -> 14)
```

- **Array.copyWithin(target, startwith)** → Copia los valores del array empezando desde la posición **startWith**, hasta la posición **target** en el resto de posiciones del array (en orden). Por ejemplo:

```
let numbers = [2, 4, 6, 9, 14, 16];
```

```
numbers.copyWithin(3, 0); // [0] -> [3], [1] -> [4], [2] -> [5]
console.log(numbers); // [2, 4, 6, 2, 4, 6]
```

Rest y spread

Rest es la acción de transformar un grupo de parámetros en un array, y **spread** es justo lo opuesto, extraer los elementos de un array (o de un string) a variables.

Para usar **rest** en los parámetros de una función, se declara siempre como último parámetro (**1 máximo**) y se le ponen tres puntos'...' delante del mismo. Este parámetro se transformará automáticamente en un array conteniendo todos los parámetros restantes que se le pasan a la función. Si por ejemplo, el parámetro ***rest*** está en la tercera posición, contendrá todos los parámetros que se le pasen a excepción del primero y del segundo (a partir del tercero).

```
function getMedia(...notas) {
   console.log(notas); // Imprime [5, 7, 8.5, 6.75, 9] (está en un array)
   let total = notas.reduce((total,notas) => total + notas, 0);
   return total / notas.length;
}
console.log(getMedia(5, 7, 8.5, 6.75, 9)); // Imprime 7.25

function imprimirUsuario(nombre, ...lenguajes) {
   console.log(nombre + " sabe " + lenguajes.length + " lenguajes: " + lenguajes.join(" - "));
}

// Imprime "Pedro sabe 3 lenguajes: Java - C# - Python"
printUserInfo("Pedro", "Java", "C#", "Python");
// Imprime "María sabe 5 lenguajes: JavaScript - Angular - PHP - HTML - CSS"
printUserInfo("María", "JavaScript", "Angular", "PHP", "HTML", "CSS");
```

Spread es lo "opuesto" de **rest**. Si tenemos una variable que contiene un array, y ponemos los tres puntos '...' delante de este, extraerá todos sus valores. Podemos usar la propiedad por ejemplo con el método **Math.max,** el cual recibe un número indeterminado de parámetros y devuelve el mayor de todos.

```
let nums = [12, 32, 6, 8, 23];
console.log(Math.max(nums)); // Imprime NaN (array no es válido)
console.log(Math.max(...nums)); // Imprime 32 -> equivalente a Math.max(12, 32, 6 ,8 ,23)
```

Podemos usar también esta propiedad si necesitamos clonar un array.

```
let a = [1, 2, 3, 4];
let b = a; // Referencia el mismo array que 'a' (las modificaciones afectan a ambos).
let c = [...a]; // Nuevo array -> contiene [1, 2, 3, 4]
```

Desestructuración de arrays

Desestructuración es la acción de extraer elementos individuales de un array (o propiedades de un objeto) directamente en variables individuales. Podemos también *desestructurar* un string en caracteres.

Vamos a ver un ejemplo donde asignamos los tres primeros elementos de un array, en tres variables diferentes, usando una única asignación.

```
let array = [150, 400, 780, 1500, 200];
let [first, second, third] = array; // Asigna los tres primeros elementos del array
console.log(third); // Imprime 780
```

¿Qué pasa si queremos saltarnos algún valor? Se deja vacío (sin nombre) dentro de los corchetes y no será asignado:

```
let array = [150, 400, 780, 1500, 200];
let [first, , third] = array; // Asigna el primer y tercer elemento
console.log(third); // Imprime 780
```

Podemos asignar el resto del array a la última variable que pongamos entre corchetes usando **rest** (como en el punto anterior del tema):

```
let array = [150, 400, 780, 1500, 200];
let [first, second, ...rest] = array; // rest -> array
console.log(rest); // Imprime [780, 1500 ,200]
```

Si queremos asignar más valores de los que contiene el array y no queremos obtener undefined, podemos usar valores por defecto:

```
let array = ["Peter", "John"];
let [first, second = "Mary", third = "Ann"] = array; // rest -> array
console.log(second); // Imprime "John"
console.log(third); // Imprime "Ann" -> valor por defecto
```

También podemos desestructurar **arrays anidados**:

```
let sueldos = [["Pedro", "Maria"], [24000, 35400]];
let [[nombre1, nombre2], [sueldo1, sueldo2]] = sueldos;
console.log(nombre1 + " gana " + sueldo1 + "€"); // Imprime "Pedro gana 24000€"
```

También se puede desestructurar un array enviado como parámetro a una función en valores individuales:

```
function imprimirUsuario([id, nombre, email], otraInfo = "Nada") {
   console.log("ID: " + id);
   console.log("Nombre: " + nombre);
   console.log("Email: " + email );
   console.log("Otra info: " + otraInfo );
}

let infoUsu = [3, "Pedro", "peter@gmail.com"];
imprimirUsuario(infoUsu, "No es muy listo");
```

Map

Un Map es una colección que guarda parejas de [clave,valor], los valores son accedidos usando la correspondiente clave. En JavaScript, un objeto puede ser considerado como un tipo de Mapa pero con algunas limitaciones (Sólo con strings y enteros como claves).

