JavaScript

Ángel Villalba Fernández-Paniagua

1-15

Contenidos

Chapter 1. JavaScript

JavaScript fue creado en 10 días en el año 1995 por *Brendan Eich*, que trabajaba en Netscape. Al principio, no se llamaba JavaScript, sino que su nombre era **Mocha**, un nombre que le eligió *Marc Andreessen* (fundador de Netscape). En septiembre de ese mismo año, se le cambia el nombre por primera vez a **LiveScript**. Un poco más tarde en ese mismo año, Nescape y Sun Microsystems firman una alianza para desarrollar el nuevo lenguaje de programación, ahora con el nombre de JavaScript, el cual se lo dan como una estrategia de marketing, ya que en aquella época **Java** estaba de moda en el mundo informático y añadiendole Java al nombre, lo que intentaban era atraer a programadores de Java para que probaran este lenguaje.

La primera versión de JavaScript es un exito y *Microsoft* decide sacar **JScript** con su navegador Internet Explorer 3. JScript era una copia de JavaScript a la que le habían cambiado el nombre para evitar problemas legales. Y viendo esto Netscape decidió estandarizar el lenguaje JavaScript.

En los años 1996 y 1997 JavaScript es llevado a **ECMA** para labrarse una especificación estándar, que otros proveedores de navegadores prodrían implementar basandose en el trabajo realizado en Netscape. Este trabajo lleva a la liberación oficial de **ECMAScript** que es el nombre de la norma oficial, siendo JavaScript la implementación más conocida.

El proceso de las normas continuó con los lanzamientos de ECMAScript 2 (en 1998) que traía unas pequeñas modificaciones para adaptar el lenguaje a una ISO (Organización Internacional para la Estandarización) y ECMAScript 3 (en 1999) que es la línea base para el JavaScript moderno.

En 2005, las comunidades se pusieron a trabajar para revolucionar lo que podría hacerse con JavaScript. Este esfuerzo de las comunidades se desató cuando en 2005, *Jesse James Garrett* publicó un libro en el que se usaba el término **AJAX** y describió un conjunto de tecnologías, de las cuales JavaScript era la columna vertebral, que se usaba para crear aplicaciones web en las que los datos pueden ser cargados desde el servidor, evitando la necesidad de cargar la página completa y como resultado: aplicaciones más dinámicas. Esto dio lugar a un periodo de renacimiento del uso de JavaScript encabezado por librerías de código abierto y las comunidades que se formaron a su alrededor, con algunas librerías como **JQuery**, **Prototype**…

En julio de 2008 se decide cambiar el nombre de ECMAScript 3.1 a ECMAScript 5 e impulsar el lenguaje.

Todo esto nos trae a la actualidad, con JavaScript que ha entrado en un nuevo y emocionante ciclo de evolución, innovación y normalización, con nuevos desarrollos como la plataforma **nodejs**, que nos permite usar JavaScript en el lado del servidor, y las **APIs de HTML5** que nos ofrecen funcionalidades com acceder a la ubicación del dispositivo, usar sockets…

Y en el año 2015 sale la **ECMAScript 6** que trae muchos cambios significativos al lenguaje.

Chapter 2. Uso de JavaScript

Básicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario hacer nada con estos programas, ni siquiera hay que compilarlos. Las aplicaciones escritas en JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

Entonces, JavaScript es uno de los lenguajes más usados en el mundo, es el único lenguaje de programación que entienden los navegadores, y se puede usar para hacer las páginas web dinámicas, para desarrollar videojuegos, aplicaciones móviles…

Podemos incluir scripts en nuestras páginas web de alguna de las siguientes formas:

- Escribiendo el código JavaScript en archivos externos a nuestras páginas HTML:
  - El código JavaScript se puede escribir en un archivo externo de tipo JavaScript (con la extensión `.js`) que los documentos HTML enlazan con la etiqueta `<script src='archivo.js'>`.
  - Se pueden enlazar todos los archivos JavaScript que se necesiten y cada documento HTML puede enlazar tantos archivos JavaScript como vaya a utilizar.
  - La mayor ventaja de esta forma de usar el código JavaScript, es la reutilización.
- Incluyendo scripts en las páginas HTML:
  - El código JavaScript se encierra entre las etiquetas `<script>···</script>` dentro del archivo HTML, y estas etiquetas se pueden incluir en cualquier parte del documento HTML, aunque se recomienda hacerlo dentro de la cabecera del documento, es decir, entre las etiquetas `<head>···</head>`.
  - Este método se usa cuando vamos a definir un bloque pequeño de código JavaScript, o cuando se quieren incluir códigos específicos en un determinado documento.
  - La principal desventaja es que no se puede reutilizar el código.
- Incluyendo JavaScript dentro de las propias etiquetas HTML:
  - Esta última forma es la menos usada, ya que consiste en crear bloques de código JavaScript dentro de las etiquetas HTML del documento.
  - La mayor desventaja de esta forma es que ensucia innecesariamente el código HTML del documento y hace más dificil el mantenimiento del bloque de código.

2.1. Lab: JavaScript en archivos externos

En este laboratorio vamos a ver como importar código JavaScript que se encuentra en un archivo `.js` dentro de una página HTML.

Empezamos por crear un archivo de javascript donde pondremos nuestro código de JavaScript.

/javascript-uso-de-js-en-archivo-externo-lab/main.js

```
console.log('Código JS en un archivo externo');
```

Una vez que tenemos nuestro archivo, vamos a importarlo en un archivo de HTML usando la etiqueta `script` y pasandole la ruta al archivo con el código de JavaScript al atributo `src` de esta etiqueta.

/javascript-uso-de-js-en-archivo-externo-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <script src="main.js"></script>
</head>
<body>
</body>
</html>
```

2.2. Lab: JavaScript en la página HTML

En este laboratorio vamos a ver como añadiro código de JavaScript en el propio documento de HTML.

Creamos el archivo de HTML, y entre las etiquetas `script` pondremos el código que queremos ejecutar. Una vez que se cargue la página en el navegador y lea dicho código, lo ejecutará.

/javascript-uso-de-js-en-cabecera-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <script>
    console.log('Código JS en el HTML');
  </script>
</head>
<body>
</body>
</html>
```

2.3. Lab: JavaScript en las etiquetas HTML

En este laboratorio vamos a ver como poner código de JavaScript en las propias etiquetas HTML.

Creamos el archivo HTML donde vamos a poner el JavaScript, y dentro de una etiqueta, vamos a añadir un evento `onclick` al que le vamos a igualar el código de JavaScript que queremos que se ejecute cuando se pulse sobre la etiqueta.

/javascript-uso-de-js-en-etiquetas-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
</head>
<body>
  <h1 onclick="console.log('Código JS en la etiqueta h1')">Título de la página</h1>
</body>
</html>
```

Chapter 3. Sintaxis

La sintaxis de JavaScript son aquellas reglas que hay que seguir al escribir el código y que este funcione correctamente. La sintaxis de JavaScript es muy parecia a la de otros lenguajes como Java. Las reglas que hay que tener en cuenta son las siguientes:

- Tipo de las variables: en la mayor parte de los lenguajes a las variables se les indica que tipo de valor van a almacenar, pero en JS no hay que hacerlo, y de esta forma, podemos hacer que una variable contenga distintos tipos de datos a lo largo de la ejecución del script.
- `;` al final de cada instrucción: en la mayor parte de lenguajes, cada instrucción acaba en `;`. En JS no es necesario ponerlo `var j = 0`, aunque se recomienda hacerlo `var j = 0;`.
- No se tienen en cuenta los espacios y los saltos de línea: da igual si añadimos muchos espacios, o algunos saltos de línea para dejar el código más legible, porque JS ignora aquellos espacios que sobran.
- Es *case sensitive*: en JS una variable llamada *coche* es totalmente distinta a una variable llamada *Coche*, y por lo tanto si no usamos la correcta, el script no funcionará como se espera. No es lo mismo `new Array()` que `new array()`, donde el primero crea un array, mientras que el segundo no.
- Se pueden incluir comentarios: podemos añadir comentarios en nuestro código para aclarar lo que hacen las instrucciones. Podemos incluir comentarios de una sola línea `// ⋯` o de múltiples líneas `/* ⋯ */`.

Chapter 4. Variables

Las variables nos permiten almacenar datos. Y estos datos pueden cambiar su valor mientras se ejecuta el script.

Las variables en JS se declaran usando la palabra reservada `var` seguida del nombre que le vamos a asignar a la variable. Este nombre nos va a permitir usar la variable en otros sitios del script.

```
var numero;
```

En caso de que el nombre de la variable esté compuesto por más de una palabra, esta se suele escribir en **camelcase**, es decir la primera palabra en minúsculas, y el resto de palabras que van a continuación con la primera letra en mayúscula.

```
var unNumero;
```

Las variables no saben de que tipo son hasta que se les asigna un valor, hasta entonces son de tipo **undefined**. A las variables se les asignan valores de la siguiente forma.

```
var unaVariable = 'Hola mundo!';

unaVariable = 10;
```

Se pueden declarar e inicializar varias variables en una misma línea de código de la siguiente forma:

```
var alto, ancho, area;
alto = 10;
ancho = 5;
area = alto * ancho;

var precio = 1.5, cantidad = 5;
var total = cantidad * precio;
```

Los nombres de las variables deben cumplir las siguientes reglas:

- El nombre de la variable solo puede estar formado por letras, números y los simbolos `$` y `_`.
- El primer carácter del nombre de la variable no puede ser un número, ni un carácter de los que no están permitidos.
- El nombre no puede ser una palabra reservada del lenguaje: for, if, while, var, throw...

4.1. Lab: ¿Qué variables están declaradas correctamente?

En este laboratorio, tenemos que indicar cuales de las variables que se muestran a continuación se han declarado de forma correcta.

```
var una_variable;
var una-variable;
var 1_variable;
var $1variable;
var una;variable;
```

- La primera variable está **bien** declarada.
- La segunda variable está **mal** declarada porque los nombre de las variables no pueden tener un `-`.
- La tercera variable está **mal** declarada porque el nombre de las variables no pueden empezar con un número.
- La cuarta variable está **bien** declarada porque los únicos símbolos por los que pueden empezar los nombres de las variables son `$` y `_` y en este caso empieza por el primero.
- La quinta variable está declarando `una` como variable, y está intentando acceder a la variable con el nombre `variable` que en este caso nos dará un error indicandonos que dicha variable no está definida.

Chapter 5. Tipos de datos

JavaScript distingue entre los siguientes tipos de valores: strings, números, booleanos, nulos, indefinidos, arrays y objetos.

5.1. String

Los datos de tipo **string** representan cadenas de texto. Estas cadenas tienen que ir entre comillas (simples o dobles) y en una línea.

```
var saludo = 'Hola mundo!';
```

En caso de necesitar mostrar las comillas en el string, hay que usar las comillas que no hemos usado para inicializar la variable o *escaparlas* usando las **backslash** (que le indican al interprete que el siguiente caracter es parte del string).

```
var unTexto = "Esto es un 'texto'";
var otroTexto = 'Esto es \'otro\' texto';
```

5.2. Number

Los datos de tipo **number** representan valores númericos, tanto valores enteros como valores con decimales.

```
var altura = 180;
```

5.3. Boolean

Los datos de tipo **boolean** solo pueden tener uno de los siguientes dos valores, **true** (verdadero) o **false** (falso).

```
var estaEncendido = false;
```

5.3.1. Valores Truthy y Falsy

Debido a la **conversión de tipos**, cada valor en JavaScript puede ser tratado como si fuera un *Boolean*. **Falsy** son los valores que son tratados como si fueran `false`, mientras que **Truthy** son aquellos que son tratados como si fueran `true`.

Table 1. Valores Falsy

Valor
El booleano *false*
El número *0*
Un string vacío '' o con espacios en blanco
El valor indefinido *undefined*
El valor nulo *null*

Table 2. Valores Truthy

Valor
Cualquier otro valor que no sea *Falsy*

5.4. Null

Null es un valor primitivo que representa la ausencia de valor. Cuando se usa en una expresión booleana, actua como *false* y de esta forma podemos comprobar que una variable tiene valor o no.

```
var nula = null;

if (!nula) {
  console.log('Nula es una variable ' + nula)
}
```

5.5. Undefined

Undefined es un valor primitivo que representa que un valor es desconocido, por ejemplo cuando no se le ha asignado un valor a una variable. También actua como *false* cuando se usa en una expresión booleana.

```
var indefinida;

if (!indefinida) {
  console.log('Indefinida es una variable ' + indefinida)
}
```

5.6. Array

Un **array** es un tipo especial de variable que no almacena un valor, sino que almacena una lista de valores. Se suele usar para trabajar con listas o con un conjunto de valores que están relacionados entre ellos. Al crear el array no es necesario indicar cual es el tamaño que va a tener la lista.

Los arrays se pueden crear de las siguientes dos formas:

```
var dias = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo'];

var colores = new Array('Blanco', 'Negro', 'Rojo', 'Azul', 'Verde');
```

Se recomienda usar la primera forma (**array literal**) a la hora de crear arrays, a no ser que queramos crear un array vacío con una longitud muy alta, en el que vendría mejor usar la segunda forma (**array constructor**):

```
var arrayVacioDeCienElementos = new Array(100); // En lugar de [,,,,,,,,,,,...,]
```

Los arrays guardan los elementos empezando por la posición 0. Y para poder acceder a los elementos tenemos que poner entre corchetes la posición en la que se encuentra el elemento.

```
var dias = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo'];

var martes = dias[1];
var domingo = dias[6];
```

Para modificar un elemento, solo hay que indicar el elemento que queremos (indicando la posición entre corchetes) e igualarle el nuevo valor que se le va a dar.

```
var colores = new Array('Blanco', 'Negro', 'Rojo', 'Azul', 'Verde');

colores[4] = 'Amarillo';
colores[5] = 'Naranja';
```

Chapter 6. Operadores

Los operadores realizan cambios sobre los datos y los podemos clasificar en los siguientes grupos:

Table 3. Operadores aritméticos

Operador	Ejemplo
+ (Suma)	5 + 10 = 15
- (Resta)	8 - 3 = 5
* (Multiplicación)	2 * 4 = 8
/ (División)	9 / 4 = 2.25
% (Módulo)	10 % 3 = 1
++ (Incremento)	a=1; a++; ⇒ a=2;
— (Decremento)	a=5; a--; ⇒ a=4;

Table 4. Operadores de strings

Operador	Ejemplo
+ (Concatenación)	'5' + '10' = '510'
+ (Concatenación)	221 + 'B Baker Street' = '221B Baker Street'

Table 5. Operadores relacionales

Operador	Ejemplo
< (Menor que)	3 < 7 (true)
> (Mayor que)	8 > 9 (false)
<= (Menor o igual que)	2 <= 2 (true)
>= (Mayor o igual que)	9 >= 4 (true)
== (Igual a)	3 == 2 (false)
!= (Distinto de)	4 != 5 (true)

Table 6. Operadores lógicos

Operador	Ejemplo
&& (AND)	10 > 3 && false (false)
|| (OR)	10 > 3 || false (true)
! (NOT)	!(10 > 3) (false)

Table 7. Operadores de asignación (a = 5, b = 4)

Operador	Ejemplo
= (Igual)	a = b (4)
+= (Suma)	a += b => a = a + b (9)

Operador	Ejemplo
-= (Resta)	a -= b => a = a - b (1)
*= (Multiplicación)	a *= b => a = a * b (20)
/= (División)	a /= b => a = a / b (1.25)

Operador de tipo (typeof)

Este operador nos dice el tipo de una variable o valor.

