

# DESARROLLO DE APLICACIONES WEB EN ENTORNO CLIENTE

GRADO SUPERIOR FP  
DESARROLLO DE APLICACIONES WEB

2/10/2023

JOSE MANUEL MARTINEZ ROCA

## Tabla de contenido

INTRODUCCIÓN.....	7
CONFIGURANDO EL ENTORNO .....	7
CONCEPTOS BÁSICOS.....	8
HERRAMIENTAS .....	8
¿DÓNDE INSERTAMOS EL CÓDIGO JAVASCRIPT? .....	8
VARIABLES EN JAVASCRIPT .....	11
ESTRUCTURA BÁSICA APLICACIÓN JAVASCRIPT .....	13
DOS EJEMPLOS SENCILLOS.....	15
LA WEB MÁS SENCILLA .....	15
WEB NORMALIZADA .....	17
FUNDAMENTOS JAVASCRIPT .....	21
HISTORIA DE JAVASCRIPT.....	21
CARACTERÍSTICAS DE JAVASCRIPT .....	22
POO .....	23
DOM .....	24
BASADO EN EVENTOS .....	24
ASINCRONISMO .....	25
VARIABLES.....	26
VAR, LET Y CONST .....	26
FUNCIONES .....	27
Funciones anónimas .....	29
Funciones flecha .....	29
ESTRUCTURAS Y BLOQUES.....	30
CONDICIONAL IF.....	30
CONDICIONAL SWITCH.....	32
BUCLE WHILE .....	32
BUCLE FOR .....	32
BUCLE .....	33
TIPOS DE DATOS BÁSICOS.....	33
CASTING DE VARIABLES .....	33
NUMBER.....	33
STRING .....	33
BOOLEAN.....	33
MANEJO DE ERRORES .....	33
BUENAS PRACTICAS .....	34

use strict.....	34
Variables.....	34
Errores.....	34
OBJETOS .....	34
INTRODUCCION.....	34
PROPIEDADES.....	35
METODOS.....	36
ARRAYS.....	36
Arrays de objetos .....	37
Operaciones con Arrays .....	38
length .....	38
Añadir elementos.....	38
Eliminar elementos .....	38
splice.....	39
slice .....	39
Arrays y Strings .....	40
sort.....	41
Otros métodos comunes .....	42
Functional Programing.....	42
filter .....	43
find.....	44
findIndex .....	45
every / some.....	45
map.....	46
reduce .....	46
forEach .....	47
includes .....	48
Array.from.....	48
Referencia vs Copia.....	49
Rest y Spread.....	51
Desestructuración de arrays .....	52
Map .....	53
Set .....	53
Programación orientada a Objetos en Javascript .....	54
Introducción.....	54

Herencia .....	55
Métodos estáticos.....	55
Método toString() .....	56
Método valueOf().....	58
Organizar el código .....	59
Ojo con this .....	59
PROMESAS .....	60
¿Qué es una promesa en JavaScript? .....	60
Métodos de las promesas en JavaScript.....	60
Estados de las promesas en JavaScript.....	61
Cómo se crea una promesa en JavaScript .....	61
Cómo implementar las promesas en JavaScript: más ejemplos.....	62
Ejemplo de transacción de pago en línea .....	62
Ejemplo de base de datos .....	63
DOCUMENT OBJECT MODEL (DOM) .....	64
Introducción.....	65
Acceso a los nodos .....	67
Acceso a nodos a partir de otros .....	69
Propiedades de un nodo .....	70
Manipular el árbol DOM .....	71
Modificar el DOM con ChildNode .....	74
Atributos de los nodos .....	75
Estilos de los nodos.....	75
Atributos de clase .....	76
EVENTOS .....	77
Introducción.....	77
Cómo escuchar un evento .....	77
Event listeners.....	79
Tipos de eventos .....	80
Eventos de página.....	80
Eventos de ratón.....	80
Eventos de teclado.....	81
Eventos de toque .....	81
Eventos de formulario.....	81
Los objetos this y event .....	81
Bindeo del objeto this .....	83

Propagación de eventos (bubbling) .....	84
innerHTML y escuchadores de eventos .....	86
Eventos personalizados.....	86
WEB API.....	87
API LocalStorage .....	89
A tener en cuenta.....	90
Storage vs cookies.....	90
Cookies.....	91
REACT .....	92



## INTRODUCCIÓN

Vamos a estudiar los fundamentos del lenguaje interpretado JavaScript, pero, antes de ponernos de lleno con esta tarea, es necesario:

- Definir un entorno de partida en el ordenador que vayamos a usar para ello.
- Sentar algunos conceptos básicos sobre los que se asienta este lenguaje de desarrollo web.
- Aprender la estructura básica de cualquier aplicación escrita en JavaScript.
- Establecer un conjunto de convenciones y buenas prácticas a la hora de manejarnos con el lenguaje JavaScript y con el ecosistema que hemos creado para tal fin.

## CONFIGURANDO EL ENTORNO

Para el seguimiento de este aprendizaje vamos a usar como IDE de programación [Visual Studio Code](#) y, además, usaremos unas cuantas extensiones que nos van a facilitar la labor de programar y de testear la web que vamos desarrollando.

Las extensiones que vamos a usar son:

- Live Server
- Material Icon Theme
- Prettier – Code Formatter
- SonarLint

La extensión [Live Server](#) nos provee de un servidor virtual que es capaz de mostrar páginas dinámicas que usen tecnología JavaScript.

La extensión [Material Icon Theme](#) es una mejora visual para VS Code que nos permite ver los iconos de cada uno de los archivos que componen nuestros proyectos ya que es capaz de reconocer que contiene cada uno de esos ficheros y así nos muestra el icono correspondiente para cada tipo de fichero incluido en nuestros proyectos.

La extensión [Prettier](#) es una extensión, igual que la anterior, para mejora visual del código que nos colorea las distintas estructuras, tal como, condicionales IF, bucles de tipo WHILE, FOR, etc. Solamente para que nos sea más fácil leer e interpretar ese código que estamos construyendo.

La extensión SonarLint se encarga de mostrar los fallos de programación que vamos cometiendo al escribir nuestros proyectos. Y no solamente de mostrar el fallo, si no, también de darnos el lugar de nuestro código donde se ha cometido el error y posibles explicaciones sobre el mismo.

Para acabar de configurar el ecosistema de programación que estamos montando debemos acudir a la opción “Auto guardado” o “Auto Save” que se encuentra dentro del menú “Archivo” o “File”, dependiendo del idioma que tengamos configurado en VS Code. Y marcar dicha opción para que el IDE VS Code se encargue de guardar automáticamente todo el proyecto cada vez que realicemos un cambio en dicho proyecto.

## CONCEPTOS BÁSICOS

### HERRAMIENTAS

Para aprender JavaScript lo más práctico es empezar a utilizar dicho lenguaje desde el primer momento, y para ello, podemos hacer uso de:

- Un IDE especializado, en nuestro caso, VS Code.
- Ejecutar el código desde la consola del navegador que usemos.
- Ejecutar el código desde un editor de código online, como puede ser: [jsfiddle.net](https://jsfiddle.net)

Ya veremos por qué se usa cada una de estas opciones para ejecutar código, pero, se puede adelantar que según lo que queramos conseguir al ejecutar código será más conveniente usar una herramienta u otra para dicho fin. Por ejemplo, si queremos ver el valor de alguna variable es común poner una orden “console.log” en el código de nuestro proyecto para espiar y conocer dicho valor. Pero, igual, nos es más cómodo usar la consola del navegador, que ir a modificar un fichero y después volver a la ejecución de la aplicación web a ver el resultado.