```
let obj = {
   0: "Hello",
   1: "World",
   prop1: "This is",
```

```
    prop2: "an object"
}

console.log(obj[1]); // Imprime "World"
console.log(obj["prop1"]); // Imprime "This is"
console.log(obj.prop2); // Imprime "an object"
```

La nueva colección **Map** permite usar cualquier objeto como clave. Creamos la colección usando el constructor **new Map()**, y podemos usar los métodos **set**, **get** y **delete** para almacenar, obtener o eliminar un valor basado en una clave.

```
let person1 = {name: "Peter", age: 21};
let person2 = {name: "Mary", age: 34};
let person3 = {name: "Louise", age: 17};

let hobbies = new Map(); // Almacenará una persona con un array de hobbies (string)
hobbies.set(person1, ["Tennis", "Computers", "Movies"]);
console.log(hobbies); // Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers", "Movies"]}

hobbies.set(person2, ["Music", "Walking"]);
hobbies.set(person3, ["Boxing", "Eating", "Sleeping"]);
console.log(hobbies);
```

```
Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers", "Movies"], Object
▼{name: "Mary", age: 34} => ["Music", "Walking"], Object {name: "Louise", age: 17}
=> ["Boxing", "Eating", "Sleeping"]} i
  size: (...)
  ▶ __proto__: Map
  ▼ [[Entries]]: Array[3]
    ▼ 0: {Object => Array[3]}
      ▼ key: Object
          age: 21
          name: "Peter"
        ▶ __proto__: Object
      ▼ value: Array[3]
          0: "Tennis"
          1: "Computers"
          2: "Movies"
          length: 3
```

Cuando usamos un objeto como clave, debemos saber que almacenamos una **referencia a ese objeto** (Luego veremos WeakMap). Por tanto, se debe **usar la misma referencia** para acceder a un valor que para eliminarlo en ese mapa.

```
console.log(hobbies.has(person1)); // true (referencia al objeto original almacenado)
console.log(hobbies.has({name: "Peter", age: 21})); // false (mismas propiedades pero objeto diferente!)
```

La propiedad **size** devuelve la longitud del mapa y podemos iterar a través por él usando [Symbol.iterator] o el bucle **for..of**. Para cada iteración, se devuelve un array con dos posiciones **0 → key** y **1 → value**.

```
console.log(hobbies.size); // Imprime 3
hobbies.delete(person2); // Elimina person2 del Map
console.log(hobbies.size); // Imprime 2
console.log(hobbies.get(person3)[2]); // Imprime "Sleeping"

/** Imprime todo:
 * Peter: Tennis, Computers, Movies
```

```
 * Louise: Boxing, Eating, Sleeping */
for(let entry of hobbies) {
   console.log(entry[0].name + ": " + entry[1].join(", "));
}
for(let [person, hobArray] of hobbies) { // Mejor
   console.log(person.name + ": " + hobArray.join(", "));
}

hobbies.forEach((hobArray, person) => { // Mejor
   console.log(person.name + ": " + hobArray.join(", "));
});
```

Si tenemos un tenemos un array que contiene otros arrays con dos posiciones (key, value), podemos crear un mapa directamente a partir del mismo.

```
let prods = [
   ["Computer", 345],
   ["Television", 299],
   ["Table", 65]
];

let prodMap = new Map(prods);
console.log(prodMap); // Map {"Computer" => 345, "Television" => 299, "Table" => 65}
```

Set

Set es como **Map**, pero no almacena los valores (sólo la clave). Puede ser visto como una colección que **no permite valores duplicados** (en un array puede haber valores duplicados). Se usa, **add**, **delete** y **has** → son métodos que devuelven un booleano para almacenar, borrar y ver si existe un valor.

```
let set = new Set();
   set.add("John");
   set.add("Mary");
   set.add("Peter");
   set.delete("Peter");
   console.log(set.size); // Imprime 2

   set.add("Mary"); // Mary ya existe
   console.log(set.size); // Imprime 2

   // Itera a través de los valores
   set.forEach(value => {
      console.log(value);
   })
```

Podemos crear un Set desde un array (lo cual elimina los valores duplicados).

```
   let names = ["Jennifer", "Alex", "Tony", "Johny", "Alex", "Tony", "Alex"];
   let nameSet = new Set(names);
   console.log(nameSet); // Set {"Jennifer", "Alex", "Tony", "Johny"}
```