```
// typeof [variable o valor]
var a = 6;
typeof a; // => number
```

Chapter 7. Consola y Popups en JavaScript

JavaScript nos permite mostrar mensajes en pantalla a través de la *consola* o de un *popup*. Además con los popups podemos hacer que el usuario interaccione con el programa.

7.1. Consola

Los navegadores disponen de una consola sobre la que escribir mensajes que aparecerán en el navegador. Esta es la forma más fácil de mostrar mensajes, y podemos mostrar mensajes de distintos tipos:

- `console.log('mensaje')`: muestra mensajes generales
- `console.warn('mensaje')`: muestra mensajes de alerta
- `console.error('mensaje')`: muestra mensajes de error
- `console.dir('mensaje')`: muestra las propiedades de un objeto

7.2. Popups

JavaScript también permite mostrar popups en el navegador para mostrar mensajes al usuario o para permitir a este interactuar con el programa. Los distintos tipos de popups que podemos mostrar son:

- `alert('mensaje')`: muestra un popup con un mensaje de alerta. Este popup tiene un botón de *aceptar* y devuelve un valor `undefined`.
- `confirm('mensaje')`: muestra un popup con un mensaje (suele ser una pregunta). Este popup tiene dos botones, uno es *cancelar* que en caso de pulsarlo devuelve `false` y el otro es *aceptar* y en este caso devuelve `true`.
- `prompt('mensaje')`: muestra un popup con un mensaje (se suele pedir que se introduzca una respuesta) y un campo de texto. Este popup tiene dos botones, uno es *cancelar* que en caso de pulsarlo devuelve `null` y el otro es *aceptar* y en este caso devuelve el texto que se haya introducido en el campo.

7.3. Lab: mostrar respuestas a preguntas

En este laboratorio vamos a ver como realizar preguntas al usuario a través de los modales que vienen de forma nativa con los navegadores, y mostrar las respuestas por la consola del navegador.

7.3.1. Realizar preguntas

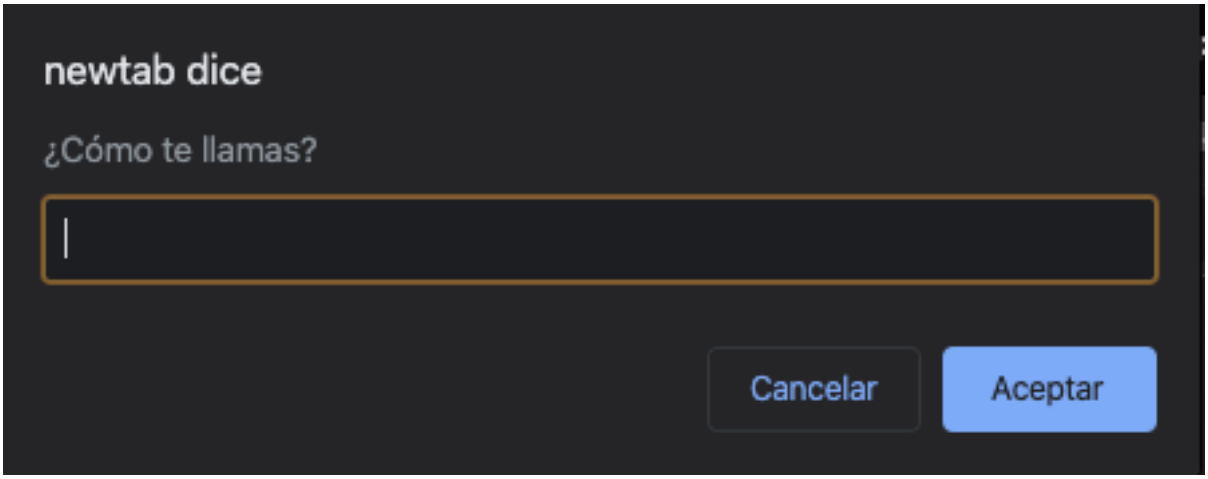
Vamos a empezar por realizar algunas preguntas al usuario, y para ello usaremos los distintos métodos que nos proporciona JavaScript para mostrar los popups del navegador.

/javascript-consola-y-popups-lab/app.js

```
const nombre = prompt('¿Cómo te llamas?')
```

```
const apellidos = prompt('¿Y tus apellidos?')
const tieneTrabajo = confirm('¿Tienes trabajo?')
```

Si probamos a ejecutar este código en la consola del navegador, tienen que ir apareciendo los popups permitiendonos contestar las preguntas con nuestros datos.



7.3.2. Mostrar respuestas

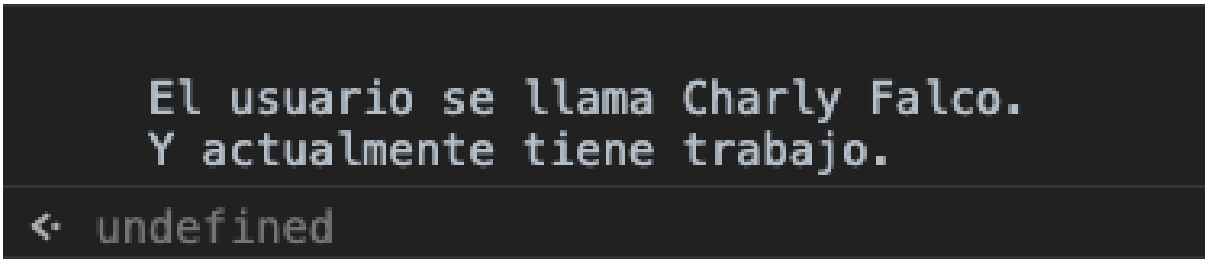
Una vez que tenemos la parte de las preguntas, vamos a mostrar las respuestas del usuario por consola.

/javascript-consola-y-popups-lab/app.js

```
const nombre = prompt('¿Cómo te llamas?')
const apellidos = prompt('¿Y tus apellidos?')
const tieneTrabajo = confirm('¿Tienes trabajo?')

console.log(`
  El usuario se llama ${nombre} ${apellidos}.
  Y actualmente ${tieneTrabajo ? '' : 'no '}tiene trabajo.
`)
```

Y si volvemos a ejecutar el código, deberíamos de ver el resultado en la misma consola del navegador.



Chapter 8. Condicionales

Una vez que ya tenemos el valor que nos devuelven estos operadores, vamos a usar las instrucciones condicionales (`if`, `if ⋯ else` ...) para elegir el código que debe de ejecutarse.

8.1. IF

La instrucción condicional `if` comprueba una condición, y si esta condición resulta ser `true`, entonces se ejecutará el bloque de código perteneciente al `if`. En caso de ser `false`, el bloque de código del `if` se salta y la ejecución sigue a partir de donde termina ese bloque.

```
if (nota >= 5) {
  console.log('Has aprobado');
}
```

8.2. IF - ELSE

La instrucción condicional `if ⋯ else` también va a comprobar una condición. En este caso, si la condición resulta ser `true` se va a ejecutar el bloque de código del `if`, mientras que si la condición es `false`, se ejecutará el bloque del `else`.

```
if (nota >= 5) {
  console.log('Has aprobado');
} else {
  console.log('Has suspendido');
}
```

8.3. IF - ELSE IF

Si necesitamos tener en nuestro código más de un camino, podemos añadir más instrucciones `if` justo después de la instrucción del `else`. De esta forma, si la primera condición no se cumple, comprobará la del segundo `if` y así sucesivamente hasta el último `else`.

```
if (nota < 5) {
  console.log('Has suspendido');
} else if (nota < 6) {
  console.log('Suficiente');
} else if (nota < 8) {
  console.log('Bien');
} else if (nota < 10) {
  console.log('Notable');
} else {
  console.log('Sobresaliente');
}
```

8.4. SWITCH

La instrucción `switch` va a comparar el valor que recibe, con cada uno de los valores que hay en los casos (instrucción `case`), y va a ejecutar el bloque de código correspondiente al caso cuyo valor coincida con el valor que se le pasa al `switch`. El `switch` tiene un caso especial (`default`) cuyo bloque de código se va a ejecutar cuando ninguno de los casos coincida con el valor que se está comparando. Al final de cada `case`, se pone la palabra instrucción `break` que le dice al interprete de JavaScript que ya ha terminado de ejecutar el `switch` y que puede seguir ejecutando el código a partir de su bloque.

```
switch (nota) {
  case 5:
    console.log('Suficiente');
    break;
  case 6:
    console.log('Bien');
    break;
  case 7:
    console.log('Bien alto');
    break;
  case 8:
    console.log('Notable');
    break;
  case 9:
      console.log('Notable alto');
      break;
  case 10:
    console.log('Sobresaliente');
    break;
  default:
    console.log('Suspenso');
    break;
}
```

8.5. Operador ternario ?

El operador ternario (**?**) actua como si fuera un `if - else`, solo que en este caso se usa como parte de una expresión en la que una instrucción `if - else` no sería muy práctica.

```
// [condicion] ? [código si la condición es true] : [código si la condición es false]
var resultado = (5 < 3) ? 'Es menor' : 'Es mayor'; // resultado = 'Es mayor'
```

8.6. Lab: Fizz-Buzz

En este laboratorio vamos a crear el juego de Fizz-Buzz. Las reglas del juego son las siguientes:

- Consiste en preguntar un número al usuario un número
- Si el número es múltiplo de 3 se muestra el mensaje *Fizz* junto al número
- Si el número es múltiplo de 5 se muestra el mensaje *Buzz* junto al número

Empezamos por crear nuestro archivo `fizz-buzz.js` en el que vamos a pedir al usuario que introduzca un número.

/javascript-condicionales-lab/fizz-buzz.js

```
let n = prompt('Introduce un número...')
```

Este número que introduce el usuario y hemos guardado en la variable `n` llegará como un **String**, pero nosotros lo vamos a comparar con números, así que lo vamos a convertir en un número pasandoselo a `Number(n)`.

/javascript-condicionales-lab/fizz-buzz.js

```
let n = prompt('Introduce un número...')
n = Number(n)
```

A continuación, vamos a poner las condiciones en dos sentencias `if`, una para ver si el número es múltiplo de 3 y la otra para comprobar si lo es de 5.

/javascript-condicionales-lab/fizz-buzz.js

```
let n = prompt('Introduce un número...')
n = Number(n)

if (n % 3 == 0) {

}

if (n % 5 == 0) {

}
```

Y por último vamos a generar el mensaje que se tiene que mostrar finalmente, que empezará con el número seguido de `Fizz`, `Buzz` o `Fizz Buzz`

/javascript-condicionales-lab/fizz-buzz.js

```
let n = prompt('Introduce un número...')
n = Number(n)

let msg = n
if (n % 3 == 0) {
  msg += ' Fizz'
}
```

```
if (n % 5 == 0) {
  msg += ' Buzz'
}

console.log(msg)
```

Una vez que tenemos el código, vamos a crear un archivo `index.html` en el que importaremos este script para comprobar que funciona correctamente.

/javascript-condicionales-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="fizz-buzz.js"></script>
</head>
<body>

</body>
</html>
```

Chapter 9. Bucles

Los bucles van a comprobar una condición. Si esta condición es `true`, el bloque de código se va a ejecutar. Una vez ejecutado el bloque de código, se vuelve a comprobar la condición y si sigue siendo `true` se vuelve a ejecutar otra vez, y así sucesivamente hasta que la condición sea `false` que será cuando deje de ejecutar ese bloque de código correspondiente al bucle.

9.1. FOR

El `for` se usa para cuando necesitamos ejecutar un bloque de código un número de veces conocido. Este el bucle más común a la hora de usarse. En este bucle la condición es el número de veces que se tiene que repetir el bloque de código. El valor que se usa en la condición tendremos que inicializarlo, e ir incrementandolo/decrementandolo para que no se quede en un bucle infinito al final de la ejecución del bloque.

```
for (var i = 0; i < 5; i++) {
  console.log('Iteración ' + (i + 1));
}
```

9.2. FOR IN

El `for in` es una versión del bucle `for` que se usa para iterar sobre Arrays u Objetos. En cada iteración se va a ir guardando en la variable la posición (si se está usando para iterar sobre un objeto, se va a guardar la *clave*).

```
for (var j in [1, 2, 3, 4]) {
  console.log('Posición del array en esta iteración' + j);
}
```

9.3. WHILE

En caso de no conocer el número de veces que se tiene que ejecutar el bloque de código, deberíamos usar el `while`, en el que la condición puede ser otra expresión que no sea un contador (como en el `for`) y el código se repetirá hasta que esta condición sea `false`.

```
var num = prompt('Dame un número... Introduce -1 si quieres terminar.');
while (num != -1) {
  console.log('Has introducido el número ' + num);
  num = prompt('Dame un número... Introduce -1 si quieres terminar.');
}
```

9.4. DO - WHILE

Este bucle es muy parecido al `while`. La única diferencia entre ellos, es que el bloque de código correspondiente a este bucle, se va a ejecutar la primera vez aunque la condición sea `false`. En este caso, primero se ejecuta el bloque, luego comprueba la condición, y a partir de aquí todo funciona igual que el `while`.

```
do {
  var num = prompt('Dame un número... Introduce -1 si quieres terminar.');
  console.log('Has introducido el número ' + num);
} while (num != -1);
```

9.5. Lab: Adivina el número al azar

En este laboratorio, usaremos los bucles para intentar adivinar el número que se va a generar de forma aleatoria. Para realizar el ejercicio tienes que tener en cuenta lo siguiente:

- El programa saca un número al azar entre 0 y 50.
- El usuario introduce el número que piensa que ha salido en un poput del navegador.
- Si el usuario acierta:
  - Se le indica que ha ganado.
  - Se le muestra el número de turnos jugados.
  - Y se le pregunta si quiere volver a jugar.
- Si el usuario no acierta:
  - Se le indica si el número que ha dicho es mayor o menor.
  - Se le vuelve a pedir un número.

Vamos a empezar por crear nuestra página HTML en la que importaremos el archivo de JavaScript en el que vamos a añadir el código del ejercicio.

/javascript-bucles-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>

</body>
</html>
```

Chapter 10. Funciones

Cuando las aplicaciones se van haciendo las aplicaciones más grandes, es importante organizar el código para que quede limpio y no repetirlo agrupando los bloques de código según la funcionalidad. Para hacer todo eso, vamos a usar las funciones.

Las **funciones** nos permiten agrupar una serie de instrucciones que se encargan de realizar una tarea específica. Si esta tarea se repite en distintas partes de nuestro código podemos usar las funciones para evitar repetir código en todos estos sitios. Estas funciones no se ejecutan directamente cuando se ejecuta el script, sino que se van a ejecutar cuando se realiza una llamada a la función.

Para crear una función se usa la palabra reservada `function` seguida del nombre de la función y el cuerpo de esta (entre llaves `{}`).

```
function saludar() {
  alert('Hola');
}
```

Una vez que una función está declarada, para ejecutar el código que contiene solo hay que llamarla de la siguiente forma:

```
saludar();
```

Estas funciones pueden recibir datos externos para poder realizar las tareas, y estos datos son los **parámetros** que se le pasan entre los paréntesis. Estos parámetros se comportan como **variables** dentro de la función.

```
function calcularArea(ancho, alto) {
  return ancho * alto;
}
var ancho = 5;
var area = calcularArea(ancho, 10);
```

En JavaScript, a las funciones que reciben unos parámetros se les puede pasar más de los que espera recibir. En este caso podemos acceder a los parámetros que se pasan de más en la variable **arguments**. Esta variable la tienen todas las funciones y en ella se encuentran todos los parámetros que recibe la función.

```
function devolverParametro(param1) {
  console.log('Argumentos: ', arguments);
  return param1;
}
var p = devolverParametro('Un texto', 2, true);
```

También podemos no pasarle parámetros aunque esté esperando recibirlos, en este caso estos parámetros que no reciban el valor cuando se llama a la función tendrán el valor **undefined**.

```
function devolverParametro(param1) {
  return param1;
}
var p = devolverParametro();
```

10.1. Funciones recursivas

Una **función recursiva** es aquella que se llama a sí misma para realizar unos calculas y evitar usar lo bucles. Estas funciones tienen que estar bien pensadas para que no se conviertan en un bucle infinito, tenemos que darle unos casos básicos en los que se terminará de llamar a si misma y devolverá el resultado.

```
function factorial(num) {
  if (num == 1) {
    return 1;
  } else {
    return factorial(num-1) * num;
  }
}

var factorial5 = factorial(5);
```

10.2. Expresión de funciones y funciones anónimas

Las **expresiones de funciones** son aquellas funciones que se encuentran donde el interprete espera encontrar una expresión, y en estos casos se suelen usar las **funciones anónimas** que son aquellas que no tienen un nombre, y se van a ejecutar en cuanto el interprete se las encuentra.

```
var calcularArea = function(ancho, alto) {
  return ancho * alto;
}

var area = calcularArea(5, 10);
```

10.3. Funciones autoejecutables

Estas funciones no tienen nombre, y se van a ejecutar una vez que el interprete se las encuentre en el script. El interprete tiene que tratar estas funciones como expresiones, por lo que se meten entre parentesis, donde se declara una función anónima y se llama añadiendo justo despues los parentesis (con los argumentos necesarios, en caso de necesitarlos).

```
var saludar = (function() {
    console.log('Hola');
}());
```

Puede ser que esa función necesite parámetros para que realice la tarea correctamente, y en ese caso, los parametros se le pasan en los paréntesis que realizan la llamada a la función.