Por otro lado, pensad en una aplicación enorme, unas 10.000 líneas de código aproximadamente o similar, sería una barbaridad intentar escribir esa aplicación web en una consola e incluso en un editor de código online, ya que este último no da facilidades para manejar más allá de unos cuatro o cinco ficheros y no está preparado para integrar tremendo tamaño de código, para este cometido lo más lógico sería usar el IDE VS Code que si nos da esas facilidades.

Como vemos, en cada uno de los casos presentados, debemos valorar y saber hacer uso de la herramienta más adecuada para cada uno de los fines que pretendemos alcanzar, la herramienta más eficiente, rápida y cómoda para nuestro mejor aprovechamiento tanto del tiempo como de la eficacia a la hora de escribir código.

## ¿DÓNDE INSERTAMOS EL CÓDIGO JAVASCRIPT?

Este código JavaScript se puede enlazar con la web HTML de diversas formas:



- Directamente entremezclando etiquetas HTML5 con código JavaScript mediante la etiqueta que rodeara siempre al código JavaScript “SCRIPT”.
- En uno o varios ficheros aparte que contendrán exclusivamente el código JavaScript y enlazándolo desde el archivo HTML con la instrucción:

```
<script src="main.js"></script>
```

Al igual que enlazamos el archivo con los códigos JavaScript desde el fichero HTML con la instrucción que acabamos de ver, para que todo resulte armonioso y quede completamente ajustado y bonito para el usuario. Debemos enlazar el fichero CSS, también desde el HTML, mediante la siguiente instrucción.

```
<link rel="stylesheet" href="style.css">
```

Atención que si se nos olvida enlazar estos ficheros y ejecutamos la aplicación web seguramente no funcione y podemos consumir un gran espacio de tiempo en descubrir que es esta falta de enlaces con sus respectivos ficheros la que nos hace que la web falle. No suelta ningún error el navegador ni la web si se ha dejado de enlazar los ficheros, solo que ves que no funciona, pero silenciosamente.

## CONVENCIONES JAVASCRIPT

JavaScript es un lenguaje interpretado que es case sensitive, o, sensible a mayúsculas, esto es fácil de comprender, si entendemos que cualquier variación en el nombre de una variable crea una nueva variable, es decir:

var miVariable, este nombre de variable es distinto de var mivariable. Aunque este último nombre de variable ya veremos un poco más adelante que no es aceptable en JavaScript, aunque si este permitido. También son variables distintas miVariable, miVARIABLE, mIVARIABLE, etc.

El nombre de las variables siempre se escribe con la tipografía camelCase, es decir, cuando el nombre de una variable tiene más de una palabra, la primera letra de la variable siempre es minúscula y el resto de primeras letras de cada palabra que forma el nombre se colocan en mayúsculas. Sin incluir espacios, ni caracteres especiales o ilegales. Son ejemplos correctos de camelCase:

miNombre, miApellidoPrimero, miApellidoSegundo, miEdad.

No serian correctos los siguientes nombres de variables ni ajustados a la nomenclatura camelCase:

MiNombre, mi Nombre, mi\_Nombre, MINOMBRE, minombre.

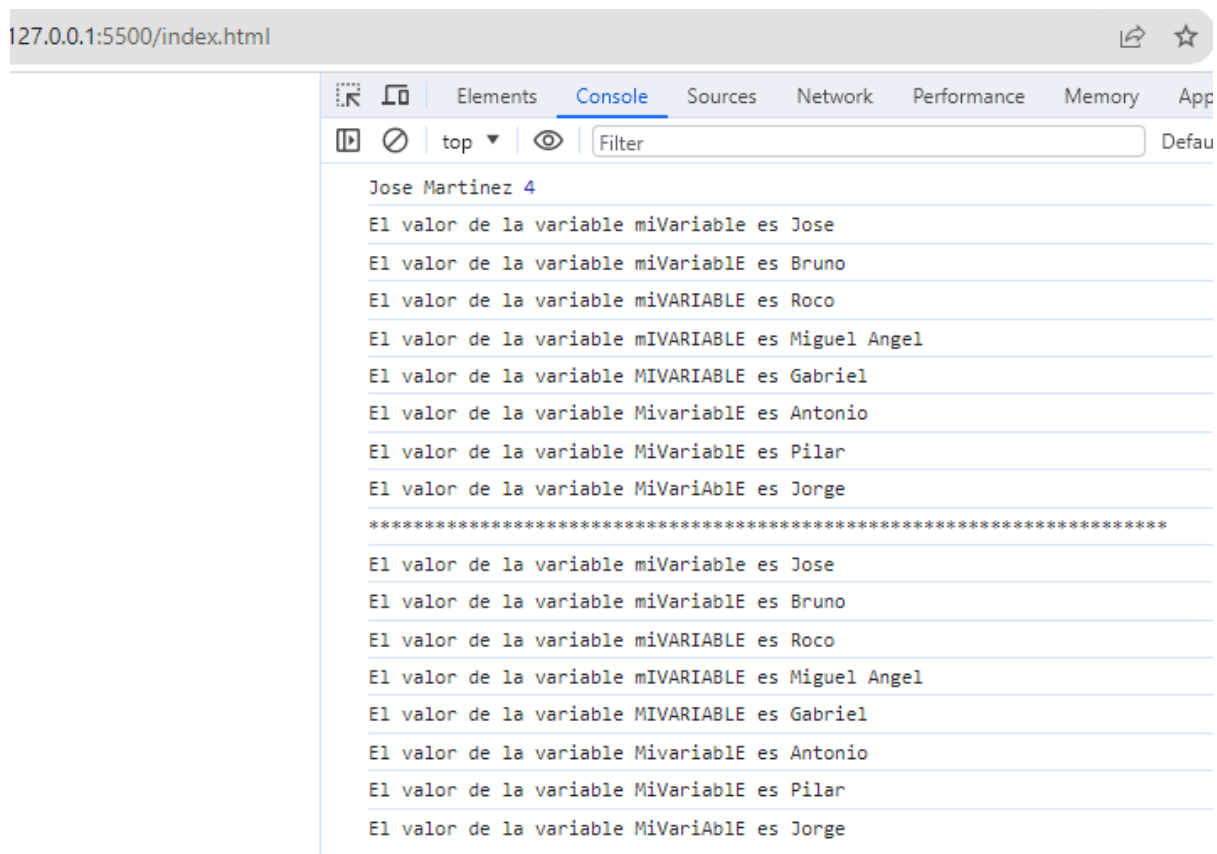
Lo podemos ver en el funcionamiento del siguiente código:

```
var miVariable = "Jose";
var miVariableE = "Bruno";
var miVARIABLE = "Rocco";
var mIVARIABLE = "Miguel Ángel";
var MIVARIABLE = "Gabriel";
var MivariablE = "Antonio";
var MiVariable = "Pilar";
var MiVariAblE = "Jorge";

console.log('El valor de la variable miVariable es ' + miVariable);
console.log('El valor de la variable miVariableE es ' + miVariableE);
console.log('El valor de la variable miVARIABLE es ' + miVARIABLE);
console.log('El valor de la variable mIVARIABLE es ' + mIVARIABLE);
console.log('El valor de la variable MIVARIABLE es ' + MIVARIABLE);
console.log('El valor de la variable MivariablE es ' + MivariablE);
console.log('El valor de la variable MiVariable es ' + MiVariable);
console.log('El valor de la variable MiVariAblE es ' + MiVariAblE);

// Y si las vuelvo a imprimir contiene los mismos datos
console.log("*****");
console.log('El valor de la variable miVariable es ' + miVariable);
console.log('El valor de la variable miVariableE es ' + miVariableE);
console.log('El valor de la variable miVARIABLE es ' + miVARIABLE);
console.log('El valor de la variable mIVARIABLE es ' + mIVARIABLE);
console.log('El valor de la variable MIVARIABLE es ' + MIVARIABLE);
console.log('El valor de la variable MivariablE es ' + MivariablE);
console.log('El valor de la variable MiVariable es ' + MiVariable);
console.log('El valor de la variable MiVariAblE es ' + MiVariAblE);
```

Ya que la salida que provoca este código es la siguiente:



Como vemos, el más mínimo cambio en el nombre de la variable, aunque solamente sea en una mayúscula, genera una nueva variable totalmente independiente.

## VARIABLES EN JAVASCRIPT

JavaScript tiene un tipado débil, no os asustéis por los tecnicismos, son fácilmente comprensibles y estamos aquí para aprenderlos y descubrir que significa cada cosa. Este tipado débil que en un principio puede parecer una ventaja, para mí, es un inconveniente. Ya que esa “libertad” que te ofrece JavaScript de declarar variables sin especificar el tipo de variable que es (literal, numérica, booleana, etc.) y después, cambiar al vuelo la variable, o sea, que declaro una variable, por ejemplo, tipo booleana que se llame apellidos y le asigno el valor de false porque en ese momento estoy pensando que no tiene apellidos conocidos, y después unas cuantas líneas más allá digo que esa variable ahora contiene el valor literal de los apellidos del usuario. Esto es, bajo mi humilde punto de vista, una bomba de relojería. Ya que además de la dificultad de la abstracción que debemos realizar para poder crear código, se añade la dificultad de que JavaScript nos vuelva locos cambiando las variables de tipo en cualquier momento.

De ahí, que se recomienda siempre, no usar variables globales con la orden “var” y se recomienda el uso de variables locales con la orden “let”, se pueden usar cualquiera de las dos, pero como buenas prácticas de programación se recomienda el uso de la orden “let” sobre la orden “var”, seguramente el programa funcionará bien en ambos casos, sobre todo si es una aplicación de pocas líneas de código. Pero es extremadamente aconsejable y hasta exigible que se use “let” sobre “var”

```

// Variables y constantes

const nombre = "Jose";
const apellido = "Martinez";

let valorDado = 5;
valorDado = 4

// let valorDado = 4;

/* Esta última variable está mal declarada y salta un error porque no
se pueden declarar dos variables con el mismo nombre dos veces. Si
descomentamos la segunda línea donde se declara la variable por segunda vez
veremos que el navegador envía un error por consola avisando de ello */

if (true) {
    let nombre2 = nombre + " Manuel";
    //Si quitas el siguiente comentario antes comenta la línea anterior
    //var nombre2 = nombre + " Manuel"; //Prueba a descomentar esta línea a ver que sucede
    // ¿Qué ha ocurrido?
}

//console.log(nombre2); //Quita este comentario para probar el caso anterior

//¿Qué ocurrirá si quito el comentario de la línea siguiente?
//console.log(construirNombreCompleto("Luis", "Alfonso"));

function construirNombreCompleto(nombre1, nombre2){
    return nombre1 + " " + nombre2;
};

console.log(nombre, apellido, valorDado);
// ¿Qué ocurrirá si quito los comentarios de las líneas siguientes?
//console.log(nombre2);
//console.log(nombre1);

```

Con el código anterior podemos ver varios ejemplos de ámbito de variables y la comentada recomendación de usar var lo mínimo posible y el uso de let en su lugar tomaran consistencia práctica.

Quitando comentarios y volviendo a colocarlos podemos hacer que la aplicación funcione o no, según las instrucciones que vemos en el propio código.

En este caso concreto, entre las líneas 1 a 14, están dedicadas a la repetición de variables con el mismo nombre, si quitamos el comentario de la línea nueve y dejamos que se vuelva a declarar una variable con el mismo nombre salta un error en la consola avisándonos de ello.

En el caso de las líneas 16 y 22, están dedicadas al ámbito de una variable según se declare la misma.

En las líneas 27 a 30 además de ver por primera vez como se declara una función en JavaScript y como se usan parámetros de entrada para la misma función, tocamos el asunto espinoso de los ámbitos de las variables dando una vuelta de tuerca más a la dificultad que ello conlleva. Ya que el resultado puede parecer sorprendente a quien no este familiarizado con ámbitos de variables en JavaScript.

Por último, las líneas 33 y 34 van relacionadas también con los ámbitos de las variables implicadas.

## ESTRUCTURA BÁSICA APLICACIÓN JAVASCRIPT

Siempre que durante este curso vayamos a crear una aplicación básica de JavaScript se debe recurrir a la estandarización en dicho funcionamiento, para ello, crearemos los siguientes archivos:

- index.html
- style.css
- main.js

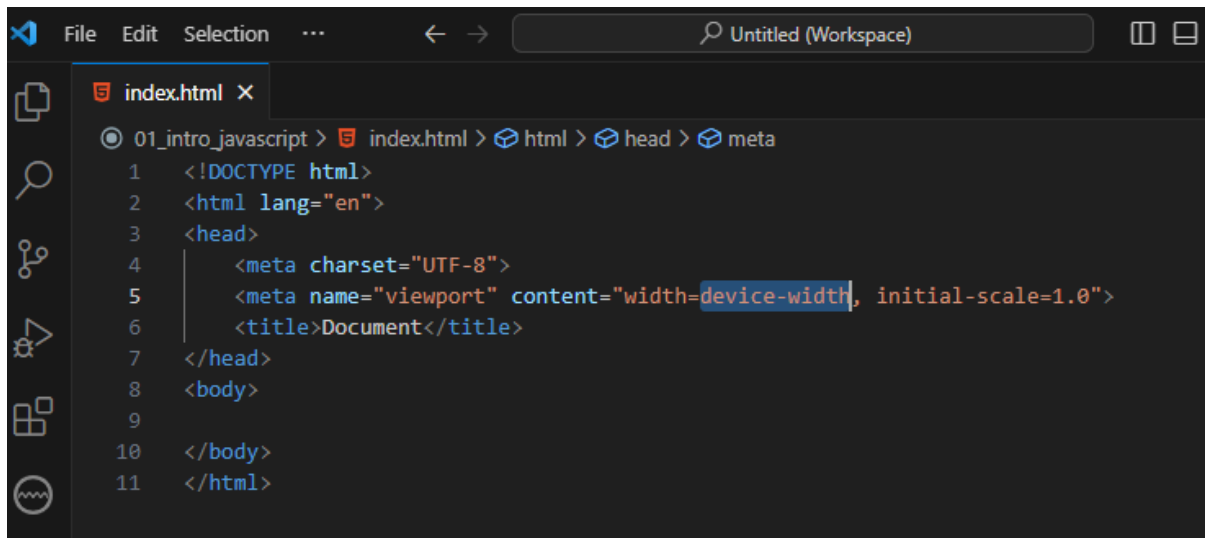
Ojo con las extensiones de los archivos implicados que deben ser exactamente esas y no otras, para que el IDE y el resto de aplicaciones que deban usar dichos ficheros, lo veremos cuando descubramos el framework REACT, puedan ser capaces de funcionar sin fallos, ni problemas adicionales.

Toda esta estructura se complicará posteriormente con el uso del Framework, pero hasta que llegue ese momento haremos la estructura más sencilla posible, sin subcarpetas donde guardar cada uno de los distintos tipos de ficheros que componen cada uno de los proyectos que realicemos. Esto quiere decir que por cada proyecto crearemos una carpeta con su nombre y dentro de esa carpeta se encontraran los tres archivos anteriores que hemos visto.

### **Fichero index.html**

Este fichero contiene toda la estructura e información de la web que vamos a construir, en formato HTML5 y no debe, normalmente, incluir código JavaScript, ni estilos CSS que irán alojados cada uno de ellos en el resto de los ficheros destinados a tal fin.