```
var saludar = (function(nombre) {
    console.log('Hola ' + nombre);
}('Robb'));
```

Estas funciones se suelen usar aquellas veces que el código solo necesita ejecutarse una vez, por ejemplo, para asignar los *listeners* y los *manejadores de eventos* a los elementos cuando estos tienen que ejecutar algún código si el usuario realiza una acción (como pulsar un botón). También evita que haya conflictos entre variables con el mismo nombre que se encuentran en distintos scripts.

10.4. Lab: crea tu propia librería de traducciones

- Crear una función autoejecutable que contenga lo necesario para poder realizar traducciones desde el español a inglés y francés.
- Dentro tendremos los objetos por cada idioma con las traducciones.
- Tendrá una función que reciba el texto a traducir y el idioma en el que queremos la traducción, que por defecto será el inglés.

10.4.1. Creando la función autoejecutable

Empezamos creando una función autoejecutable donde meteremos toda la lógica para poder realizar las traducciones.

/javascript-funciones-fn-autoejecutables-lab/traductor.js

```
(function() {
  console.log('Se ejecuta')
})()
```

Una vez que vemos que se está ejecutando automáticamente, vamos a añadir los objetos con las distintas traducciones dentro de la función.

/javascript-funciones-fn-autoejecutables-lab/traductor.js

```
(function() {
  const EN = {
    hola: 'hello',
    adios: 'bye',
    'buenos dias': 'good morning'
  }
```

```
  const FR = {
    hola: 'salut',
    adios: 'adieu',
    'buenos dias': 'bonjour'
  }
})()
```

Una vez que tenemos los objetos, vamos a añadir la función que será accesible desde cualquier sitio, y que recibirá el texto del cual queremos la traducción y el idioma que por defecto será el inglés.

/javascript-funciones-fn-autoejecutables-lab/traductor.js

```
(function() {
  const EN = {
    hola: 'hello',
    adios: 'bye',
    'buenos dias': 'good morning'
  }
  const FR = {
    hola: 'salut',
    adios: 'adieu',
    'buenos dias': 'bonjour'
  }

  function traducir(texto, lang='en') {
    switch(lang) {
      case 'en':
        return EN[texto] ? EN[texto] : 'No tenemos la traducción que pides';
      case 'fr':
        return FR[texto] ? FR[texto] : 'No tenemos la traducción que pides';
      default:
        return 'En este momento no tenemos traducciones para este idioma :('
    }
  }
})()
```

Una vez que ya tenemos la función que se va a encargar de devolvernos la traducción pedida, vamos a devolver un objeto con dicha función para poder llamarla desde fuera de la función autoejecutable. También vamos a exportar la función autoejecutable para poder usar la función `traducir` desde cualquier archivo en el que la importemos.

/javascript-funciones-fn-autoejecutables-lab/traductor.js

```
module.exports = (function() {
  const EN = {
    hola: 'hello',
    adios: 'bye',
    'buenos dias': 'good morning'
  }
```

```
  const FR = {
    hola: 'salut',
    adios: 'adieu',
    'buenos dias': 'bonjour'
  }

  function traducir(texto, lang='en') {
    switch(lang) {
      case 'en':
        return EN[texto] ? EN[texto] : 'No tenemos la traducción que pides';
      case 'fr':
        return FR[texto] ? FR[texto] : 'No tenemos la traducción que pides';
      default:
        return 'En este momento no tenemos traducciones para este idioma :('
    }
  }

  return {
    traducir: traducir
  }
})()
```

10.4.2. Usando las traducciones en un archivo distinto

Ahora vamos a crearnos un archivo nuevo que sería donde vamos a necesitar hacer uso de las traducciones y donde vamos a importar aquello que nos devuelve la función autoejecutable.

Una vez importada, vamos a mostrar por consola los resultados de las traducciones pedidas, y para ello tenemos que llamar a la función `traducir` que se ha devuelto desde la función autoejecutable que se ha exportado del archivo `traductor.js`.

/javascript-funciones-fn-autoejecutables-lab/app.js

```
const Traductor = require('./traductor');

console.log(Traductor.traducir('hola'))
console.log(Traductor.traducir('hola', 'fr'))
console.log(Traductor.traducir('esta no la tenemos'))
console.log(Traductor.traducir('adios', 'fr'))
console.log(Traductor.traducir('buenos dias', 'en'))
console.log(Traductor.traducir('adios', 'este-idioma-no-lo-tenemos'))
```

Este archivo lo vamos a ejecutar usando `node` como aparece a continuación:

```
$ node app.js
```

Y una vez ejecutado nos mostrará los resultados de las traducciones como se muestra a continuación:

```
hello
salut
No tenemos la traducción que pides
adieu
good morning
En este momento no tenemos traducciones para este idioma :(
```

Chapter 11. Scopes

El lugar donde podemos usar las variables depende del lugar en el que las hemos declarado, y esto se conoce como **scope** de variables.

11.1. Variables locales

Cuando declaramos una variable (usando la palabra *var*) dentro de una función, la variable se crea de forma local a la función, por lo que no se podrá acceder al valor de esa variable desde el exterior de la función. Cada vez que se ejecuta la función se crea la variable, y cuando se termina de ejecutar se destruye esa variable por lo que no se guardará el valor que tenía para las próximas ejecuciones.

```
function suma(n1, n2) {
  var resultado = n1 + n2;
  return resultado;
}

function resta(n1, n2) {
  var resultado = n1 - n2;
  return resultado;
}

suma(5, 5);
console.log(resultado);
resta(6, 2);
```

11.2. Variables globales

Al definir una variable sin la palabra `var`, esta variable será global, y puede darse el caso de que conlleve comportamientos inesperados (conflicto de nombres de variables) por lo que hay que poner la palabra `var` siempre que no vayamos a declarar una variable global. También se toma como variable global aquella que se declara con la palabra `var`, y se declara fuera de cualquier función.

```
function suma(n1, n2) {
  num = n1;
  return num + n2;
}
function muestraConsola(mensaje) {
  console.log('Mensaje: ' + mensaje + ', global = ' + global);
}

muestraConsola('1 + 2 = ' + suma(1, 2));
muestraConsola('num -> ' + num);
var global = 5;
```

```
muestraConsola('4 + 6 = ' + suma(4, 6));
num = 10;
muestraConsola('num -> ' + num);
```

Chapter 12. Objetos

Los **objetos** agrupan un conjunto de variables y funciones que representan cualquier cosa que podamos necesitar, por ejemplo, un coche, una persona, una serie...

Las variables que se definen en el objeto se llaman **propiedades** y nos dan información sobre el objeto, por ejemplo la marca del coche, o el nombre de una persona.

Mientras que las funciones se llaman **métodos** y representan tareas asociadas a ese objeto, por ejemplo mostrar la velocidad a la que va el coche, o cambiar el valor que indica si una serie ha finalizado o no. Y para acceder a las propiedades del objeto dentro de un método se usa la propiedad `this` del objeto.

12.1. Creando objetos de forma literal

Un objeto se crea añadiendo estas claves-valores entre llaves como se puede ver a continuación:

```
var serie = {
  nombre: 'Vikings',
  temporadas: 5,
  episodios: 69,
  episodiosVistos: 45,
  episodiosPorVer: function() {
    return this.episodios - this.episodiosVistos;
  }
}
```

Una vez que hemos creado un objeto, podemos acceder a sus propiedades y métodos de varias formas.

12.2. Acceso a propiedades: Notación de punto

Está es la forma que se suele usar para acceder a las propiedades y métodos de los objetos. Solo hay que poner un `.` seguido del nombre de la propiedad/método al que queramos acceder detrás del objeto.

```
var numTemp = serie.temporadas;
var episodiosSinVer = serie.episodiosPorVer();
```

12.3. Acceso a propiedades: Usando corchetes

Está forma de acceso se suele usar cuando el nombre de la propiedad es un *número* (deberiamos de evitar poner números como nombres de las propiedades), o cuando vamos a usar el valor de una variable como nombre de la propiedad. En este caso hay que poner el nombre de la propiedad a la que queremos acceder entre corchetes.

```
var numEpisodios = serie['episodios'];
```

12.4. Creando objetos con el constructor

Otra forma de crear los objetos es usando `new Object()`, lo que crearía un objeto en blanco. Una vez creado el objeto, se le pueden añadir las propiedades y métodos usando la notación de punto o los corchetes (para acceder a la propiedad/método) y dandole un valor (al no estar definida esa propiedad, se añade con el valor dado).

```
var serieDirk = new Object(); // equivalente a -> var serieDirk = {};
serieDirk.nombre = "Dirk Gently's Holistic Detective Agency";
serieDirk['temporadas'] = 2;
serieDirk['episodios'] = 18;
serieDirk.episodiosVistos = 16;
serieDirk.episodiosPorVer = function() {
  return this.episodios - this.episodiosVistos;
};
```

12.5. Actualizando los objetos

Para actualizar el valor de las propiedades de los objetos, solo hay que darle un nuevo valor a estas (tanto con usando la notación de punto, o los corchetes).

```
serieDirk.episodiosVistos = 17;
serieDirk['episodiosVistos'] = 18;
```

Si se intenta actualizar una propiedad que no existe en el objeto, esta se creará con el valor que se le está asignando.

```
serieVikings.finalizada = false;
serieDirk.finalizada = false;
```

12.6. Eliminando propiedades

Se pueden eliminar las propiedades de un objeto usando **delete** seguido de la propiedad a eliminar.

```
delete serieVikings.finalizada;
```

En caso de querer unicamente limpiar el valor de la propiedad, sirve con asignarle un string vacio.

```
serieDirk.finalizada = '';
```

12.7. Creando varios objetos

Hasta ahora hemos creado dos objetos que representan lo mismo pero con distintos valores. Se pueden usar funciones como plantillas (clases) para crear estos objetos, a las cuales le vamos a pasar como parámetros los valores con los que vamos a inicializar sus propiedades. El nombre de las funciones constructoras empiezan en mayúscula por convención, de esta forma sirve para recordar que hay que usar la palabra **new** a la hora de crear los objetos.

```
function Serie (nombre, temporadas, episodios, episodiosVistos) {
  this.nombre = nombre;
  this.temporadas = temporadas;
  this.episodios = episodios;
  this.episodiosVistos = episodiosVistos;
  this.episodiosPorVer = function () {
    return this.episodios - this.episodiosVistos;
  }
}
```

La palabra **this** indica que la propiedad o el método pertenece al objeto que crea esta función. Se usa en lugar de usar el nombre del objeto como se ha visto anteriormente.

Para crear instancias de los objetos usando estas funciones constructoras, tenemos que ponerlas después de la palabra **new**.

```
var breakingBad = new Serie("Breaking Bad", 5, 62, 54);
var sonsOfAnarchy = new Serie("Sons of Anarchy", 7, 92, 92);
```

12.8. Arrays

Los **arrays** son un tipo especial de objeto que guarda un conjunto de pares clave-valor, pero en este caso la clave es la posición en la que se encuentran los elementos en la lista.

Se pueden combinar con los objetos para crear estructuras de datos más complejas, por ejemplo los arrays pueden guardar una serie de objetos, y los objetos pueden guardar arrays. En los objetos el orden en que aparecen las propiedades no es importante, pero en los arrays, la posición indica el orden de las propiedades.

```
sonsOfAnarchy.aparicionPersonajesTemporadas = {
  'jax': [1, 2, 3, 4, 5, 6, 7],
  'chibs': [1, 2, 3, 4, 5, 6, 7],
  'nero': [5, 6, 7]
};

breakingBad.datosTemporadas = [
  { anyo: 2008, notaMedia: 8.8 },
  { anyo: 2009, notaMedia: 8.5 },
```