Si al crear el documento en VS Code, usamos el símbolo “!”, admiración hacia abajo, como primer carácter a incluir en el documento y pulsamos la tecla intro veremos que VS Code nos rellena el documento con el texto básico y mínimo para que ese fichero HTML se distinga como un fichero HTML5, también podemos conseguir esto escribiendo como primeros caracteres en el fichero recién creado de HTML el código “HTML5”, tiene el mismo resultado en ambos casos.



```
File Edit Selection ... ← → Untitled (Workspace)
index.html X
01_intro_javascript > index.html > html > head > meta
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Document</title>
7 </head>
8 <body>
9
10 </body>
11 </html>
```

Ilustración 1: Código básico de un archivo HTML5

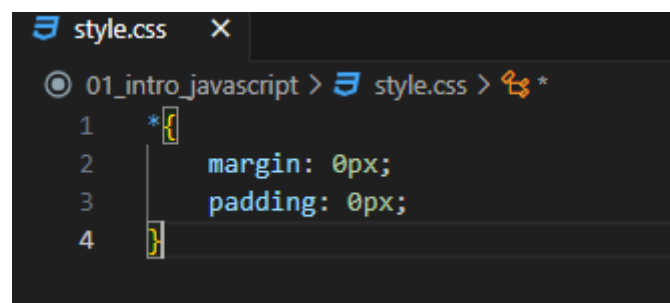
El código resultante de aplicar la técnica explicada se puede ver en la imagen anterior.

Por otro lado, podemos crear un fichero HTML con mucha menos información, o etiquetas HTML, ya que simplemente escribiendo la etiqueta “HTML” en dicho archivo funcionaría igualmente, pero estamos omitiendo mucha información que posteriormente puede ser contraproducente. Así que el archivo HTML de nuestros proyectos, aunque estos proyectos sean ahora muy sencillos, siempre con la estructura básica de html5.

### Fichero style.CSS

Dicho fichero va a albergar todos los estilos del proyecto, es decir, que solamente incluirá las instrucciones CSS para tal fin.

Como curiosidad, para estandarizar las webs, los profesionales del desarrollo web usan la estructura básica siempre al inicio de cualquier CSS ya que los navegadores no alinean todos los elementos igualmente, y para evitar problemas, tienen la costumbre de incluir siempre al inicio de sus CSS:



```
style.css X
01_intro_javascript > style.css > *
1 *{
2   margin: 0px;
3   padding: 0px;
4 }
```

Ilustración 2: código básico CSS.

## Fichero main.js

Este fichero contendrá todos los códigos JavaScript necesarios para que la aplicación web a desarrollar funcione correctamente.

## DOS EJEMPLOS SENCILLOS

Para comprobar y experimentar con todo lo dicho hasta el momento vamos a realizar un par de ejemplos sencillos.

### LA WEB MÁS SENCILLA

Aunque escribamos nuestras aplicaciones desde cero no tiene por qué ser una realización lenta y pesada. Es más, podemos escribir la web más sencilla que existe con poquísimas líneas de código, y, para ello vamos a verlo con un ejemplo:

El nombre de los ficheros, las carpetas o subcarpetas que incluyan son una fuente de problemas si no se siguen una serie de normas. Como las más importantes e inevitables:

- No incluir espacios en los nombres de ficheros o carpetas.
- No incluir caracteres especiales.
- Usar el guion bajo para representar espacios.
- Usar un numero al principio por mantener nuestros proyectos ordenados y saber cual es el orden que seguimos al crearlos.
- No usar mayúsculas y minúsculas en los nombres.

Muchos de estos consejos vienen por que los servidores son incapaces de trabajar con estos caracteres o estos nombres “mal generados”. No es que estén mal escritos simplemente es que si no seguimos estas normas cuando estemos trabajando con servidores en la nube vamos a tener muchísimos problemas a la hora de ejecutar nuestras aplicaciones.

Vamos primeramente a crear una carpeta en la que incluir los ficheros habituales de nuestros proyectos en JavaScript, yo la he llamado 01\_web\_mas\_sencilla, y dentro de ella crearemos los siguientes documentos con el código que se indica a continuación.

Fichero index.html:

```
<html>
  web más sencilla
</html>
```

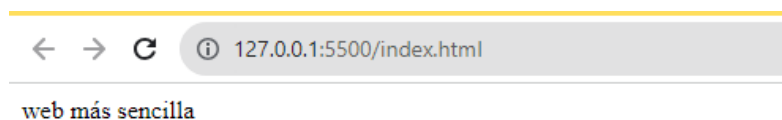
Fichero style.css:

```
*{  
  margin: 0px;  
  padding: 0px;  
}
```

Fichero main.js.

En este caso este fichero estará vacío.

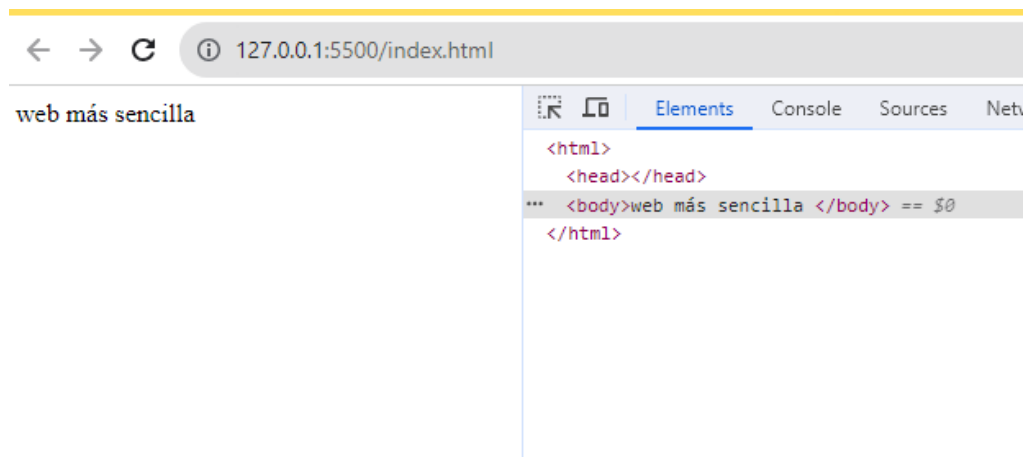
Como vemos, el fichero HTML no contiene más que una etiqueta, que es la única realmente imprescindible para crear una web. El fichero style.css contiene lo convenido para ajustar la web. Y el fichero main.js no incluye nada de código en su interior. Y realmente funciona:



*Ilustración 3: resultado de salida de la web más sencilla.*

Aunque esa web funcione, deja mucho que desear, sobre todo en cuanto a estandarización y buenas prácticas se refiere.





*Ilustración 4: Inspeccionando la web más sencilla.*

Al inspeccionar dicha web mediante el inspector de código del navegador que estemos usando podemos visualizar que el propio intérprete del fichero HTML ha añadido etiquetas HTML por su cuenta sin la participación nuestra. Se puede observar que ha añadido las etiquetas HEAD y BODY. Dichas etiquetas aparecen en el inspector de elementos que hemos conseguido mostrar al llamar a la consola del navegador (tecla F12 en Google Chrome) y seleccionando la pestaña elementos, ya en la consola.

Estas etiquetas no se han añadido a nuestro fichero original en HTML, porque, si lo revisamos ahora mismo, se puede comprobar que no se encuentran dentro del fichero. Pero el intérprete de Google Chrome sí que las incluye virtualmente con la copia que tiene de la web en memoria para estandarizar la web.

Como se puede comprobar tenemos una web totalmente funcional, aunque no podemos esperar mucho de ella, pero es que el código que hemos introducido tampoco ha sido demasiado, para poder esperar algo más de esas pocas líneas de código.

## WEB NORMALIZADA

Ahora vamos a volver a crear una web similar a la anterior pero esta vez con algún que otro código más para ver la diferencia entre una web reducida al máximo con las carencias que tenía y una web bastante más estándar y ajustada a los cánones actuales de programación web.

Fichero HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="style.css">
  <title>Web normalizada</title>
</head>
<body>
  <div>
    Ejemplo de web sencilla pero completa y normalizada para HTML5
  </div>
  <script src="main.js"></script>
</body>
</html>
```

A resaltar en este fichero:

- Hemos incluido una estructura simple de HTML5.
- Hemos incluido un DIV, que no es más que un subespacio creado dentro de la etiqueta BODY.
- También hemos incluido un texto dentro de este espacio DIV.

Fichero CSS;

```
*{
  margin: 0px;
  padding: 0px;
}
body{
  background-color: blue;
  color: white;
  font-size: 30px;
  font-family: Arial, Helvetica, sans-serif;
}
div{
  background-color: black;
  border: 5px;
  margin: 5px;
  border-radius: 15px;
  padding: 10px;
}
```

Para el fichero CSS se puede ver que además de los valores del “margin” y “padding” que solemos incluir en todos los proyectos hemos hecho:

- Para el BODY hemos asignado:
  - Un color de fondo, en este caso, azul.
  - Un color de letra blanco.
  - Un tamaño de letra de 30 pixeles.
  - La familia de fuentes Arial.
- Para el subespacio asignado a DIV hemos fijado:
  - Un color de fondo negro.
  - Un borde de 5 pixeles por todos los lados del espacio que ocupa la etiqueta DIV.
  - Un margen de 5 pixeles por todos los lados del espacio que ocupa la etiqueta DIV.
  - Un radio de borde de 15 pixeles para hacer esa caja del espacio DIV con los bordes redondeados.
  - Un padding o espacio interior de 10 pixeles para que el texto se ajuste perfectamente a esta subárea que hemos creado.

Fichero main.js

Como nuestro empeño en este módulo es aprender JavaScript y aunque la web debe ser lo más sencilla posible, vamos a poner un comentario en el archivo main.js para que aparezca algo en consola y se pueda saber que se está ejecutando correctamente el JavaScript que incluimos.