```
  { anyo: 2010, notaMedia: 9.1 }
];
```

Chapter 13. Objetos intrínsecos

Los navegadores traen una serie de objetos intrínsecos que representan cosas como la ventana del navegador, o la página web que se está mostrando. Estos objetos nos van a permitir crear páginas web interactivas.

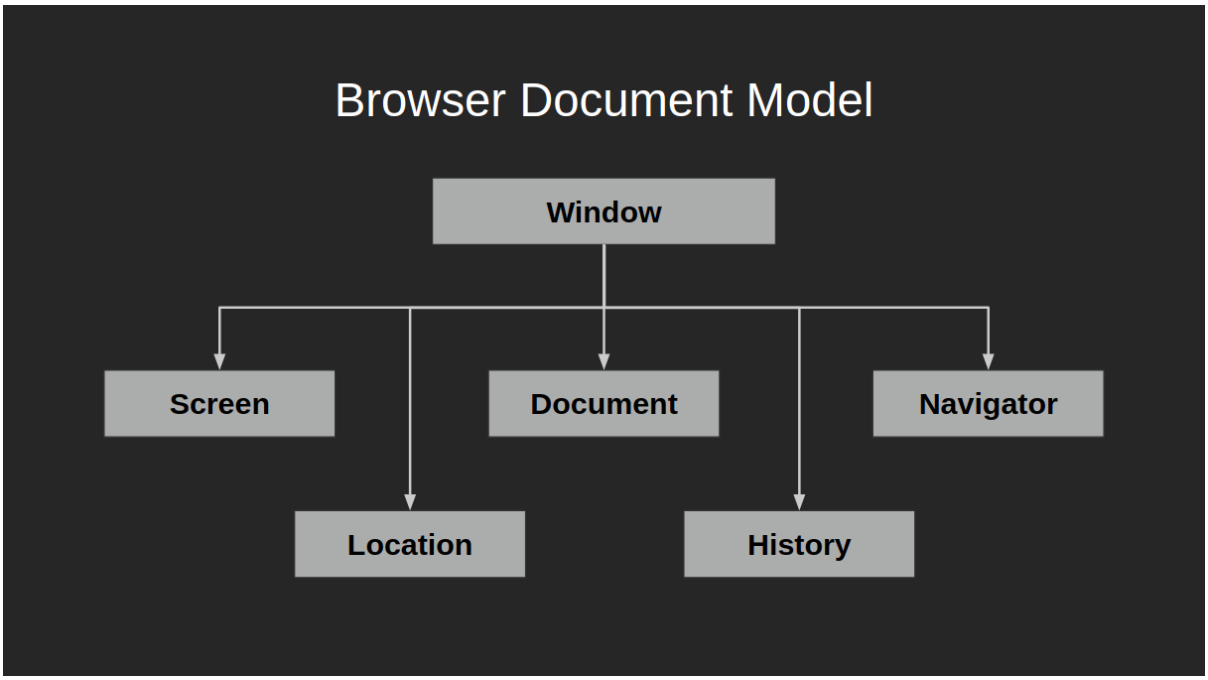
Los objetos que nosotros creamos, los diseñamos para ajustarlos a nuestras necesidades, mientras que los objetos intrínsecos, tienen definida una funcionalidad que es usada por muchos scripts. Estos objetos nos van a permitir acceder a información como el ancho de la ventana del navegador, las coordenadas donde se encuentra el ratón, el contenido de la página web… Y a estas propiedades podemos acceder al igual que hacemos con nuestros objetos.

Estos objetos los podemos dividir en tres grupos:

- **Browser Object Model**: este grupo contiene objetos que nos van a dar información sobre la ventana del navegador, el historial, la pantalla del dispositivo…
- **Document Object Model**: este grupo contiene objetos que representan el contenido de la página actual. Se crea un objeto por cada elemento dentro de la página (cabecera, footer, botón, campo de texto…).
- **Objetos Javascript Globales**: este grupo contiene objetos que representan cosas que vamos a necesitar usar en JavaScript como obtener la fecha y hora, u obtener el valor de constantes matemáticas…

13.1. Browser Object Model

En este grupo de objetos, el que está por encima del resto es el objeto **window** que es el que representa la ventana actual del navegador. Este objeto tiene otros objetos hijos, que agrupan información sobre otras características del navegador.



13.1.1. Window

El objeto **window** representa la ventana actual del navegador y este objeto tiene unas propiedades y métodos que se pueden ver a continuación:

Table 8. Propiedades de window

Propiedad	**Descripción**
window.innerHeight	Altura de la ventana
window.innerWidth	Ancho de la ventana
window.pageXOffset	Distancia que se ha hecho scroll horizontalmente en la ventana
window.pageYOffset	Distancia que se ha hecho scroll verticalmente en la ventana
window.screenX	Coordenada X a la que se encuentra el navegador de la esquina superior izquierda de la pantalla
window.screenY	Coordenada Y a la que se encuentra el navegador de la esquina superior izquierda de la pantalla

Table 9. Métodos de window

Método	**Descripción**	**Ejemplo**
window.alert(msg)	Muestra un popup con el mensaje *msg* y un botón OK	window.alert("Hola mundo");
window.confirm(msg)	Muestra un popup con el mensaje *msg* y dos botones, *Aceptar* (devuelve *true*) y *Cancelar* (devuelve *false*).	var res = window.confirm("Quieres continuar?");
window.prompt(msg)	Muestra un popup con el mensaje *msg*, un campo de texto y dos botones, *Aceptar* (devuelve el texto introducido) y *Cancelar* (devuelve *null*).	var res = window.prompt("Cómo te llamas?");
window.open(url)	Abre una pestaña nueva en el navegador con la URL *url*.	window.open("https://www.google.es");
window.print()	Abre la ventana de impresión del navegador con la página actual.	window.print();
window.scrollBy(x, y)	Hace scroll horizontal (*x* pixeles) y vertical (*y* pixeles).	window.scrollBy(0, 1000);

13.1.2. History

El objeto **history** es uno de los hijos del objeto **window**, lo que quiere decir que es un objeto que se encuentra dentro de otro objeto. Este objeto nos va a dar información sobre el historial del navegador y metodos para cambiarlo.

Table 10. Propiedades de History

Propiedad	Descripción
window.history.length	Devuelve el número de páginas almacenadas en la pila de navegación

Table 11. Métodos de History

Método	Descripción	Ejemplo
window.history.back()	Vuelve una página atrás. Es como pulsar el botón de *retroceder* en el navegador	window.history.back();
window.history.forward()	Va una página adelante. Es como pulsar el botón de *avanzar* en el navegador	window.history.forward();
window.history.go(numPaginas)	Va *numPaginas* hacia delante (valor positivo) o hacia atrás (valor negativo)	window.history.go(-2);

13.1.3. Location

El objeto **location** es otro de los hijos del objeto **window** y este se encarga de darnos información sobre la URL de la página actual.

Table 12. Propiedades de Location

Propiedad	Descripción
window.location.href	Devuelve la URL de la página actual
window.location.port	Devuelve el puerto
window.location.host	Devuelve el host
window.location.pathname	Devuelve el pathname
window.location.protocol	Devuelve el protocolo
window.location.search	Devuelve los parámetros de búsqueda de la página

Table 13. Métodos de Location

Método	Descripción	Ejemplo
window.location.reload()	Recarga la página actual	window.location.reload();

Método	Descripción	Ejemplo
window.location.replace(url)	Reemplaza la página actual por la de la URL *url*	window.location.replace(url);

13.1.4. Navigator

El objeto **navigator** también es hijo del objeto **window** y nos proporciona información sobre el navegador y acceso al hardware del dispositivo como puede ser a la batería o la localización.

Table 14. Propiedades de Navigator

Propiedad	Descripción
window.navigator.appName	Devuelve el nombre oficial del navegador (puede devolver un valor erroneo)
window.navigator.appVersion	Devuelve la versión del navegador (puede devolver un valor erroneo)
window.navigator.language	Devuelve el lenguaje del navegador elegido por el usuario
window.navigator.languages	Devuelve un array de lenguajes que va a usar el navegador por orden de preferencia
window.navigator.onLine	Devuelve un booleano indicando si el navegador se encuentra trabajando en línea
window.navigator.platform	Devuelve la plataforma en la que el navegador se encuentra trabajando (puede devolver un valor erroneo)
window.navigator.vendor	Devuelve el nombre del fabricante del navegador actual

Table 15. Métodos de Navigator

Método	Descripción	Ejemplo
window.navigator.getBattery()	Devuelve una promesa que cuando se cumple nos da información de la bateria	window.navigator.getBattery();
window.navigator.geolocation	Nos da acceso a la localización del dispositivo	window.navigator.geolocation.getCurrentPosition(function(position) { … });

Método	Descripción	Ejemplo
window.navigator.vibrate(patron)	Hace que aquellos dispositivos que lo soportan vibren siguiente el patrón de vibraciones y pausas. Este valor puede ser un número (tiempo que va a vibrar el dispositivo, solo una vez) o un array de números (alterna entre tiempo que vibra y tiempo que está en pausa)	window.navigator.vibrate();

13.1.5. Screen

El objeto **screen** también es hijo del objeto **window** y nos da información sobre la pantalla del dispositivo en que se ha abierto el navegador.

Table 16. Propiedades de Screen

Propiedad	Descripción
window.screen.width	Devuelve el ancho de la pantalla (no del navegador)
window.screen.height	Devuelve la altura de la pantalla (no del navegador)
window.screen.orientation	Devuelve información sobre la orientación de la pantalla

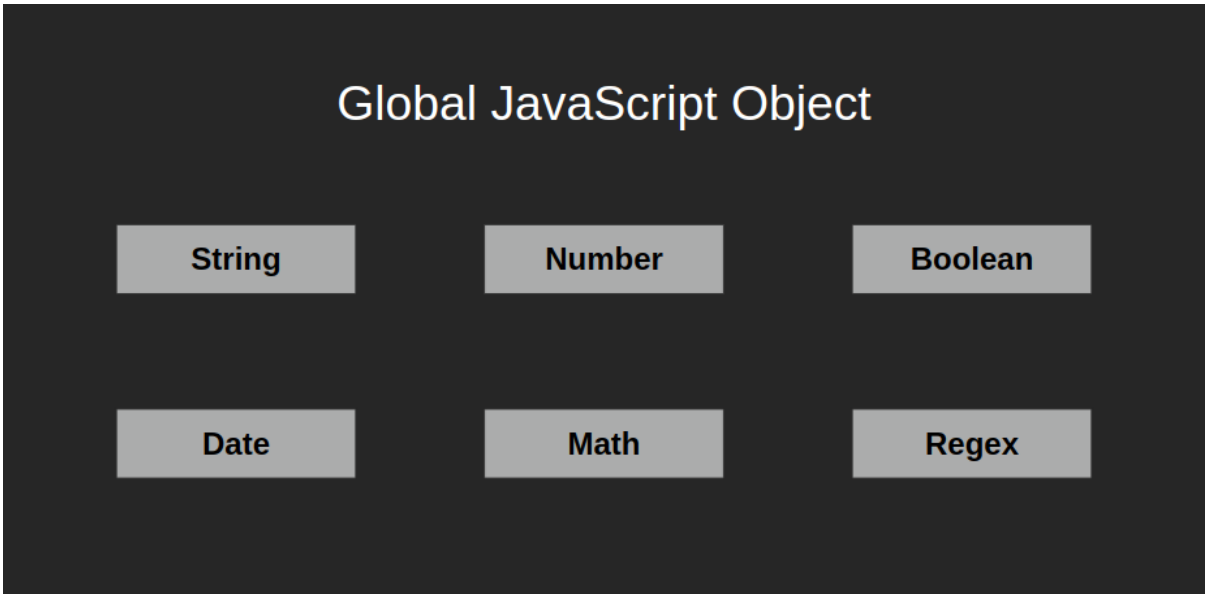
13.1.6. Document

El objeto **document** es otro de los hijos del objeto **window** y nos va a este nos va a permitir interactuar con la página web que se ha cargado, permitiendo modificarla mediante código. Este objeto lo veremos más adelante.

13.2. Global JavaScript Object

Los objetos globales son un grupo de objetos individuales que se encargan de algunas de las distintas partes del lenguaje JavaScript. Estos objetos los podemos dividir en:

- Aquellos que representan tipos de datos básicos (String, Boolean y Number).
- Aquellos que ayudan con algunos conceptos usados en el mundo real (Date, Math y Regex).



13.2.1. String

Siempre que tenemos un string podemos acceder a las siguientes propiedades y métodos para trabajar con el.

Table 17. Propiedades de String

Propiedad	**Descripción**
length	Devuelve la longitud del String

Table 18. Métodos de String

Método	**Descripción**	**Ejemplo**
toUpperCase()	Convierte el string a mayúsculas	"hola mundo".toUpperCase()
toLowerCase()	Convierte el string a minúsculas	"HOLA MUNDO".toLowerCase()
charAt(index)	Devuelve el caracter que se encuentra en la posición *index* del String	"Hola mundo".charAt(2)
indexOf(char)	Devuelve la posición en la que se encuentra el primer caracter *char* del String	"Hola mundo".indexOf('m')
lastIndexOf(char)	Devuelve la posición en la que se encuentra el último caracter *char* del String	"Hola mundo".lastIndexOf('m')
substring(index1, index2)	Devuelve el String que hay entre las dos posiciones dadas como parámetros, donde la primera posición (*index1*) se incluye, y la segunda (*index2*) no se incluye	"Hola mundo".substring(0, 6)

Método	Descripción	Ejemplo
split(char)	Separa el String en un Array con los substring separados por el caracter *char*	"Hola a todos".split(' ')
trim()	Elimina los espacios del inicio y final del String	" Hola a todos ".trim()
replace(str1, str2)	Reemplaza el *str1* por el *str2* en el String. Por defecto solo reemplaza la primera aparición de *str1* en el String.	"Hola a todos".replace('Hola', 'Adios')

13.2.2. Number

A continuación se muestran los métodos que se pueden usar con los números.

Table 19. Métodos de Number

Método	Descripción	Ejemplo
Number.isInteger(num)	Comprueba si el número *num* es un entero (*true*) o no (*false*)	Number.isInteger(2.4)
Number.isNaN(num)	Comprueba si el parámetro *num* es un NaN (*true*) o no (*false*)	Number.isNaN("un string")
toFixed(numDecimales)	Redondea un número decimal a tantos *numDecimales* decimales y lo devuelve como String	2.43467.toFixed(3)
toPrecision(num)	Devuelve un String con tantos números como *num* le llega como parámetro	2.43467.toPrecision(3)
toExponential(num)	Devuelve un String representando el número en una notación exponencial	0.0000000025.toExponential(3)

13.2.3. Array

Para trabajar con los Arrays, podemos usar las siguientes propiedades y métodos.

Table 20. Propiedades de Array

Propiedad	Descripción
length	Devuelve la longitud del Array

Table 21. Métodos de Array

Método	Descripción	Ejemplo
pop()	Elimina el último elemento del Array y lo devuelve	[1, 3, 5, 7, 2].pop()
push(it1, it2, ...)	Añade uno o varios elementos al final del array y devuelve la nueva longitud	[1, 3, 5, 7, 2].push(4, 3)
shift()	Elimina el primer elemento del Array y lo devuelve	[1, 3, 5, 7, 2].shift()
unshift(it1, it2, ...)	Añade uno o varios elementos al principio del array y devuelve la nueva longitud	[1, 3, 5, 7, 2].unshift(4, 11)
reverse()	Da la vuelta a todos los elementos del Array	[1, 3, 5, 7, 2].reverse()
sort()	Ordena los elementos del Array	[1, 3, 5, 7, 2].sort()
join(str)	Une todos los elementos del Array separandolos por el *str* y los devuelve en forma de String	[1, 3, 5, 7, 2].join(', ')
splice(pos, numElem, el1, el2, ...)	Desde la posición *pos* elimina *numElem* elementos y añade los elementos *el1, el2, ...*	[1, 3, 5, 7, 2].splice(1, 1, 10, 11)
slice(pos1, pos2)	Devuelve un Array con los elementos del Array entre *pos1* y *pos2*	[1, 3, 5, 7, 2].slice(1, 3)
indexOf(item)	Devuelve la posición del primer elemento *item* que se encuentra en el Array. Si no existe ese *item* en el Array, devuelve -1	[1, 3, 5, 7, 2].indexOf(3)
lastIndexOf(item)	Devuelve la posición del último elemento *item* que se encuentra en el Array. Si no existe ese *item* en el Array, devuelve -1	[1, 3, 5, 7, 2].lastIndexOf(0)
forEach()	Ejecuta una función por cada elemento del Array. Esta función recibe como parámetros el elemento y la posición en la que se encuentra	[1, 3, 5, 7, 2].forEach(function(elem, pos) {...})
map()	Devuelve un nuevo Array con los elementos que devuelve la función que recibe como parámetro	[1, 3, 5, 7, 2].map(function(elem, pos) {...})

Método	Descripción	Ejemplo
filter()	Devuelve un nuevo Array con aquellos elementos que cumplen con la condición establecida en la función que recibe como parámetro	[1, 3, 5, 7, 2].filter(function(elem, pos) {return true;})
concat(arr1, arr2, ...)	Devuelve un Array con la unión de varios Arrays	[1, 3, 5, 7, 2].concat([2, 5], [1, 8])

13.2.4. Date

Para poder trabajar con las fechas, tenemos que crear una instancia del objeto **Date** que nos dará la fecha actual. Esta fecha se obtiene del reloj del dispositivo, por lo que cada usuario obtendra la fecha actual correspondiente a su zona horaria.

```
var hoy = new Date();
```

También podemos crear la fecha que queremos representar pasandole al constructor del objeto unos parámetros con los siguientes formatos.