```
console.log("Ejecutando la web más sencilla....Pero normalizada.");
```

Solamente incluimos la línea anterior en nuestro fichero “main.js” y el resultado será algo similar a lo que se puede ver en la siguiente imagen:



Ilustración 5: resultado web normalizada.

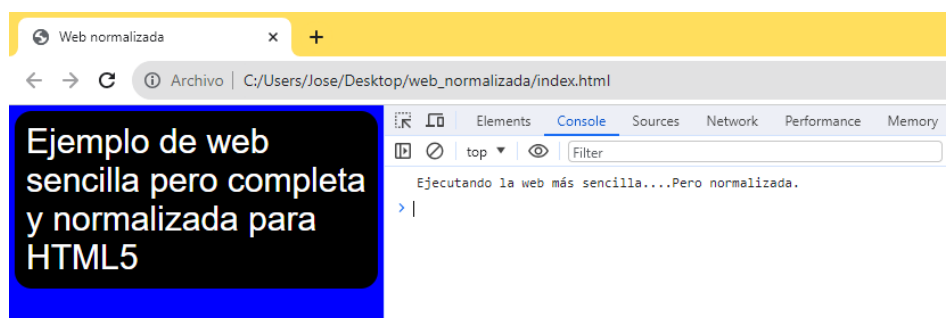


Ilustración 6: resultado y consola de web normalizada.

Se puede observar que se muestra por la consola lo que hemos escrito en el fichero main.js, por tanto, nuestro proyecto está funcionando correctamente y sin ningún fallo, si tuviéramos algún fallo nos aparecería justamente ahí donde nos ha mostrado el texto que le pedíamos a la aplicación que mostrase.

Todos estos mini proyectos que hacemos tienen una explicación de por qué se hacen ahora al principio.

- Sirven para situar al usuario y hacerle ver cómo se va a actuar en cada proyecto estandarizando y aplicando siempre la misma metodología.
- Nos muestra sitios claves a los que controlar a la hora de escribir nuestro código.
- Nos van enseñando gradualmente la potencia, complejidad y elegancia al escribir código y como se debe hacer correctamente.

Con cada ejemplo nos van introduciendo en la magia del desarrollo de las webs dinámicas que se consigue mezclando tecnologías como HTML5, CSS, ambos estáticos, y JavaScript que es quien pone toda la magia para conseguir esa dinamicidad que hacemos mención.



*Ilustración 7: Iconos de HTML5, JavaScript y CSS3.*

Resumiendo, para acabar esta larga introducción, hemos aprendido a:

- Configurar el entorno para ser más rápidos y eficientes escribiendo código.
- Manejar las herramientas necesarias en cada momento según las necesidades que tengamos: Consola, IDE o editor en línea.
- Evitar errores con las variables en JavaScript.
- Construir la estructura básica de cualquier proyecto JavaScript.

Además, hemos visto dos pequeños proyectos en los que hemos visto aplicado casi todo lo aprendido:

- La web más sencilla que existe. Nos ha enseñado que con unas pocas líneas ya tenemos una web totalmente funcional.
- La web normalizada. Nos ha mostrado que para escribir una web normalizada no hace falta tanto.

Y además de todo esto hemos dado nuestros primeros pasitos en JavaScript sin darnos apenas cuenta.

## FUNDAMENTOS JAVASCRIPT

Tal como hemos visto, JavaScript es un lenguaje interpretado, o sea, no es compilado, de desarrollo para webs dinámicas. Con este lenguaje se pueden crear estructuras de datos y manejar elementos del DOM y el BOM para conseguir que una web sea totalmente dinámica y no tan estática como queda con HTML y CSS.

JavaScript interactúa con el DOM para manipular y modificar elementos HTML y CSS en la página. Puede acceder a los elementos del DOM utilizando métodos y propiedades proporcionados por el navegador. Además, JavaScript puede registrar callbacks para eventos específicos, como clics de botón o cambios en el valor de un campo de entrada, y responder a ellos ejecutando el código asociado.

## HISTORIA DE JAVASCRIPT

JavaScript nació en el año 1995 centrado en el navegador (ya desaparecido) Netscape. Todos los navegadores lo adoptaron desde entonces para dotar a las webs de ese dinamismo que no se consigue con HTML y CSS únicamente.

Aunque JavaScript surgió como un lenguaje de script para mejorar las capacidades de la web de la época allá por 1995 por la extinta Netscape, JavaScript no ha dejado de evolucionar desde entonces. Originalmente el lenguaje se basaba en CEnví desarrollado por Nombas.

Brendan Eich, un programador que trabajaba en Netscape pensó que podría solucionar **las limitaciones de la web de entonces**, adaptando otras tecnologías existentes (como ScriptEase) al navegador Netscape Navigator 2.0, que iba a lanzarse en aquel año. Inicialmente, Eich denominó a su lenguaje LiveScript y fue un éxito.

Fue entonces cuando, justo antes del lanzamiento, Netscape decidió **cambiar el nombre por el de JavaScript y firmó una alianza con Sun Microsystems** para continuar el desarrollo del nuevo lenguaje de programación.

Microsoft, al ver el movimiento de uno de sus principales competidores, también decidió incorporar **su propia implementación de este lenguaje**, llamada JScript, en la versión 3 de su navegador Internet Explorer.

Esto contribuyó todavía más al empuje y popularización del lenguaje, pero comenzaron a presentarse pequeños problemas por las **diferencias entre implementaciones**. Por lo que se decidió **estandarizar ambas mediante la versión JavaScript 1.1** como propuesta a ECMA, que culminó con el estándar ECMA-262. Este estándar dicta la base del lenguaje ECMAScript a través de su sintaxis, tipos, sentencias, palabras clave y reservadas, operadores y objetos, y sobre la cual se

pueden construir distintas implementaciones. La versión **JavaScript 1.3** fue la primera implementación completa del estándar **ECMAScript**.

VERSION	AÑO	
ECMAScript 3	1999	<ul style="list-style-type: none"><li>• Soporte expresiones regulares.</li><li>• Manejo de excepciones (try-catch)</li><li>• Nuevas sentencias de control.</li><li>• Definición de errores más precisa.</li><li>• Formateo de salidas numéricas.</li><li>• Manejo de cadenas literales más avanzado.</li></ul>
ECMAScript 4	2004	<ul style="list-style-type: none"><li>• Introduce tipado de variables.</li><li>• Introduce el concepto tradicional de POO con clases e interfaces.</li></ul>
ECMAScript 5	2009	<ul style="list-style-type: none"><li>• Getters y setters.</li><li>• Manipulación de propiedades de un objeto.</li><li>• Crear objetos de forma dinámica.</li><li>• Impedir que un objeto sea modificado.</li><li>• Cambios en el objeto Date.</li><li>• Soporte nativo JSON.</li></ul>
ECMAScript 6	2015	<ul style="list-style-type: none"><li>• Uso de módulos.</li><li>• Ámbito a nivel de bloque (let)</li><li>• Desestructuración.</li><li>• Y muchas novedades de JavaScript avanzado...</li></ul>
ECMAScript 7	2016	

Como curiosidad aquí tenéis un [timeline](#) realizado usando JavaScript que habla sobre la historia de JavaScript. Un claro ejemplo de las cosas espectaculares que se pueden hacer con este lenguaje de script llamado JavaScript.

## CARACTERISTICAS DE JAVASCRIPT

JavaScript es un lenguaje interpretado debido a la forma en que se ejecuta y procesa su código.

- **Entorno de ejecución en el navegador:** JavaScript se ejecuta principalmente en los navegadores web. Cuando se carga una página web que contiene código JavaScript, el navegador interpreta y ejecuta ese código directamente en su entorno de ejecución interno. Esto significa que no se requiere un paso de compilación previo antes de ejecutar el código JavaScript.
- **Análisis y ejecución línea por línea:** a diferencia de los lenguajes compilados, en los que todo el código se traduce en instrucciones de máquina antes de su ejecución, JavaScript se interpreta línea por línea a medida que se encuentra. El motor de JavaScript del navegador analiza y ejecuta cada línea de código a medida que se encuentra durante el proceso de carga de la página.

- **Flexibilidad y facilidad de desarrollo:** el enfoque interpretado de JavaScript brinda flexibilidad y facilidad de desarrollo. No es necesario compilar el código antes de probarlo, lo que permite un desarrollo rápido y una iteración ágil. Los cambios realizados en el código JavaScript se pueden probar y ver inmediatamente sin la necesidad de un proceso de compilación.
- **Plataforma independiente:** el hecho de que JavaScript sea un lenguaje interpretado contribuye a su capacidad de ser ejecutado en diferentes plataformas y sistemas operativos. Los navegadores web son la plataforma principal para la ejecución de JavaScript, y los navegadores modernos han implementado sus propios motores para interpretar y ejecutar el código en diferentes sistemas.
- **Fácil actualización y distribución:** con JavaScript interpretado, los cambios o actualizaciones en el código se pueden implementar fácilmente, ya que los usuarios solo necesitan cargar la nueva versión del código en sus navegadores. No es necesario que los usuarios realicen una instalación manual o actualización de un programa o aplicación.

## POO

JavaScript es un lenguaje de programación orientado a objetos (OOP), que proporciona funcionalidades para trabajar con objetos y clases. Aunque no sigue un enfoque de OOP estricto como lenguajes como Java o C++, ofrece características que permiten la implementación de conceptos orientados a objetos.

- **Objetos:** en JavaScript, los objetos son colecciones de propiedades y métodos que representan entidades del mundo real. Las propiedades son variables que almacenan valores, mientras que los métodos son funciones que realizan acciones relacionadas con el objeto. Los objetos se crean utilizando la sintaxis de llaves {} y pueden ser personalizados para contener propiedades y métodos específicos.
- **Prototipos:** utiliza un modelo de herencia basado en prototipos en lugar de clases tradicionales. Cada objeto tiene un prototipo, que es otro objeto del cual hereda propiedades y métodos. Cuando se accede a una propiedad o método en un objeto, si no se encuentra directamente en ese objeto, JavaScript busca en su prototipo y continúa escalando por la cadena de prototipos hasta que se encuentra o se alcanza el objeto raíz.
- **Constructores:** permite la creación de objetos utilizando funciones constructoras. Una función constructora es una función regular que se invoca utilizando la palabra clave «new». Dentro de la función constructora se definen las propiedades y métodos del objeto que se está creando. Cada vez que se crea un objeto utilizando una función constructora, se crea una instancia independiente con sus propias propiedades y métodos.
- **Herencia:** aunque JavaScript no ofrece una sintaxis de clase tradicional para la herencia, se puede conseguir la herencia utilizando prototipos. Un objeto puede heredar propiedades y métodos de otro objeto estableciendo su prototipo en ese objeto padre. Esto permite la reutilización de código y la creación de jerarquías de objetos.
- **Encapsulamiento:** JavaScript no tiene un mecanismo nativo de encapsulamiento como las clases en otros lenguajes, pero se puede lograr usando funciones para crear ámbitos locales. Al definir variables y funciones dentro de una función, se pueden ocultar y proteger del acceso externo, creando así un nivel básico de encapsulamiento.
- **Polimorfismo:** permite la implementación de polimorfismo, que es la capacidad de objetos de diferentes tipos para responder al mismo mensaje o llamada de método. Dado que JavaScript es un lenguaje dinámico, las variables pueden contener diferentes tipos de

objetos en distintos momentos, y se puede acceder a sus métodos y propiedades de manera flexible.

## DOM

La integración de JavaScript con el DOM (Document Object Model) es una de las características fundamentales de JavaScript y es lo que permite interactuar y manipular elementos HTML y CSS en una página web.

- **Acceso al DOM:** JavaScript utiliza el objeto `document` para acceder y manipular el DOM. El objeto `document` representa el documento HTML cargado en el navegador y proporciona métodos y propiedades para interactuar con él. Puedes acceder a elementos específicos del DOM utilizando métodos como `getElementById()`, `querySelector()`, `getElementsByName()`, entre otros.
- **Manipulación de elementos:** una vez que tienes una referencia a un elemento en el DOM, puedes utilizar las propiedades y métodos disponibles para manipularlo. Por ejemplo, puedes cambiar el contenido de un elemento utilizando la propiedad `innerHTML`, modificar los estilos utilizando la propiedad `style`, agregar o eliminar atributos utilizando los métodos `setAttribute()` y `removeAttribute()`, y mucho más.
- **Eventos del DOM:** JavaScript permite responder a eventos que ocurren en los elementos del DOM, como hacer clic en un botón, pasar el cursor sobre un elemento o enviar un formulario. Puedes registrar eventos utilizando métodos como `addEventListener()`, especificando el tipo de evento y una función de callback que se ejecutará cuando ocurra el evento. Dentro de la función de callback, puedes escribir el código que deseas ejecutar en respuesta al evento.
- **Manipulación de la estructura del DOM:** además de manipular los elementos individuales, JavaScript también permite manipular la estructura del DOM, es decir, agregar, eliminar y modificar elementos en la página. Puedes crear nuevos elementos utilizando el método `createElement()`, agregarlos al DOM usando métodos como `appendChild()` o `insertBefore()`, y eliminar elementos empleando `removeChild()`.
- **Actualización dinámica de la página:** una de las ventajas clave de la integración de JavaScript con el DOM es la capacidad para realizar actualizaciones dinámicas en la página sin tener que recargarla completa. Puedes cambiar el contenido de un elemento en respuesta a eventos, actualizar estilos, mostrar u ocultar elementos, realizar animaciones y mucho más. Esto permite crear experiencias interactivas y dinámicas para los usuarios.

## BASADO EN EVENTOS

JavaScript es un lenguaje «event-driven» o basado en eventos, lo que significa que se ejecuta en respuesta a eventos que ocurren en el entorno en el que se está ejecutando, como acciones del usuario o eventos del sistema.

- **Registro de eventos:** para comenzar a responder a eventos, se deben registrar los eventos que se desean detectar. En JavaScript, esto se realiza utilizando el método `addEventListener()` en un elemento específico del DOM (Document Object Model) o en un



objeto relevante. Se especifica el tipo de evento (por ejemplo, «click» para un clic de ratón) y una función de «callback» que se ejecutará cuando ocurra el evento.

- **Funciones de callback:** las funciones de callback se ejecutan en respuesta a un evento específico. Estas funciones se definen previamente y se pasan como argumento al método `addEventListener()`. Cuando se produce el evento registrado, JavaScript ejecuta la función de callback correspondiente.
- **Ejecución asíncrona:** el enfoque «event-driven» en JavaScript permite la ejecución asíncrona, lo que significa que el código no se bloquea esperando eventos. El flujo de ejecución continúa mientras el programa espera eventos, y cuando ocurre un evento registrado, se dispara la ejecución de la función de callback asociada.
- **Event bubbling y event capturing:** JavaScript también proporciona dos modelos de propagación de eventos: «event bubbling» (burbujeo de eventos) y «event capturing» (captura de eventos). El «event bubbling» hace que los eventos se propaguen desde el elemento objetivo hacia los elementos padres, mientras que el «event capturing» hace que los eventos se propaguen desde los elementos padres hacia el elemento objetivo. Esto permite controlar cómo se manejan los eventos en diferentes elementos.
- **Manipulación del DOM:** una de las principales aplicaciones del enfoque «event-driven» en JavaScript es la manipulación del DOM. Al registrar eventos en elementos del DOM, se puede responder a las interacciones del usuario, como clics, cambios de valor en campos de entrada o movimiento del ratón. Esto posibilita crear interacciones dinámicas y responsivas en aplicaciones web.

## ASINCRONISMO

El asincronismo en JavaScript es fundamental para realizar tareas sin bloquear la ejecución del código, lo que permite que el programa sea más eficiente y receptivo.

- **Callbacks:** antes de que las promesas y el `async/await` estuvieran disponibles, eran una forma común de implementar la asincronía en JavaScript. Un callback es una función que se pasa como argumento a otra función y se ejecuta una vez que se completa una operación asíncrona. Por ejemplo, en una llamada AJAX, se proporciona un callback que se ejecuta cuando se recibe la respuesta del servidor.
- **Event Loop:** JavaScript utiliza un mecanismo llamado Event Loop (bucle de eventos) para manejar la asincronía. El Event Loop se encarga de monitorear constantemente la pila de llamadas y la cola de tareas. Cuando todas las tareas en la pila de llamadas se han ejecutado, el Event Loop verifica la cola de tareas en busca de tareas pendientes y las ejecuta en orden.
- **Promesas:** las promesas son una forma más moderna y legible de manejar la asincronía en JavaScript. Una promesa es un objeto que representa la eventual finalización o falla de una operación asíncrona y permite encadenar acciones después de que se haya resuelto o rechazado. Puedes definir una promesa que realiza una tarea y luego utilizar los métodos `then()` y `catch()` para manejar el resultado.
- **Async/await:** el `async/await` es la sintaxis más reciente que se ha introducido en JavaScript y simplifica aún más el manejo de la asincronía. Permite escribir código asíncrono de manera más similar al código síncrono, lo que lo hace más legible y fácil de entender. Puedes marcar una función como `async` y utilizar la palabra clave `await` para esperar la resolución de una promesa antes de continuar la ejecución.

- **Llamadas a API:** una de las aplicaciones comunes del asincronismo en JavaScript es realizar llamadas a API, como solicitudes AJAX o fetch para obtener datos de servidores externos. Estas operaciones suelen ser asíncronas, y el código espera la respuesta de la API antes de continuar con otras tareas.

## VARIABLES

El lenguaje JavaScript es un lenguaje débilmente tipado lo que significa, entre otras implicaciones, que no es necesario declarar que tipo de variable es al declarar dicha variable, así, como tampoco es necesario que se mantengan durante toda la ejecución de la aplicación siendo de ese mismo tipo con el que se inicio al principio de dicha aplicación. Ejemplos:

```
let miVariable;           // declaro miVariable y como no se asignó un valor valdrá undefined
miVariable='Hola';        // ahora su valor es 'Hola', por tanto, contiene una cadena de texto
miVariable=34;            // pero ahora contiene un número
miVariable=[3, 45, 2];    // y ahora un array
miVariable=undefined;     // para volver a valer el valor especial undefined
```

**EJERCICIO:** Ejecuta en la consola del navegador las instrucciones anteriores y comprueba el valor de miVariable tras cada instrucción (para ver el valor de una variable simplemente ponemos en la consola su nombre: miVariable).

JavaScript es tan laxo con el uso de las variables que ni siquiera nos obliga a declarar dichas variables antes de usarlas, pero, es un problema que posteriormente pagaremos en la depuración, perdiendo más tiempo que el que puede suponer declarar correctamente una variable.

## VAR, LET Y CONST

Se usa la instrucción var para declarar variables de tipo global, es decir, el ámbito de dicha variable será la aplicación completa.

Se usa la instrucción let (recomendado desde ES2015) para declarar variables de bloque o locales ya que su ámbito es reducido al bloque o a un ámbito más pequeño que en el caso de la instrucción var.

Veamos un ejemplo de su uso:

```
let edad = 17;

if (edad >= 18) {
  let textoLet = 'Eres mayor de edad';
  var textoVar = 'Eres mayor de edad';
} else {
  let textoLet = 'Eres menor de edad';
}
```

```
var textoVar = 'Eres menor de edad';  
}  
//console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable  
console.log(textoVar); // mostrará la cadena
```

Es altamente recomendable, por no decir, obligatorio el uso de la tipología camelCase en JavaScript (ej.: miPrimerApellido)

Desde ES2015 podemos hacer uso de la palabra reservada “const” para declarar constantes se les asigna un valor al declarar dichas constantes y si posteriormente intentamos cambiar su valor nos producirá un error.

```
const miNombre = "José Manuel";  
const miTelefono = 123456789;  
const miSituacionLaboral = true; // true --> trabajando false ---> desempleado  
const misAnimales = [true,true, true]; // 0: perro 1:gato 2:pajaro
```

Cualquier variable que no sea declarada dentro de un bloque o una función, o si usamos una variable sin declarar previamente es considerada una variable global.

## FUNCIONES

Las funciones se declaran con la palabra reservada “function” y en el caso de que necesite parámetros se le pasan entre paréntesis. Las funciones siempre devuelven un valor, en el caso de contener un return dentro de la función se devuelve el valor asignado a esa orden, en caso contrario, se devuelve undefined por parte de la función implicada.

Las funciones en JavaScript no es necesario declararlas al principio (hoisting de JavaScript), se pueden declarar incluso todas al final del código que estamos creando ya que el intérprete del navegador al principio lee todas las variables y funciones y las mueve al principio del código para posteriormente empezar la ejecución línea por línea.

**Ejercicio:** Crea una función con un parámetro de entrada llamado “texto”, la ejecución de esa función debe mostrar una alerta con el texto que has introducido como parámetro al hacer la llamada de dicha función y al que le agregaras anteriormente el texto: “Has escrito...” seguido de tu texto escrito en la llamada a la función.

### Parámetros

Si llamo a una función con menos parámetros de los que se han declarado al crear la función, el valor de esos parámetros no declarados será undefined.