```
var fecha1 = new Date(1992, 5, 10);
var fecha2 = new Date(1992, 5, 10, 20, 14, 38);
var fecha3 = new Date(1992, 5, 10);
```

Table 22. Métodos de Date

Método	Descripción	Ejemplo
getDate()	Devuelve el día del mes (1-31)	fecha.getDate()
setDate()	Establece el día del mes (1-31)	fecha.setDate(2)
getMonth()	Devuelve el mes (0-11)	fecha.getMonth()
setMonth()	Establece el mes (0-11)	fecha.setMonth(6)
getFullYear()	Devuelve el año (4 dígitos)	fecha.getFullYear()
setFullYear()	Establece el año (4 dígitos)	fecha.setFullYear(2015)
getDay()	Devuelve el día de la semana (0-6)	fecha.getDay()
getHours()	Devuelve la hora (0-23)	fecha.getHours()
setHours()	Establece la hora (0-23)	fecha.setHours(20)
getMinutes()	Devuelve los minutos (0-59)	fecha.getMinutes()
setMinutes()	Establece los minutos (0-59)	fecha.setMinutes(13)
getSeconds()	Devuelve los segundos (0-59)	fecha.getSeconds()

Método	Descripción	Ejemplo
setSeconds()	Establece los segundos (0-59)	fecha.setSeconds(5)
getMilliseconds()	Devuelve los milisegundos (0-999)	fecha.getMilliseconds()
setMilliseconds()	Establece los milisegundos (0-999)	fecha.setMilliseconds(154)
getTime()	Devuelve el número de milisegundos que han pasado desde el 01/01/1970 (si la fecha es anterior a esta se devuelve un número negativo)	fecha.getTime()
setTime()	Establece el número de milisegundos que han pasado desde el 01/01/1970 (si la fecha es anterior a esta hay que establecer un número negativo)	fecha.setTime(711064800000)
toDateString()	Devuelve un String con solo la fecha	fecha.toDateString()
toTimeString()	Devuelve un String con solo la hora	fecha.toTimeString()
toString()	Devuelve la fecha y hora como String	fecha.toString()
getTimezoneOffset()	Devuelve en minutos la diferencia de la zona horaria para la configuración	fecha.getTimezoneOffset()

No hay métodos que nos devuelvan los nombres de los días o de los meses porque estos pueden variar entre los distintos lenguajes. Para obtenerlos, podemos usar un array para guardar los nombres y así poder acceder a ellos.

13.2.5. Math

El objeto **Math** tiene propiedades y métodos que podemos usar para realizar operaciones matemáticas.

Table 23. Propiedades de Math

Propiedad	Descripción
Math.PI	Devuelve el valor del número PI

Table 24. Métodos de Math

Método	Descripción	Ejemplo
Math.round(num)	Redondea un número al entero más cercano	Math.round(2.6)

Método	Descripción	Ejemplo
Math.sqrt(num)	Devuelve la raíz cuadrada de un número	Math.sqrt(9)
Math.ceil(num)	Redondea un número al entero superior más cercano	Math.ceil(2.2)
Math.floor(num)	Redondea un número al entero inferior más cercano	Math.floor(2.9)
Math.random()	Devuelve un número al azar entre 0 (incluido) y 1 (no incluido)	Math.random()
Math.min(n1, n2, ...)	Devuelve el número mínimo de los números que se le pasan como argumentos	Math.min(3, 6)
Math.max(n1, n2, ...)	Devuelve el número máximo de los números que se le pasan como argumentos	Math.max(3, 6)
Math.abs(num)	Devuelve el absoluto de un número	Math.abs(-3)
Math.sin(num)	Devuelve el seno de un número (en radianes)	Math.sin(90)
Math.cos(num)	Devuelve el coseno de un número (en radianes)	Math.cos(90)
Math.tan(num)	Devuelve la tangente de un número (en radianes)	Math.tan(90)

13.2.6. JSON

El objeto **JSON** tiene unos métodos que nos van a permitir trabajar con objetos JSON y objetos JavaScript.

Table 25. Métodos de JSON

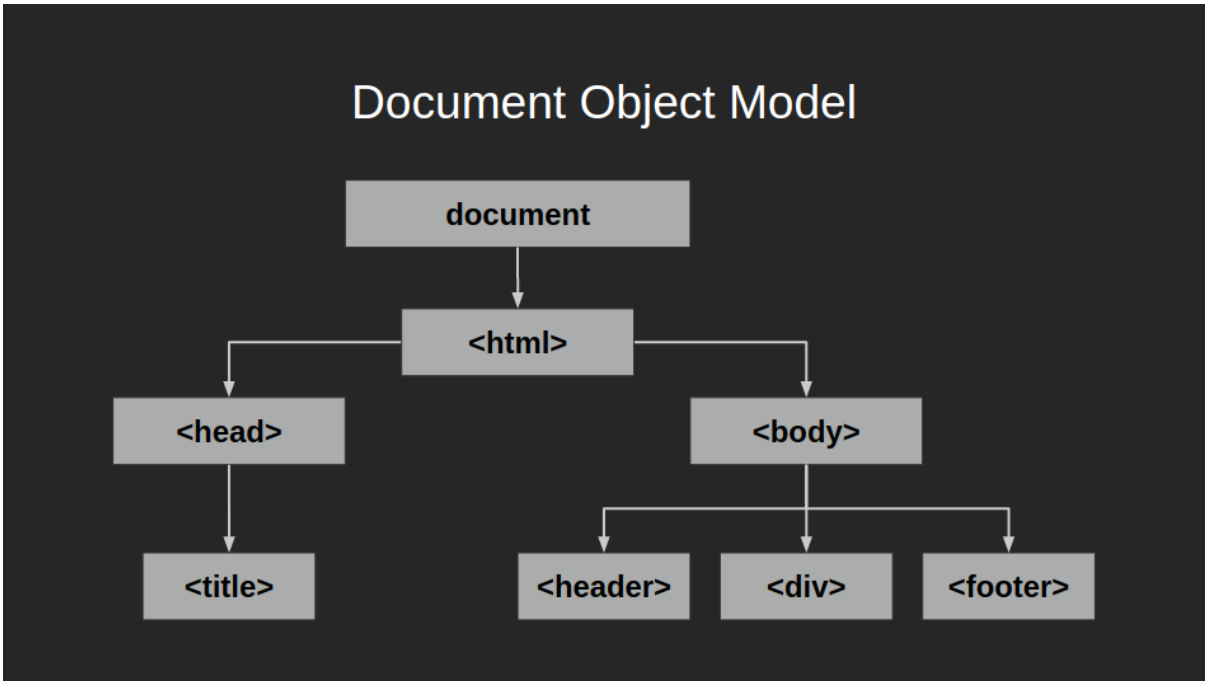
Método	Descripción	Ejemplo
JSON.parse(objJson)	Convierte un objeto JSON en un objeto JavaScript	JSON.parse("{\"nombre\":\"Ramsay\",\"apellido\":\"Bolton\"}");
JSON.stringify(obj)	Convierte un objeto JavaScript en un objeto JSON	JSON.stringify({ nombre: 'Ramsay', apellido: 'Bolton' })

Chapter 14. Document Object Model

Cuando el navegador carga la página web, crea un árbol del DOM que es un módelo de la página. Este módelo se almacena en la memoria del navegador y puede contener los siguientes tipos de nodos:

- Nodo documento: cada nodo del árbol del DOM representa un elemento, un atributo o un trozo de texto que se encuentra en la página web. El nodo del documento (`document`) representa toda la página web, por lo que es el nodo raíz del árbol y a partir del cual hay que navegar hasta llegar al resto de nodos y poder acceder a ellos.
- Nodo elemento: todos los elementos de HTML se encargan de estructurar la página web, y cada uno de ellos representa algo de esa página (una párrafo, un botón...), donde empieza un elemento y donde acaba. Para acceder a estos nodos, tendremos que usar algunos métodos que veremos más adelante. Y una vez que hayamos obtenido estos nodos, podremos acceder a sus nodos de texto y de atributos, tanto para obtener los valores, como para modificarlos.
- Nodo atributo: las etiqueta de apertura de un elemento HTML puede llevar atributos, y estos se representan como nodos en el árbol del DOM. Estos nodos no son hijos del elemento que lo contiene sino que son parte de ese elemento. Podemos obtener o modificar el valor de estos nodos usando unos métodos, una vez que ya hemos obtenido el elemento. Estos nodos se suelen modificar para cambiar el aspecto del elemento.
- Nodo texto: los nodos de texto nos devuelven el texto del elemento que los contiene. Estos nodos no pueden tener nodos hijos, por lo que si nos encontramos un nodo con texto, y parte de ese texto esta entre etiquetas HTML, ese elemento tendra un texto como hijo, y a su vez un nodo elemento como hijo (hermano del nodo texto), que tendrá el resto del texto como un nodo de texto que será hijo de este último.

Todos estos nodos se relacionan entre ellos como si fueran una familia y tuvieramos un árbol familiar. Tendremos a los padres, los hijos, los hermanos, los descendientes... Dentro del árbol del DOM, todos los nodos van a ser descendientes del nodo *document*.



Cada uno de estos nodos es un objeto que contiene unos métodos y unas propiedades. Nosotros podrémos modificarlos mediante los scripts, y cada cambio se va a realizar sobre el árbol del DOM, y por lo tanto se reflejará en el navegador.

Para saber de que tipo es un nodo, una vez que lo hemos obtenido, podremos acceder a su propiedad `nodeType` que nos devolverá un número. Este número se corresponde con el tipo de nodo que es, los cuales se pueden ver en la siguiente tabla:

Table 26. Tipos de Nodos

Tipo	Número
ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

14.1. Árbol del DOM

El árbol del DOM es el árbol que contiene la representación de la página web. Para poder acceder y modificar los elementos de este árbol, necesitamos seguir los siguientes pasos:

1. Buscamos el nodo que representa al elemento que queremos obtener para acceder a el o modificarlo.
2. Usamos su contenido (texto, nodos hijos o atributos).

Podemos acceder a los elementos de tres formas distintas:

- Seleccionando un nodo de elemento: para acceder a un nodo se puede usar `getElementById()`, `querySelector()` o los métodos que nos permiten pasar entre los nodos de elementos.
- Seleccionando múltiples nodos de elementos: para obtener múltiples nodos, se puede usar `getElementsByClassName()`, `getElementsByTagName()` y `querySelectorAll()`.
- Pasar a través de nodos de elementos: de esta forma podemos movernos entre los nodos que están relacionados, `parentNode`, previousSibling `, `nextSibling`, `firstChild` y `lastChild`.

Y una vez que tenemos los elementos, ya se puede trabajar con ellos, aunque con lo que se trabaja

es con los nodos que representan a esos elementos. En este caso podemos interactuar con los siguientes casos:

- Trabajar con el contenido HTML: podemos acceder al contenido HTML (`innerHTML`), al texto del elemento (`textContent`) o incluso crear nuevos nodos (`createElement()` y `createTextNode()`), añadirlos al árbol del DOM (`appendChild()`) y borrarlos (`removeChild()`).
- Acceso o actualización de los valores de los atributos: podemos obtener/modificar las clases (`className`) o el id (`id`), obtener/modificar un atributo (`getAttribute()` o `setAttribute`), comprobar que tiene el atributo (`hasAttribute()`) y borrarlo (`removeAttribute()`).
- Acceso o actualización de los nodos de texto: podemos obtener el valor de un nodo con `nodeValue` o incluso modificar ese valor asignandoselo, pero antes hay que poder acceder a ese nodo de texto usando `firstChild`.

Si obtenemos un elemento que vamos a usar varias veces, deberíamos de guardarlo en una variable para no tener que estar buscandolo cada vez que lo necesitemos.

14.2. Acceso a los elementos

Los métodos que se encargan de buscar en el árbol del DOM, pueden devolvernos un elemento, una lista de nodos (**NodeList**) o una colección de elementos HTML (**HTMLCollection**).

Algunas veces querremos obtener un único elemento y otras querremos obtener varios elementos. Si usamos un método que para obtener varios elementos, siempre nos devolverá una lista, incluso en el caso en que encuentre solo un elemento que coincida con lo que se busca. Y para acceder a estos elementos que nos llegan en la lista, usaremos los índices (acceso a un array).

También hay que tener en cuenta que si queremos que nuestra página sea rápida, tenemos que usar el método de búsqueda más adecuado en cada caso, es decir, si vamos a buscar un elemento por el id, deberíamos usar el método que se encarga de realizar una búsqueda por id en el árbol.

A continuación se van a mostrar todos los métodos que se pueden usar para acceder a los elementos del árbol del DOM.

14.2.1. Métodos que devuelven un elemento

Primero vamos a ver los métodos que nos permiten obtener un solo elemento.

getElementById()

Este método busca un elemento por su atributo **id**. Para poder devolver este elemento, tiene que existir un elemento con ese id dentro del documento HTML. Este es el método de búsqueda más rápido y eficiente que se puede usar porque reduce la búsqueda a un solo elemento, ya que dos elementos no pueden compartir el mismo valor para su atributo id.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var parrafo1 = document.getElementById('parrafo1');
  console.log(parrafo1);
</script>
</html>
```

querySelector()

Usa la sintaxis de los **selectores de css** para obtener un elemento. En caso de que haya múltiples elementos que coincidan con el selector pasado como parámetro, se devolverá solamente la primera coincidencia.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
```

```
<script>
  var parrafo2 = document.querySelector('body p');
  console.log(parrafo2);
</script>
</html>
```

14.2.2. Métodos que devuelven varios elementos

Ahora vamos a ver los métodos que nos devuelven varios elementos.

getElementsByClassName()

Devuelve una lista de elementos (*HTMLCollection*) que contienen la **clase** que se le pasa como parámetro al método. Este método es más rápido que el método `querySelectorAll()`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var titulos = document.getElementsByClassName('titulo');
  console.log(titulos);
</script>
</html>
```

getElementsByTagName()

Este método realiza la búsqueda por el **nombre de la etiqueta**, por lo que devolverá todos los elementos (*HTMLCollection*) **tagName** que se encuentre en el documento HTML. Este método también es más rápido que el método `querySelectorAll()`.

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var listItems = document.getElementsByTagName('li');
  console.log(listItems);
</script>
</html>
```

getElementsByName()

Devuelve una lista de elementos (*NodeList*) que contienen el atributo **name** que se le pasa como parámetro al método.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
```

```
<script>
  var labels = document.getElementsByName('input');
  console.log(labels);
</script>
</html>
```

querySelectorAll()

Usa la sintaxis de los **selectores de css** para obtener todos los elementos (*NodeList*) que coincidan con el selector pasado como parámetro en el método.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var parrafos = document.querySelectorAll('body p');
  console.log(parrafos);
</script>
</html>
```

14.2.3. Métodos que devuelven varios elementos

Estos métodos que pueden devolver más de un elemento, devuelven un NodeList o HTMLCollection aunque solo haya encontrado un único elemento. El orden en que se guardan los nodos en la lista, es el mismo que el orden en el que aparecen en la página HTML.

Una vez que ya tenemos la lista de elementos, podremos acceder a uno de ellos, o recorrerla para ejecutar código JavaScript por cada uno de los elementos.

Los NodeList son como arrays, pero en realidad no lo son. Son un tipo de objeto llamado **collection**, y como cualquier otro objeto, tiene una propiedad que nos dice el número de *items* que hay en la lista (`length`), y tiene un método (`item(index)`) que nos devuelve el nodo que se encuentra en la

posición que se le pasa, aunque se suele acceder a ellos como a cualquier array (`listaNodos[2]`).

Los dos métodos de acceso a los nodos necesitan saber el índice del elemento al cual queremos acceder.

Podemos usar el método `item(index)` que nos devolverá un nodo de la lista de elementos. El *index* que se le pasa como parámetro indica la posición del elemento que queremos obtener.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var parrafos = document.querySelectorAll('body p');
  console.log('El primer párrafo es: ', parrafos.item(0));
</script>
</html>
```

O podemos usar la sintaxis de los Arrays para acceder a estos elementos. Este método de acceso es más rápido que el anterior, por lo que se recomienda usarlo. Para acceder a un elemento solo hay que poner entre corchetes la posición en la que se encuentra.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
```

```
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var parrafos = document.querySelectorAll('body p');
  console.log('El segundo párrafo es: ', parrafos[1]);
</script>
</html>
```

Y por último, tanto los NodeList, como los HTMLCollection se pueden recorrer para ejecutar el mismo código por cada uno de los elementos que hay en la lista. En este caso, como no son Arrays, sino que son Objetos, es mejor recorrerlas usando un bucle `for` normal.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="titulo">Un título</h1>
  <p id="parrafo1">Un párrafo</p>
  <label for="nombre">Nombre: </label>
  <input type="text" name="input" id="nombre">
  <label for="email">Email: </label>
  <input type="email" name="input" id="email">
  <h1 class="titulo">Otro título</h1>
  <p>Otro párrafo</p>
  <ul>
    <li>Uno</li>
    <li>Dos</li>
  </ul>
</body>
<script>
  var parrafos = document.querySelectorAll('body p');

  for (var i = 0; i < parrafos.length; i++) {
    console.log(parrafos[i].firstChild.textContent);
  }
</script>
</html>
```

14.2.4. Pasar a través de los nodos de elementos

Cuando ya tenemos un nodo seleccionado, se puede acceder a otro nodo que esté relacionado con el, usando alguna de las siguientes propiedades:

- **parentNode**: esta propiedad devuelve el nodo que contiene al elemento actual.
- **previousSibling**: esta propiedad devuelve el nodo siguiente al elemento actual.
- **nextSibling**: esta propiedad devuelve el nodo anterior al elemento actual.
- **firstChild**: esta propiedad devuelve el primer nodo hijo del elemento actual.
- **lastChild**: esta propiedad devuelve el último nodo hijo del elemento actual.
- **children**: esta propiedad devuelve todos los hijos del elemento actual.

Hay que tener cuidado, porque la mayor parte de los navegadores añaden un nodo de texto vacío por cada espacio en blanco o por cada retorno de carro. Antes de tratar de usar las propiedades anteriores, hay que asegurarse de que no se encuentran esos nodos vacíos.

En caso de que el elemento no tenga un elemento anterior, siguiente, o hijos, devolverán un valor **null**, y nunca nos van a servir para modificar esos nodos (son **propiedades de lectura**).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <ul id="lista"><li id="uno">Uno</li><li id="dos">Dos</li><li id="tres">Tres</li><li id="cuatro">Cuatro</li></ul>
</body>
<script>
  var lista = document.getElementById('lista');
  var listItems = document.getElementsByTagName('li');
  console.log('Padre: ', listItems.item(0).parentNode);
  console.log('Primer hijo: ', lista.firstChild);
  console.log('Segundo hijo: ', listItems[0].nextSibling);
  console.log('Tercer hijo: ', listItems[3].previousSibling);
  console.log('Último hijo: ', lista.lastChild);
  console.log('Todos los hijos: ', lista.children);
</script>
</html>
```

14.3. Obtener/Actualizar contenido de los elementos

Una vez tenemos un elemento, podemos acceder a su contenido, incluso actualizarlo. Hay que tener en cuenta el tipo del contenido del elemento para elegir la forma correcta de hacerlo.

- Podemos ir hasta los nodos de texto, y de esta forma nos aseguramos que el elemento solo contiene texto, y no otros elementos.
- Podemos trabajar con el contenido del elemento, lo que nos dará acceso a los elementos hijos y

a los nodos de texto. Esta forma viene bien cuando tenemos que trabajar con un elemento que tiene nodos hijos y nodos de texto.

14.3.1. nodeValue

Cuando ya hemos seleccionado un nodo de texto, podemos acceder/modificar el texto usando la propiedad `nodeValue`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <style>
    .ocultar {
      display: none;
    }
  </style>
</head>
<body>
  <h1 id="uno">Hola <span class="ocultar">a todos!</span>mundo</h1>
  <h1 id="dos">Hola <span class="ocultar">a todos!</span>mundo</h1>
  <h1 id="tres">Hola <span class="ocultar">a todos!</span>mundo</h1>
</body>
<script>
  var elemUno = document.getElementById('uno');
  console.log(elemUno.firstChild.nodeValue);
  elemUno.firstChild.nodeValue = 'ooo';
</script>
</html>
```

14.3.2. textContent

Cuando tenemos un elemento, podemos obtener todo el texto que contiene el elemento y sus hijos usando `textContent`. Esta propiedad no nos va a devolver ninguna de las etiquetas, solo devuelve el texto.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <style>
    .ocultar {
      display: none;
    }
  </style>
</head>
<body>
  <h1 id="uno">Hola <span class="ocultar">a todos!</span>mundo</h1>
```

```
  <h1 id="dos">Hola <span class="ocultar">a todos!</span>mundo</h1>
  <h1 id="tres">Hola <span class="ocultar">a todos!</span>mundo</h1>
</body>
<script>
  var elemDos = document.getElementById('dos');
  console.log(elemDos.textContent);
</script>
</html>
```

En caso de usar esta propiedad para cambiar el texto, se cambiará todo el contenido, incluyendo las etiquetas que tuviera como nodos hijos el elemento.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <style>
    .ocultar {
      display: none;
    }
  </style>
</head>
<body>
  <h1 id="uno">Hola <span class="ocultar">a todos!</span>mundo</h1>
  <h1 id="dos">Hola <span class="ocultar">a todos!</span>mundo</h1>
  <h1 id="tres">Hola <span class="ocultar">a todos!</span>mundo</h1>
</body>
<script>
  var elemDos = document.getElementById('dos');
  console.log(elemDos.textContent);
  elemDos.textContent = 'ooo';
</script>
</html>
```

14.3.3. innerText

Actua como la propiedad anterior, aunque esta al no estar en ningún estandar, los navegadores no tienen porque tenerla implementada, y puede llegar a darse el caso en que falle la aplicación. Además de que ignora el texto que se encuentra en las etiquetas que se han ocultado mediante *CSS* (`display:none`) o alguna propiedad de *HTML* (`hidden`). Esta propiedad deberiamos de evitar usarla.

14.4. Añadir/Eliminar contenido HTML

Hemos visto como modificar los elementos que ya existen en el DOM, y como acceder a su contenido. Ahora vamos a ver como se pueden añadir nuevos elementos mediante JavaScript, y como eliminar los que ya están en el DOM. Para realizar esto tenemos dos formas de hacerlo, una es usando la propiedad `innerHTML` y la otra es usando los métodos de manipulación del DOM.

14.4.1. innerHTML

Esta propiedad se puede usar en cualquier nodo elemento. Puede ser usada tanto para obtener contenido, como para modificarlo y eliminarlo. Para actualizar el elemento, podemos asignarle a esta propiedad un string (que puede contener lenguaje de marcado para añadir elementos hijos) con el contenido. En cuanto a eliminar un elemento, lo que hay que hacer es igualar la propiedad a un string vacío.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
</head>
<body>
</body>
<script>
  var body = document.body;
  var contenido = '<ul id="mascotas">' +
                              '<li id="perro">Un Perro</li>' +
                              '<li id="gato">Un Gato</li>' +
                              '<li id="canario">Un Canario</li>' +
                              '</ul>';
  body.innerHTML = contenido;
</script>
</html>
```

Este método es mejor usarlo cuando vamos a cambiar fragmentos enteros de código.

14.5. Métodos de manipulación del DOM

Con estos métodos que vamos a ver, es más fácil acceder a nodos individuales para modificarlos o eliminarlos. Estos métodos son más seguros de usar que el `innerHTML` (ya que este puede dar lugar a más fallos si escribimos algo mal), aunque requiren usar mucho más código.

Con estos métodos vamos a poder crear nodos de texto, nodos de elementos, eliminarlos, añadir un nodo dentro de otro para ir creando el árbol del DOM...

Añadir elementos al DOM requiere seguir los siguientes tres pasos:

- Crear un elemento (nodo elemento)
- Crear el contenido (nodo texto)
- Añadirlo al DOM

Los métodos que vamos a usar para seguir esos tres pasos son los siguientes:

14.5.1. createElement()

Este método se encarga de crear un **nodo elemento**. Cuando se ha creado el nodo, todavía no se

encuentra en el DOM, hasta que nosotros lo añadamos. Este método recibe como parámetro el *nombre de la etiqueta* que queremos crear.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Document</title>
</head>
<body>
</body>
<script>
  var elemH1 = document.createElement('h1');
</script>
</html>
```

14.5.2. createTextNode()

Este método se encarga de crear un **nodo texto**. Recibe como parámetro el *texto* que queremos mostrar dentro del elemento. En caso de querer crear un noco elemento vacío, nos podemos saltar este paso.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Document</title>
</head>
<body>
</body>
<script>
  var elemH1 = document.createElement('h1');
  var textH1 = document.createTextNode('Un título');
</script>
</html>
```

Una vez que ya se ha creado el nodo de texto, se lo tenemos que añadir al elemento que lo va a mostrar, y para eso usaremos el método que vamos a ver a continuación.

14.5.3. appendChild()

Este es el método que se encarga de añadir un elemento, como elemento hijo de otro. Este elemento que queremos añadir se le pasa como parámetro.

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="utf-8">
  <title>Document</title>
</head>
<body>
</body>
<script>
  var elemH1 = document.createElement('h1');
  var textH1 = document.createTextNode('Un título');
  elemH1.appendChild(textH1);
  var body = document.body;
  body.appendChild(elemH1);
</script>
</html>
```

14.5.4. insertBefore()

Este método añade el elemento que se le pasa como primer parámetro justo antes del elemento que se le pasa como segundo parámetro. El encargado de llamar a este método es el elemento que contiene a los otros dos (elemento padre).

14.5.5. replaceChild()

Este método reemplaza el elemento que se le pasa como primer parámetro por el elemento que se le pasa como segundo parámetro. El encargado de llamar a este método es el elemento que contiene a los otros dos (elemento padre).

Para eliminar un elemento del DOM vamos a seguir los siguientes pasos:

- Obtenemos el elemento que queremos eliminar.
- Obtenemos el elemento padre del elemento a eliminar.
- Eliminamos el elemento hijo del elemento padre.

El método para eliminar los elementos es el siguiente:

14.5.6. removeChild()

Este método se encarga de eliminar el elemento hijo que se le pasa como parámetro, del elemento padre que es el que se encarga de llamar al método. Cuando eliminamos un elemento, también se eliminan todos los elementos hijos de este.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <ul id="lista">
```

```
    <li id="perro">Perro</li>
    <li id="gato">Gato</li>
    <li id="canario">Canario</li>
  </ul>
</body>
<script>
  var lista = document.getElementById('lista');
  var gato = document.getElementById('gato');
  lista.removeChild(gato);
</script>
</html>
```

14.6. Nodos de atributos

Los nodos de elementos también pueden contener nodos de atributos a los cuales podemos acceder para obtener, modificar o eliminar alguna de las propiedades que tiene el elemento. En este caso tenemos que seguir los siguientes pasos:

- Obtener el nodo del elemento que contiene el nodo atributo
- Realizar alguna acción sobre el nodo atributo

Los métodos que vamos a usar para realizar las acciones antes mencionadas son:

14.6.1. getAttribute()

Con este método vamos a poder obtener el valor del atributo que se le pasa como parámetro.

14.6.2. setAttribute()

Con este método vamos a poder cambiar el valor del atributo. Tanto el atributo al que se le va a cambiar el valor, como el valor nuevo, se tienen que pasar como parámetros. En caso de que el atributo que se va a cambiar no exista, se crea con el valor que se le está asignando.

14.6.3. hasAttribute()

Este método comprueba si el nodo del elemento tiene el atributo que se le ha pasado como parámetro.

14.6.4. removeAttribute()

Este método nos permite eliminar el atributo que se le pasa como parámetro del nodo del elemento.

Además de estos métodos, también podemos acceder a los atributos a través de propiedades como las que se muestran a continuación:

Table 27. Propiedades

Propiedad	Descripción
attributes	Devuelve un objeto con todos los atributos del elemento
className	Devuelve o asigna el valor al atributo *class* del elemento
id	Devuelve o asigna el valor al atributo *id* del elemento
href	Devuelve o asigna el valor al atributo *href* del elemento
src	Devuelve o asigna el valor al atributo *src* del elemento
checked	Devuelve o asigna el valor al atributo *checked* del elemento
title	Devuelve o asigna el valor al atributo *title* del elemento
type	Devuelve o asigna el valor al atributo *type* del elemento

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style>
    .subrayado {
      text-decoration: underline;
    }
  </style>
</head>
<body>
  <h1 id="titulo">Título subrayado</h1>
  <input type="text" disabled id="btn-submit">
  <p id="parrafo">Un párrafo subrayado</p>
</body>
<script>
  var h1Elem = document.getElementById('titulo');
  console.log('El elemento h1' + (h1Elem.hasAttribute('class') ? '' : ' no') + ' tiene un atributo clase');
  h1Elem.className = 'subrayado';
  console.log('El elemento h1' + (h1Elem.hasAttribute('class') ? '' : ' no') + ' tiene un atributo clase');

  var pElem = document.getElementById('parrafo');
  pElem.setAttribute('class', 'subrayado');

  var inputElem =  document.getElementById('btn-submit');
  console.log('Type del input: ' + inputElem.getAttribute('type'));
  inputElem.setAttribute('type', 'submit');
  console.log('Type del input: ' + inputElem.getAttribute('type'));

  console.log('Atributos del input: ', inputElem.attributes);
  inputElem.removeAttribute('disabled');
</script>
```

```
</html>
```

Chapter 15. Eventos

Cuando estamos navegando por internet, el navegador emite distintos tipos de eventos (de ratón, de teclado…), que es la forma que tiene de decirnos lo que está ocurriendo.

Los **eventos** se van creando según los usuarios interactuan con el navegador (pulsan un botón, escriben algo en un campo de texto…). Cuando estos eventos emiten, se pueden detectar para ejecutar funciones que tenemos en JavaScript. Aunque el evento sea el mismo, puede variar la función que se va a ejecutar dependiendo de que elemento sea el que haya detectado el evento. Estas funciones que se ejecutan, suelen dar alguna respuesta, por ejemplo actualizando el contenido de la página.

15.1. Tipos de eventos

Los eventos que el navegador emite se pueden dividir en distintos tipos, los que emite el teclado, el ratón, un formulario… Y estos eventos se suelen usar para ejecutar una función con código JavaScript cuando se emiten.

Table 28. Eventos de la interfaz

Evento	Descripción
load	Cuando la página se ha terminado de cargar
resize	Cuando se cambia el tamaño de la ventana del navegador
scroll	Cuando se hace scroll en la página

Table 29. Eventos de teclado

Evento	Descripción
keydown	El usuario pulsa cualquier tecla. Se emite el evento mientras la tecla está pulsada
keyup	El usuario deja de pulsar la tecla
keypress	El usuario pulsa una tecla (letras, números, caracteres especiales, espacio y enter). Se emite el evento mientras la tecla está pulsada

Table 30. Eventos de ratón

Evento	Descripción
click	El usuario pulsa sobre un elemento
dblclick	El usuario pulsa sobre un elemento dos veces
mousedown	El usuario pulsa el ratón mientras está sobre el elemento

Evento	Descripción
mouseup	El usuario deja de pulsar el ratón mientras está sobre el elemento. Si deja de pulsarlo en otro elemento distinto, no se emite el evento
mousemove	Cuando el usuario mueve el ratón
mouseover	Cuando el usuario mueve el ratón sobre un elemento
mouseout	Cuando el usuario mueve el ratón fuera del elemento

Table 31. Eventos de foco

Evento	Descripción
focus/focusin	Cuando el elemento obtiene el foco
blur/focusout	Cuando el elemento pierde el foco

Table 32. Eventos de formularios

Evento	Descripción
input	Mientras cambia el valor en cualquier elemento `input` o `textarea`
change	Cuando cambia el valor de algún campo del formulario
submit	Cuando se envía el formulario (usando un botón de tipo `submit` o pulsando la tecla `enter`)
reset	Cuando se resetea el formulario
cut	Cuando el usuario corta el contenido de algún campo del formulario
copy	Cuando el usuario copia el contenido de algún campo del formulario
paste	Cuando el usuario pega contenido en algún campo del formulario
select	Cuando el usuario selecciona texto en algún campo del formulario

15.2. Manejadores de eventos

Cuando el usuario interactua con la página, ocurren una serie de pasos a los cuales vamos a llamar **manejadores de eventos** o **event handlers**. Estos pasos son:

- Se selecciona el elemento sobre el que se quiere detectar un evento
- Se indica que tipo de evento se quiere detectar

- Se establece el código que se va a ejecutar cuando ocurra el evento sobre ese elemento

Por ejemplo, si queremos mostrar un mensaje por consola cuando se pulse sobre un botón, tendríamos que seguir los siguientes pasos. Primero obtenemos el botón usando los métodos de búsqueda del DOM. Después le asignamos el evento `click` a ese botón que hemos obtenido. Y por último, al asignar el evento se le indica que código vamos a ejecutar, en este caso se le dirá que ejecute la función `saludar()`.

Podemos manejar estos eventos de tres formas distintas, es decir, que podemos diferenciar tres tipos diferentes de *event handlers*.

15.2.1. Manejadores de eventos en etiquetas HTML

En las primeras versiones de HTML se le asignaron a los elementos unos atributos con los mismos nombres que los eventos, y los cuales pueden detectar estos eventos y reaccionar a ellos. Estos atributos reciben como valor la función que se quiere ejecutar.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <button type="button" onclick="console.log('Hola');">Saludar</button>
</body>
</html>
```

Esta primera forma es una **mala práctica** y hay que evitar usarla porque ensucia el código HTML. Debería de estar separado el código JavaScript del código HTML.

15.2.2. Manejadores de eventos en el DOM

Los event handlers del DOM nos permiten separar el código JavaScript del código HTML de una forma sencilla. La desventaja de este método es que solo podemos asignar una función a cada evento para un mismo elemento.

Para poder usarlo, tendremos que igualar la *referencia* a la función que se quiere ejecutar (o una función anónima) al `elemento.onevent`. Se le asigna la referencia a la función (sin parentesis) para que no se ejecute la función hasta que se dispare el *event handler*.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
```

```
  <button id="btn" type="button">Saludar</button>
</body>
<script>
  var btn = document.getElementById('btn');
  btn.onclick = saludar;

  function saludar() {
    console.log('Hola');
  }
</script>
</html>
```

15.2.3. Event Listeners

Los **event listeners** se introdujeron como mejora en la especificación del DOM que salió en el año 2000. Es la forma que más se usa para manejar los eventos. La sintaxis es algo distinta a la de los métodos anteriores. Este método soluciona el problema del anterior, que no podía asignarse nada más que una función a un evento para el mismo elemento. La desventaja de este método es que los navegadores antiguos no lo soportan.

En este caso para usarlo tendremos que ejecutar el método `addEventListener()` del elemento, y pasarle como parámetros, el evento que se quiere detectar, y la función que se quiere ejecutar cuando el evento ocurra. En este caso, el evento que se le pasa a la función **no va precedido de *on***.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <button id="btn" type="button">Saludar</button>
</body>
<script>
  var btn = document.getElementById('btn');
  btn.addEventListener('click', saludar);

  function saludar() {
    console.log('Hola');
  }
</script>
</html>
```

Podemos eliminar los *event listeners* asignados a un elemento usando el método `removeEventListener()` del elemento, pasandole los mismos parámetros que cuando se lo hemos asignado.

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <button id="btn1" type="button">Quitar event listener del boton 2</button>
  <button id="btn2" type="button">Saludar</button>
</body>
<script>
  function saludar() {
    console.log('Hola');
  }

  var btn2 = document.getElementById('btn2');
  btn2.addEventListener('click', saludar);

  var btn1 = document.getElementById('btn1');
  btn1.click = function() {
    btn2.removeEventListener('click', saludar);
  }
</script>
</html>
```

15.3. Objeto Event

Cuando ocurre un evento podemos acceder a la información del evento a través del objeto **Event** que se crea. Este objeto lo recibimos como parámetro en la función de callback que se ejecuta cuando se detecta dicho evento. Este objeto puede darnos información como sobre que elemento ha ocurrido el evento (propiedad `target`), que tipo de evento se ha disparado (propiedad `type`)...

15.4. Cambiando el comportamiento por defecto de los eventos

El objeto **event** tiene algunos métodos que permiten cambiar el comportamiento que tienen por defecto algunos elementos frente a unos eventos y también permite cambiar la forma en que los elementos superiores actuan frente a un evento de un componente que es descendiente de estos.

15.4.1. preventDefault()

Algunos eventos tienen comportamientos por defecto, como hacer *click* en un *link* que nos lleva a otra página. Estos comportamientos se pueden evitar usando el método `preventDefault()`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
```

```
</head>
<body>
  <a id="link" href="www.toggl.com">Ir a Toggl</a>
</body>
<script>
  var link = document.getElementById('link');
  link.addEventListener('click', function(e) {
    e.preventDefault();
    console.log('Deberías de haber ido a la página, pero hemos quitado el comportamiento por defecto');
  });
</script>
</html>
```

15.4.2. stopPropagation()

A veces nos encontramos con que dos elementos de HTML (uno dentro del otro) tienen un *event listener* escuchando al mismo evento. En este caso cuando ocurre ese evento en el elemento que está más abajo en el árbol, se disparará su *event handler* y todos los *event handlers* de los elementos que lo contienen y es muy posible que ocurra un efecto no deseado. Con el método `stopPropagation()` se consigue parar la propagación del evento hacía arriba.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <div id="caja">
    <button type="button" id="btn">Pulsa</button>
  </div>
</body>
<script>
  var caja = document.getElementById('caja');
  var btn = document.getElementById('btn');

  caja.addEventListener('click', function(e) {
    console.log('Has pulsado en el div!!!');
  });

  btn.addEventListener('click', function(e) {
    e.stopPropagation();
    console.log('Has pulsado en el botón!!!');
  });
</script>
</html>
```

Chapter 16. This

La palabra **this** en JavaScript es un poco diferente a lo suele ser en otros, lenguajes. En JavaScript, esta palabra, hace referencia al objeto donde se encuentra una función o al propietario de la función donde se está usando.

Si mostramos el valor de `this` en el script, nos dirá que su valor es el objeto **window**.

```
console.log(this); // Window
```

Cuando usamos dentro de una función el valor de `this`, este sigue siendo el objeto **window**. Esto se debe a que el propietario es el objeto *window*, es decir, no se encuentra dentro de ningún objeto que sea su propietario.

```
function a () {
  console.log(this);
}

a();
```

En caso de añadir la función en un objeto, el valor de `this` será el objeto (es el propietario) que se encarga de invocar a la función. En este caso, *persona* es el objeto que está llamando al método.

```
var persona = {
  nombre: 'Tony',
  apellido: 'Stark',
  nombreCompleto: function () {
    console.log(this);
    return this.nombre + ' ' + this.apellido;
  }
};

persona.nombreCompleto();
```

Ahora vamos a añadir una función que muestra los datos antes de devolver el nombre completo.

En este caso no vamos a poder acceder a los datos del objeto desde esta nueva función, porque no tiene propietario (se llama sin usar el objeto).

```
var persona = {
  nombre: 'Tony',
  apellido: 'Stark',
  nombreCompleto: function () {
    console.log(this);
    var muestraDatos = function () {
      console.log(this.nombre + ' ' + this.apellido);
```

```
    }
    muestraDatos(); // undefined undefined
    return this.nombre + ' ' + this.apellido;
  }
};

persona.nombreCompleto();
```

Esto lo podemos arreglar de varias formas. A continuación vamos a ver algunas de ellas.

16.1. Self o That

La primera de ellas es usar el patrón del **self** o **that**. Esto consiste en crear una variable local a la cual le vamos a asignar el valor de `this`, antes de la función. De esta forma dentro de la función que hemos creado, no podremos usar el objeto `this`, pero si que podremos usar el objeto en el que hemos guardado el valor que tiene `this` fuera de esa función.

```
var persona = {
  nombre: 'Tony',
  apellido: 'Stark',
  nombreCompleto: function () {
    console.log(this);
    var self = this;
    var muestraDatos = function () {
      console.log(self.nombre + ' ' + self.apellido);
    }
    muestraDatos();
    return this.nombre + ' ' + this.apellido;
  }
};

persona.nombreCompleto();
```

16.2. Bind

Otra forma de arreglar esto es usando el método `bind(this)` el cual devuelve una función igual que la que queremos usar, pero con un pequeño cambio. A esta función se le asigna como valor del `this` el valor que se le pasa al método `bind` como parámetro.

```
var persona = {
  nombre: 'Tony',
  apellido: 'Stark',
  nombreCompleto: function () {
    console.log(this);
    var muestraDatos = function () {
      console.log(this.nombre + ' ' + this.apellido);
    }
```

```
    var mostrarDatos = muestraDatos.bind(this);
    mostrarDatos();
    return this.nombre + ' ' + this.apellido;
  }
};

persona.nombreCompleto();
```

Chapter 17. Prototipos

Los **prototipos** se usan en JavaScript como una forma de definir propiedades y funcionalidad que se le aplicarán a todas las instancias de los objetos. Una vez que se definen las propiedades a través de un **prototype**, estas se convierten en propiedades de todas las instancias del objeto al que se le han añadido.

Los prototipos tienen un proposito similar al de las clases en cualquier lenguaje común que esté orientado a objetos.

Todas las funciones tienen una propiedad **prototype** que inicialmente hace referencia a un objeto vacío. Esta propiedad no sirve para nada hasta que las funciones son usadas como **constructores**, es decir, usando la palabra `new` para invocar la función como una función constructora.

El concepto de **clases** en JavaScript no existe (por ahora), pero hemos visto como usar `new` con una función constructora para crear un nuevo objeto. Hemos añadido las propiedades y métodos en la función constructora, pero también podemos hacerlo a través de la propiedad **prototype** que se encarga de proveer propiedades para las nuevas instancias de los objetos.

Cuando se usa el operador *new* para crear una nueva instancia de un objeto, este lo que hace es reservar un espacio de memoria para guardar el objeto que se va a crear. Al añadirle las propiedades y métodos en la función constructora, estamos usando la palabra `this` para guardarlas en el espacio que se ha reservado para el objeto, por lo tanto, si creamos 1000 objetos, tendremos 1000 veces los mismos métodos, y esto no es conveniente hacerlo porque estamos gastando bastantes recursos.

No es conveniente hacerlo con los métodos, porque al final estamos creando lo mismo una vez por objeto, pero las propiedades si que hay que añadirlas en la función constructora, porque cada una de ellas va a ser distinta a la de otros objetos y no queremos modificar una propiedad de un objeto al cambiar la de otro objeto diferente.

Como tenemos `this`, que hace referencia al objeto concreto, se puede crear un método común a todos los objetos que sean del mismo tipo, ahorrando así bastante memoria, usando **prototype**.

Prototype es una plantilla con la que se crean los objetos. Si se modifica esta plantilla (el *prototype*), todos los objetos que se creen con ella, se modificarán también, incluso aquellos que ya habían sido creados antes de modificarla.

```
function Coche1(marca, modelo, color, sonido) {
  this.marca = marca;
  this.modelo = modelo;
  this.color = color;
  this.sonido = sonido;
}

Coche1.prototype.tocaElClaxon = function () {
  console.log(this.sonido + '!!! ' + this.sonido + '!!!');
```

```
}
```

Chapter 18. AJAX

AJAX (Asynchronous JavaScript and XML) es una técnica que nos permite cargar datos en nuestra página HTML sin la necesidad de refrescarla. Normalmente, los datos que se cargan se envían en formato JSON.

Al poder cargar el contenido en la página sin la necesidad de recargarla, estamos mejorando la experiencia de usuario porque este no va a tener que esperar a que toda la página HTML se cargue, sino que solo esperará a que se actualicen los datos en una parte del contenido de la página.

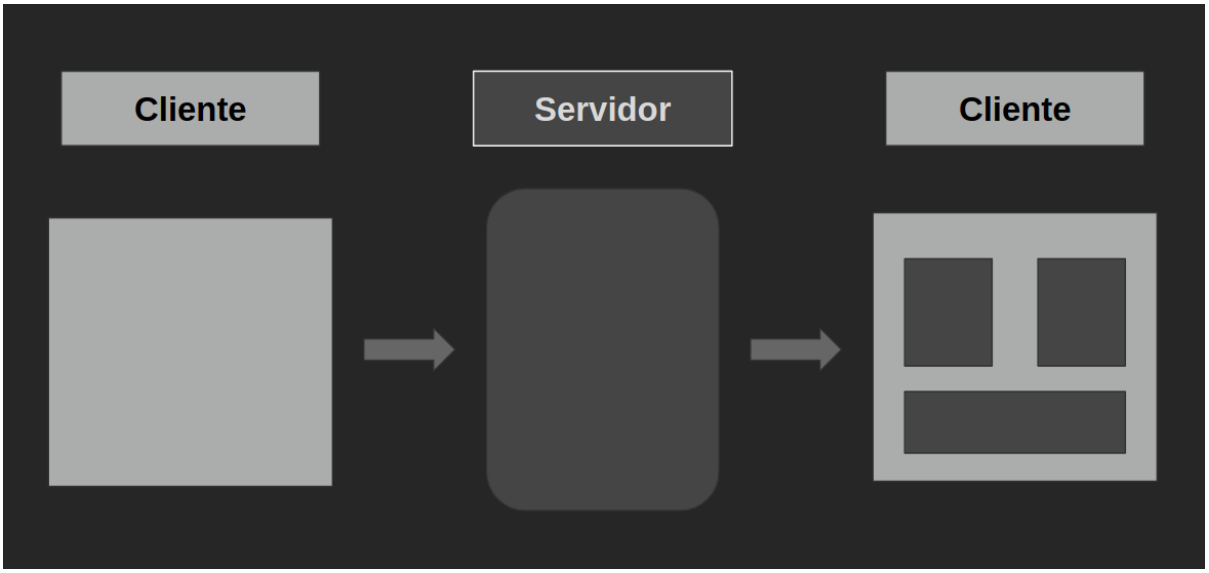
Muchos sitios de los que visitamos usan AJAX y es posible que no nos hayamos dado cuenta de ello. Por ejemplo, cuando añadimos un producto a una cesta de la compra, podemos ver como se actualiza el número de productos que aparecen en la cesta. Otro ejemplo que vemos muy a menudo, es cuando escribimos en una barra de búsqueda y esta nos empieza a mostrar las posibles búsquedas que podemos querer hacer. Otro más puede ser cuando añadimos en alguna red social una nueva publicación.

AJAX permite al usuario realizar otras acciones mientras el navegador está esperando a que los datos se carguen, y esto hace mejor la experiencia de usuario.

Cuando el navegador se encuentra con una etiqueta `<script>`, parará de procesar el resto de la página hasta que ese script haya terminado. Cuando una página que se está cargando necesita cargar datos del servidor, no deberíamos de esperar hasta obtenerlos porque no sabemos cuanto tiempo puede llevar, por lo que usando AJAX, podremos pedir esos datos, mientras el servidor los prepara y nos los envía, el navegador seguirá cargando lo necesario. Cuando el servidor nos envie los datos que hemos pedido, se disparará un evento que llamará a la función encargada de procesar esos datos. Un ejemplo sería cuando entramos en un ecommerce, y nos aparecen todos los productos.

Una vez cargada la página, el usuario puede realizar acciones que requieran actualizar los datos de la interfaz. Aquí ocurre lo mismo que hemos visto antes. El navegador enviará la petición al servidor, y mientras el servidor prepara los datos, el usuario podrá seguir haciendo cosas en la página. Una vez que el servidor nos envía los datos, se emitirá el evento y se ejecutará la función asignada a ese evento para que muestre los datos. Un ejemplo sería cuando rellenamos un formulario, y enviamos los datos para guardarlos.

AJAX es el acrónimo de *Asynchronous JavaScript and XML* que eran las tecnologías usadas para poder realizar estas operaciones asíncronas de las que hemos hablado. Pero con el tiempo, las tecnologías han ido cambiando y ahora a lo que se refiere el termino de AJAX, es al conjunto de las tecnologías que permiten trabajar con operaciones asíncronas en el navegador.



18.1. XMLHttpRequest

Todas las aplicaciones AJAX tienen que usar el objeto XMLHttpRequest que es el que nos va a permitir realizar la comunicación con el servidor, en segundo plano y sin recargar la página.

A continuación se muestran los métodos y propiedades de XMLHttpRequest necesarios para realizar estas peticiones:

Table 33. Propiedades de XMLHttpRequest

Propiedad	**Descripción**
readyState	Devuelve el estado de XMLHttpRequest: 0 (no inicializado), 1 (conexión con el servidor establecida), 2 (petición recibida), 3 (procesando la petición), 4 (petición finalizada)
responseText	Devuelve el contenido de la respuesta del servidor en forma de texto
status	Devuelve el estado de la petición HTTP: 200, 403, 404...
statusText	Devuelve el mensaje del estado de la petición HTTP: *Ok*, *Forbidden*, *Not found*...

Table 34. Métodos de XMLHttpRequest

Método	**Descripción**
open(metoto, url)	Inicializa los parámetros necesarios para crear la petición. Estos parámetros son el **método** (*GET*, *POST*...) y la **url**.
send(datos)	Realiza la petición HTTP al servidor. En caso de la petición *POST*, se envian los datos como parámetro

Método	Descripción
onreadystatechange	Cada vez que haya algún cambio en las propiedades del estado (readyState) de la petición, ejecuta la función que se le asigna
abort()	Detiene la petición actual

Los pasos a seguir para realizar una petición son los siguientes:

- Creamos el objeto XMLHttpRequest
- Inicializar los parámetros de la petición usando el método `open()`
- Definir la función que se tiene que ejecutar cuando recibamos la respuesta, usando `onreadystatechange`
- Enviar la petición usando el método `send()`

```
function cargarContenido() {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', 'datos.json');

  xhr.onreadystatechange = function() {
    if (xhr.status == 200 && xhr.readyState == 4) {
      var datos = JSON.parse(xhr.responseText);
      console.log(datos);
    }
  }

  xhr.send();
}
```

Chapter 19. Ecmascript 6

En este tema veremos las novedades que llegaron con la versión 6 del estandar de JavaScript.

19.1. Let y Const

En la nueva versión de JavaScript (ES6), se han introducido dos nuevas formas de declarar las variables.

- **let**
- **const**

19.1.1. Let

La palabra **let** es muy parecida a la palabra **var**, las dos nos permiten declarar variables locales. La principal diferencia entre ellas es el alcance de las variables que vamos a declarar.

Las variables declaradas usando *let* solo van a poder ser accesibles dentro del bloque donde se han declarado, mientras que las que se declaran con *var* son accesibles dentro de la función en la que se han declarado.

```
function getValor(param1) {
  let res1 = 0;
  var res2 = 0;
  if (true) {
    let res1 = 1;
    var res2 = 1;
    console.log('LET: ' + res1); // 1
    console.log('VAR: ' + res2); // 1
  }
  console.log('LET: ' + res1); // 0
  console.log('VAR: ' + res2); // 1
  return res1 + res2;
}

getValor()
```

19.1.2. Const

La declaración de variables usando la palabra **const** actua igual que cuando usamos **let**, es decir, que solo son accesibles desde dentro del bloque en el que se han declarado, pero con la diferencia de que va a ser una *constante*, por lo que no se puede modificar el valor de esta variable.

```
const NUM_MAX = 5;
```

Al ser una constante, si intentamos asignarle un nuevo valor, nos va a dar un error.

```
const NUM_MAX = 5;
NUM_MAX = 4;
```

19.2. Bucle: FOR OF

Este bucle es como el bucle `for in` que hemos visto anteriormente, solo que esta vez en lugar de guardar en la variable la *posición* o la *clave*, se guarda el valor del elemento.

```
for (var k of [1, 2, 3, 4]) {
  console.log('Elemento del array en esta iteración' + k);
}
```

19.3. Template literals

Los **templates literals** son una nueva forma de crear *strings*. Para usarlos hay que añadir el texto que queremos crear entre las comillas ``.

```
var texto = `Si eres bueno en algo, nunca lo hagas gratis`;
```

El principal problema a la hora de crear strings a los que les queremos asignar valores que nos devuelve alguna expresión, es que tenemos que ir concatenandolos junto al texto que queremos mostrar, y esto puede darnos bastante trabajo si tenemos que mostrar bastantes valores almacenados en variables.

Con los *template literals* vamos a evitar concatenar todo esto. Para ello, donde queramos mostrar una variable o el resultado de una expresión, vamos a añadir la expresión o la variable entre `${}`.

```
var cuenta = '2+2';
var resultado = `El resultado de ${cuenta} es ${2+2}`;
```

Por último, si queremos añadir saltos de línea, ahora lo podemos hacer de una forma mucho más sencilla que antes. Solo hay que saltar de línea dentro del *template literal*, en lugar de añadir `\n` como haríamos usando el método antiguo.

```
var textoMultiLinea = `Este texto aparece
en varias
líneas`;
```

19.4. Arrow functions (funciones flecha)

Las **arrow functions** que a veces se les llama funciones lambda, son funciones anónimas que se usan muy amenudo en JavaScript.

Hay muchos métodos que reciben como parámetro una función anónima, y es muy posible que esta función anónima tenga solo una linea de código. Para esta sola linea de código estariamos usando la palabra reservada `function`, habria que abrir llaves `{}` y en caso de devolver un valor usar la palabra reservada `return`. Las arrow functions nos ahorran el tener que escribir todo ese código para solo una linea que va a contener en el cuerpo, y se llaman arrow functions porque el cuerpo y los parámetros van separados por una flecha (⇒).

```
let misPeliculas = [
  { titulo: 'Scary movie', genero: 'comedia' },
  { titulo: 'La jungla de cristal', genero: 'accion' },
  { titulo: 'Los mercenarios', genero: 'accion' },
  { titulo: 'Salvar al soldado Ryan', genero: 'belica' }
];

let peliculasComedia = misPeliculas.filter(function (pelicula) {
  return pelicula.genero === 'comedia';
});

let peliculasAccion = misPeliculas.filter(pelicula => pelicula.genero === 'accion');
```

Cuando estas funciones no reciben parámetros o reciben más de uno, entonces hay que ponerlos entre parentesis, mientras que si reciben solo uno, da igual si ponemos los parentesis o no. Y en caso de tener más de una linea de código, el cuerpo de la función tiene que ir entre llaves.

```
misPeliculas.forEach(() => console.log('Vista!'));
misPeliculas.forEach(pelicula => console.log(pelicula.titulo + ' vista!'));
misPeliculas.forEach((pelicula, index) => console.log(pelicula.titulo + ' (' + pelicula.genero + ')' + ' vista!'));
misPeliculas.forEach((pelicula, index) => {
  let texto = pelicula.titulo + ' (' + pelicula.genero + ')' + ' vista!';
  console.log(texto);
});
```

Normalmente cuando necesitamos usar la variable **this** dentro de un *callback*, tenemos que asignarle su valor a otra variable que se suele llamar **self**. Con las arrow functions no hace falta capturar la variable **this** sino que te deja usarla directamente.

```
function pelicula() {
  let self = this;
  self.añoEstreno = 2000;
  setTimeout(function () {
    console.log(self.añoEstreno);
  }, 1500);
}
pelicula();

function peliculaArrow() {
  this.añoEstreno = 2000;
  setTimeout(() => {
    console.log(this.añoEstreno);
```

```
  }, 1500);
}
peliculaArrow();
```

19.5. Rest params

Los parámetros **rest**, se usan para recoger un conjunto de valores en un array. Este tipo de parámetros tienen que ir en la última posición cuando se definen los parámetros en las funciones.

```
function getNumeroLoteria(...nums) {
  return nums.join(', ');
}

let num = getNumeroLoteria(1, 5, 12, 22, 35, 37);
console.log(num);
```

19.6. Spread operators

El operador **spread** lo que nos permite es pasarle un array de valores a una función, y esta va a separarlos en distintos valores. Se usa igual que el anterior, al pasar el array a la función se le ponen los ··· delante del nombre de la variable, y esa función recibirá los elementos del array como valores independientes.

```
let numeros = [1, 3, 6, 2, 8, 0, 2];

let max = Math.max(...numeros);
console.log(max);
```

19.7. Destructuring Arrays/Objects

Cuando tenemos un array o un objeto con datos, y necesitamos sacar estos datos en variables independientes, teniamos que hacerlo de una en una.

```
let colores = ['Negro', 'Blanco'];
let negro = colores[0];
let blanco = colores[1];

let datosUsuario = {
  username: 'falco',
  password: 1234
};
let username = datosUsuario.username;
let password = datosUsuario.password;
```

Pero ahora ya podemos desestructurar un array u objeto para sacar los valores de una vez sin necesidad de escribir tanto código. Para ello solo tenemos que envolver las variables que van a recibir los valores entre los simbolos que representan al objeto del que van a salir estos valore, e igualarle el objeto u array.

En el caso de los objetos, las variables se tienen que llamar igual que las propiedades del objeto.

```
let colores = ['Negro', 'Blanco'];
let [negro, blanco] = colores;

let datosUsuario = {
  username: 'falco',
  password: 1234
};
let {username, password} = datosUsuario;
```

También podemos ignorar algún elemento al desestructurar un array, si no ponemos ninguna variable.

```
let colores = ['Negro', 'Blanco'];
let [, blanco] = colores;
```

Incluso podriamos obtener algunos elementos, y los elementos restantes guardarlos en una variable.

```
let colores = ['Negro', 'Blanco', 'Gris'];
let [negro, ...otros] = colores;
```

19.8. Clases

Hasta ahora en JavaScript las clases no se usaban como se hace en otros lenguajes porque no existían como tal, sino que se simulaban clases mediante las funciones.

Las clases son plantillas que nos van a permitir crear objetos. Todos los objetos que se crean con las clases van a tener las mismas propiedades y los mismos métodos. Las clases encapsulan una funcionalidad que se puede reutilizar y que se relaciona con una entidad (cualquier cosa).

```
class Serie {
  constructor(nombre, episodios, temporadas, episodiosVistos) {
    this.nombre = nombre;
    this.episodios = episodios;
    this.temporadas = temporadas;
    this.episodiosVistos = episodiosVistos;
  }
  episodiosPorVer() {
    return this.episodios - this.episodiosVistos;
```

```
  }
}

let vikings = new Serie('Vikings', 69, 5, 45);
console.log(vikings.episodiosPorVer());
console.log(vikings.finalizada());
```

Podemos añadir **métodos estáticos** añadiendo la palabra `static` delante del método. Y ahora no necesitamos una instancia del objeto para poder usar este método.

```
class Serie {
  constructor(nombre, episodios, temporadas, episodiosVistos) {
    this.nombre = nombre;
    this.episodios = episodios;
    this.temporadas = temporadas;
    this.episodiosVistos = episodiosVistos;
  }
  episodiosPorVer() {
    return this.episodios - this.episodiosVistos;
  }
  finalizada() {
    return this.episodiosPorVer() == 0;
  }
  static queEs() {
    return 'Serie';
  }
}

let vikings = new Serie('Vikings', 69, 5, 45);
console.log(Serie.queEs());
```

Ahora podemos hacer que una clase herede de otra de una forma más parecida a lo que estamos acostumbrados a ver en otros lenguajes.

Para indicar que una clase hereda de otra, solo hay que añadir `extends` seguido de la clase de la que va a heredar.

```
class Rectangulo {
  constructor(largo, ancho) {
    this.largo = largo;
    this.ancho = ancho;
  }
  getArea() {
    return this.largo * this.ancho;
  }
}

class Cuadrado extends Rectangulo {
  constructor(largo) {
```

```
    super(largo, largo);
  }
}

let cuadrado = new Cuadrado(3);
console.log(cuadrado.getArea());
```

Para añadir propiedades privadas, tenemos que declarar estas propiedades en el constructor usando la palabra **var**. Y para poder acceder a los datos o modificarlos, tendremos que crear unos métodos que nos lo permitan.

```
class Persona {
  constructor(nombre, dni) {
    this.nombre = nombre;
    var _dni = dni;
    this.setDni = function (dni) {
      _dni = dni;
    }
    this.getDni = function () {
      return _dni;
    }
  }
}

let p = new Persona('Angel', 1111);
console.log(p.getDni());
p.setDni(22222);
console.log(p.getDni());
```

19.9. Modulos

Los módulos nos permiten agrupar bloques de código y exportarlos para poder usarlos en cualquier lugar de la aplicación cuando queramos.

Todo aquello que hemos puesto en un módulo solo va a poder ser usado en otros módulos si se está exportando, por lo tanto tendremos que indicar, que cosas hay que exportar con la palabra `export`. Hay dos formas de hacerlo.

```
export class Mascota {
  constructor(nombre, tipo, sonido) {
    this.nombre = nombre;
    this.tipo = tipo;
    this.sonido = sonido;
  }
  hazSonido () {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
```

```
}
```

```
class Mascota {
  constructor(nombre, tipo, sonido) {
    this.nombre = nombre;
    this.tipo = tipo;
    this.sonido = sonido;
  }
  hazSonido () {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}

export { Mascota };
```

Y para poder usar aquello que se está exportando desde un módulo, tendremos que importarlo en el archivo donde lo necesitemos usando la palabra `import` seguida de lo que se quiere importar y indicando desde que archivo hay que importarlo. A la hora de importarlo le podemos poner un alias a lo que se importa.

```
import { Mascota } from './mascota';

let perro = new Mascota('Toby', 'perro', 'guau');
perro.hazSonido();
```

Hay otra forma de importar módulos y es usando el asterisco, que importará todo aquello que se está exportando. A la hora de usar esta forma tendremos que darle un alias para poder acceder a las cosas que se están importando.

```
import * as masc from './mascota';

let perro = new masc.Mascota('Toby', 'perro', 'guau');
perro.hazSonido();
```

19.9.1. Exportando por defecto

Hay otra opción para exportar a parte de las vistas antes, y es la exportación por defecto que se usa cuando vamos a exportar solo una cosa del módulo. Aquí no hace falta ponerle nombre a lo que estamos exportando.

```
export default class {
  constructor(nombre, tipo, sonido) {
    this.nombre = nombre;
    this.tipo = tipo;
    this.sonido = sonido;
```

```
  }
  hazSonido() {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}
```

```
import Mascota from './mascota';

let perro = new Mascota('Toby', 'perro', 'guau');
perro.hazSonido();
```

19.10. Lab: algoritmo de enfrentamientos

En este laboratorio vamos a crear un algoritmo que tiene que cumplir con lo siguiente:

- Dada una lista de equipos, mostrar los enfrentamientos entre ellos (1vs1) de forma aleatoria.
- Si el número de equipos no es par, entonces el que sobra pasa directamente a la siguiente fase.
- Usar algunas de las novedades de ES6 como:
  - Desestructuración de arrays
  - Spread Operator

Ejemplo 1:

```
Equipos -> ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis']
Resultado posible ->
        Sevilla vs Getafe
        Valencia vs At Bilbao
        Betis vs Villarreal
        At Madrid vs Real Madrid
        Espanyol vs Barcelona
```

Ejemplo 2:

```
Equipos -> ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis', 'Girona']

Resultado posible ->
        Betis vs Sevilla
        Valencia vs Villarreal
        Real Madrid vs Espanyol
        At Madrid vs Girona
        At Bilbao vs Barcelona
        Getafe pasa de ronda
```

19.10.1. Aleatorizar equipos

Vamos a empezar por coger el array de equipos que nos pasan y vamos a cambiar el orden de estos

de forma aleatoria.

Empezamos por añadir los equipos y crear una función que va a recibir el array de equipos y va a devolver un nuevo array de equipos pero ordenados de forma aleatoria.

/enfrentamientos.js

```
let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis']
// let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis', 'Girona']

function shuffle(listaEquipos) {

}
```

Dentro de la función, vamos a recorrer el array empezando por el final, y vamos a ir generando un número aleatorio en cada iteración. Este número nos sirve para ir sacando del array el equipo que se encuentra en dicha posición para añadirlo al array que se va a devolver.

Si recorrieramos el array desde el principio, se va a dar el caso de que al sacar los equipos del array inicial para que no se puedan repetir en el nuevo, la longitud del array va a ir disminuyendo provocando que la condición del for se incumpla antes de sacar todos los equipos del array.

/enfrentamientos.js

```
let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis']
// let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis', 'Girona']

function shuffle(listaEquipos) {
  const nuevaListaEquipos = []

  for (let pos = listaEquipos.length; pos > 0; pos--) {
    let randomPos = Math.floor(Math.random() * pos)
    let [ num ] = listaEquipos.splice(randomPos, 1)
    nuevaListaEquipos.push(num)
  }

  return nuevaListaEquipos
}
```

Una vez que tenemos la función, si la probamos deberíamos de obtener una lista de equipos con un orden distinto al inicial.

/enfrentamientos.js

```
let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis']
// let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis', 'Girona']

function shuffle(listaEquipos) {
  const nuevaListaEquipos = []
```

```
  for (let pos = listaEquipos.length; pos > 0; pos--) {
    let randomPos = Math.floor(Math.random() * pos)
    let [ num ] = listaEquipos.splice(randomPos, 1)
    nuevaListaEquipos.push(num)
  }

  return nuevaListaEquipos
}

console.log(shuffle(equipos));
```

```
[ 'Espanyol',
  'At Madrid',
  'At Bilbao',
  'Villarreal',
  'Barcelona',
  'Real Madrid',
  'Sevilla',
  'Valencia',
  'Betis',
  'Getafe' ]
```

19.10.2. Mostrar enfrentamientos

Ahora que ya hemos obtenido un array de equipos que se han ordenado de forma aleatoria, vamos a crear otra función que se va a encargar de ir mostrando estos equipos de dos en dos.

Esta función va a ser una función recursiva que va a recibir un array con los equipos, del cual obtendremos los dos primeros que se van a enfrentar, y después cogeremos el resto de equipos y volveremos a llamar a esta función hasta que lleguemos a la condición base que será que haya menos de 2 equipos en el array.

Para obtener los dos equipos que se van a enfrentar y dejar el resto en otro array, usaremos la desestructuración del array de equipos que se reciben como parámetro.

/enfrentamientos.js

```
let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis']
// let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis', 'Girona']

function shuffle(listaEquipos) {
  const nuevaListaEquipos = []

  for (let pos = listaEquipos.length; pos > 0; pos--) {
    let randomPos = Math.floor(Math.random() * pos)
    let [ num ] = listaEquipos.splice(randomPos, 1)
    nuevaListaEquipos.push(num)
  }

  return nuevaListaEquipos
}

function muestraEnfrentamientos(equipos) {
```

```
  const numEquipos = equipos.length;
  if (numEquipos >= 2) {
    let [equipo1, equipo2, ..._] = equipos;
    console.log(equipo1 + ' vs ' + equipo2);
    muestraEnfrentamientos(_);
  } else {
    console.log('No quedan más enfrentamientos');
  }
}

muestraEnfrentamientos(shuffle(equipos));
```

Si probamos a ejecutar nuestro algoritmo, ya debería de mostrarnos los enfrentamientos entre los equipos, como aparece a continuación.

```
Barcelona vs Villarreal
Valencia vs At Bilbao
Real Madrid vs Sevilla
Espanyol vs Betis
Getafe vs At Madrid
No quedan más enfrentamientos
```

Una vez que tenemos la parte de enfrentamientos con un número de equipos pares, vamos a añadir una condición en el caso base, para mostrar que si por último queda un solo equipo, se muestre que dicho equipo pasa a la siguiente ronda directamente.

/enfrentamientos.js

```
// let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis']
let equipos = ['Valencia', 'Sevilla', 'Real Madrid', 'Barcelona', 'At Madrid', 'At Bilbao', 'Villarreal', 'Getafe',
'Espanyol', 'Betis', 'Girona']

function shuffle(listaEquipos) {
  const nuevaListaEquipos = []

  for (let pos = listaEquipos.length; pos > 0; pos--) {
    let randomPos = Math.floor(Math.random() * pos)
    let [ num ] = listaEquipos.splice(randomPos, 1)
    nuevaListaEquipos.push(num)
  }

  return nuevaListaEquipos
}

function muestraEnfrentamientos(equipos) {
  const numEquipos = equipos.length;
  if (numEquipos >= 2) {
    let [equipo1, equipo2, ..._] = equipos;
    console.log(equipo1 + ' vs ' + equipo2);
    muestraEnfrentamientos(_);
  } else {
    if (numEquipos === 1) {
      let [equipo] = equipos;
      console.log(equipo + ' pasa a la siguiente fase directamente.');
    }
    console.log('No quedan más enfrentamientos');
  }
}
```

```
muestraEnfrentamientos(shuffle(equipos));
```

Y con este último, cambio si ejecutamos el algoritmo con un array de equipos impares nos mostrará los siguientes enfrentamientos.

```
Girona vs At Madrid
Espanyol vs Valencia
Sevilla vs Real Madrid
Betis vs Villarreal
Barcelona vs Getafe
At Bilbao pasa a la siguiente fase directamente.
No quedan más enfrentamientos
```

Chapter 20. Promesas

Las **promesas** son objetos que se usan en las operaciones asíncronas, por lo que no se sabe si vamos a obtener el resultado de la operación ahora, en el futuro o nunca.

Surgen sobre todo para mejorar la legibilidad de nuestro código y evitar tener que pasar el contenido de las funciones directamente como argumentos a nuestra llamada (el **callback hell**).

Las promesas solo pueden devolver una respuesta y no se puede cancelar la petición realizada.

Las promesas tienen 3 estados:

- **Pendiente**: es el estado inicial, ni se ha cumplido, ni se ha rechazado.
- **Cumplida**: indica que la operación se ha completado correctamente.
- **Rechazada**: indica que la operación ha fallado.

Las promesas tienen dos métodos importantes:

- `resolve(valor)`: devuelve una promesa con el valor dado, e indica que se ha resuelto esa promesa. Se ejecutará la primera función que recibe la promesa.
- `reject(razon)`: devuelve una promesa con la razón por la cual se ha rechazado la promesa. Se ejecutará la segunda función que recibe la promesa.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script src="main.js"></script>
</head>
<body>
  <div id="contenido"></div>
  <div>
    <label for="num">Número loteria:</label>
    <input type="text" id="num" maxlength="5">
  </div>
  <button type="button" id="btn-comprobar">Comprobar</button>
</body>
</html>
```

```
window.onload = function () {
  var BASE_URL = 'https://ejemplos-dc1c1.firebaseio.com/';

  function getPrimerNum(num) {
    var promise = new Promise(function (resolve, reject) {
      if (!num) {
        reject('No se ha introducido ningún número');
```

```
      }
      var url = BASE_URL + 'loteria-navidad/primero.json';
      var xhr = new XMLHttpRequest();
      xhr.open('GET', url);
      xhr.onreadystatechange = function () {
        if (xhr.status == 200 && xhr.readyState == 4) {
          var primero = JSON.parse(xhr.responseText);
          if (primero == num) {
            resolve(true);
          } else {
            resolve(false);
          }
        }
      }
      xhr.send();
    });

    return promise;
  }

  function mostrarMensaje(mensaje) {
    var contenido = document.getElementById('contenido');
    contenido.textContent = mensaje + '\n';
  }

  var btn = document.getElementById('btn-comprobar');
  btn.addEventListener('click', () => {
    var num = document.getElementById('num').value;
    getPrimerNum(num)
    .then(
      (premiado) => {
        if (premiado) {
          mostrarMensaje('Enhorabuena, has ganado el primer premio de la loteria :)');
        } else {
          mostrarMensaje('Prueba otro año');
        }
      },
      (error) => mostrarMensaje(error)
    )
  });
}
```

Chapter 21. Async/Await

Async/Await es una forma moderna de trabajar con el código asíncrono en JavaScript. Esta sintaxis hace que el código asíncrono sea más fácil de escribir y leer, ya que parece código síncrono. Pero no es más que azúcar sintáctico, por debajo sigue siendo todo promesas.

Allí donde se va a poner operaciones asíncronas, tenemos que marcarlo con la palabra **async**. Y para indicar que queremos esperar a que la operación asíncrona termine, tenemos que utilizar la palabra **await**.

```
async function obtenerDatosDeAPI() {
    const resp = await fetch('http://localhost:3000/datos');
    const datos = await resp.json()

    return datos;
}
```

Tenemos que tener en cuenta que:

- Siempre que ejecutemos una función **async**, esta va a devolver una promesa.
- El código que va detrás de un **await** solo se ejecuta cuando la promesa se resuelve.
- Los errores con esta nueva sintaxis se capturan con **try/catch**.
- La palabra **await** solo puede usarse dentro de funciones **async**.

| Por ahora, los **await** siempre tienen que ir dentro de funciones **async**, sino nos dará un error. Pero en las futuras versiones de Node esto no será necesario y se podrán usar los **await** en el top level.