```
function potencia(base, exponente) {  
    console.log(base);           // muestra 4  
    console.log(exponente);      // muestra undefined  
    let valor=1;  
    for (let i=1; i<=exponente; i++) {
```

```

        valor=valor*base;
    }
    return valor;
}

potencia(4);    // devolverá 1 ya que no se ejecuta el for

```

**Pregunta:** ¿Cómo sería la llamada correcta a la función que acabamos de declarar? Escribe al menos una llamada correcta a esa función.

**Atención:** ¿Por qué hace llamadas a la consola dentro de la función? ¿Hacer esas llamadas a consola desde dentro de la función es correcto?

Para evitar esta problemática con los parámetros de entrada de las funciones podemos usar la preasignación de valores a los parámetros de la siguiente forma:

```

function potencia(base, exponente=2) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra 2 la primera vez y 5 la segunda
    let valor=1;
    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

console.log(potencia(4));        // mostrará 16 (4^2)
console.log(potencia(4,5));     // mostrará 1024 (4^5)

```

Podemos acceder a los parámetros de una función desde el array `arguments[]` en el caso de no conocer el número de parámetros que finalmente nos pasó el usuario:

```

function suma () {
    var result = 0;
    for (var i=0; i<arguments.length; i++)
        result += arguments[i];
    return result;
}

console.log(suma(4, 2));          // mostrará 6
console.log(suma(4, 2, 5, 3, 2, 1, 3)); // mostrará 20

```

JavaScript trata las funciones como un tipo de dato más y podemos pasarlas como argumento o asignarlas a una variable:

```

const cuadrado = function(value) {
    return value * value
}

function aplica_fn(dato, funcion_a_aplicar) {
    return funcion_a_aplicar(dato);
}

```

```
}  
  
aplica_fn(3, cuadrado);    // devolverá 9 (3^2)
```

A las funciones así usadas se les llama funciones de primera clase y son muy típicas de lenguajes funcionales.

## Funciones anónimas

Acabamos de usar una función, no sé si os habréis fijado en el detalle, que no tiene nombre. Se denominan funciones anónimas. Esta función, o cualquier función, puede ser asignada a una variable, autoejecutarse o ser asignada a un manejador de eventos. Ejemplo:

```
let holaMundo = function() {  
    alert('Hola mundo!');  
}  
  
holaMundo();    // se ejecuta la función
```

En este caso concreto, debemos asignar esa función anónima a una variable para que llamando a esa variable se pueda ejecutar la función. Se debe observar que la variable que ahora contiene la función anónima se debe hacer uso de ella utilizando el paréntesis, ya que de lo contrario no ejecutaría la función que nos ocupa.

## Funciones flecha

Las funciones flecha o funciones lambda es una forma de declarar una función más corta que la forma tradicional de declaración de funciones. Se permiten desde ES2015. Ejemplo de declaración de una función de manera tradicional:

```
let potencia = function(base, exponente) {  
    let valor=1;  
    for (let i=1; i<=exponente; i++) {  
        valor=valor*base;  
    }  
    return valor;  
}
```

Ahora, si pretendemos reescribir esta misma función en modo función flecha debemos:

- Eliminar la palabra reservada function.
- Si solamente posee un parámetro podemos eliminar los paréntesis de los parámetros.
- Colocamos el símbolo =>

- Si la función solamente tiene una línea podemos eliminar las llaves {} y la palabra reservada return.

El mismo ejemplo anterior en formato flecha sería:

```
let potencia = (base, exponente) => {  
  let valor=1;  
  for (let i=1; i<=exponente; i++) {  
    valor=valor*base;  
  }  
  return valor;  
}
```

Otro ejemplo declarando la función de manera tradicional:

```
let cuadrado = function(base) {  
  return base * base;  
}
```

El mismo ejemplo en formato flecha:

```
let cuadrado = base => base * base;
```

**EJERCICIO:** Haz una función flecha que devuelva el cubo del número pasado como parámetro y pruébala desde la consola. Escríbela primero en la forma habitual y luego la “traduces” a formato flecha.

## ESTRUCTURAS Y BLOQUES

### CONDICIONAL IF

Ejemplo de un condicional de tipo IF sencillo

```
let edad = 17;  
  
if (edad >= 18) {  
  let textoLet = 'Eres mayor de edad';  
  var textoVar = 'Eres mayor de edad';  
}  
  
//console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable  
console.log(textoVar); // mostrará la cadena
```

En el caso anterior se pueden distinguir dos partes principales en el condicional IF, la primera parte principal se encuentra entre paréntesis y nos indica la condición que debe cumplirse para que se

ejecute lo que existe dentro de la segunda parte principal a destacar que está dentro de las llaves del final del condicional. En este ejemplo no existe la clausula ELSE que se ejecutaría en caso de no cumplirse la condición anteriormente comentada. Vemos a continuación el mismo condicional, esta vez si, con clausula ELSE:

```
let edad = 17;

if (edad >= 18) {
  let textoLet = 'Eres mayor de edad';
  var textoVar = 'Eres mayor de edad';
} else {
  let textoLet = 'Eres menor de edad';
  var textoVar = 'Eres menor de edad';
}

//console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable
console.log(textoVar); // mostrará la cadena
```

Como hemos dicho anteriormente, en esta última propuesta del condicional se evalúa siempre la condición entre paréntesis, si es verdadera esa condición se ejecuta el código encerrado entre llaves a continuación de la condición. Si fuese falsa esa condición se ejecutaría la segunda sección de código que va entre llaves después de la condición.

El condicional IF admite encadenar sucesivos condicionales mediante el uso de ELSE IF, esta construcción lógica se usa para contemplar todos los casos posibles a tratar frente a unas especificaciones, veamos un ejemplo:

```
let edad = 29;

if(edad >= 18 && edad < 120){
  console.log("Eres mayor de edad");
}else if(edad < 12){
  console.log("Eres un niño.");
}else if(edad >= 12 && edad < 18){
  console.log("Eres un adolescente");
}else{
  console.log("Tu edad no es correcta.");
}
```

En el caso anterior al valer la variable edad 29 el condicional primero se cumple y se ejecuta el código mostrado entre llaves a continuación, mostrando por consola el mensaje que nos indica que el usuario es un adulto.

Haciendo suposiciones y que funcionase correctamente el código, si imaginamos que ahora la variable edad puede cambiar y valer 11 se evaluaría el primer condicional y como no es verdadero ya que 11 es menor de 120 pero no es mayor que 18 y puesto que hemos usado un operador lógico AND para unir dichas condiciones, el resultado de una operación falso AND verdadero es falso, por tanto, se evaluaría el segundo condicional resultando que es verdadero, ya que 11 es menor que 12.

Por lo que se ejecutaría el código entre llaves justo tras esa condición mostrando el mensaje en consola "Eres un niño".

Volvemos a hacer otra suposición, ahora igualamos la variable edad a 15, entramos en el primer condicional y al evaluar esa condición vemos que su resultado es falso, ya que aunque es verdadero que 15 es menor que 120, también, es falso que 15 es mayor o igual a 18., por tanto, no se ejecuta el código entre llaves del primer condicional y pasamos a evaluar la segunda condición, esta vez, 15 no es cierto que sea menor que 12, o sea, condición falsa, no se ejecuta el código entre llaves tras esa segunda condición y pasamos a evaluar la tercera condición. La tercera condición nos asegura que si edad es menor que 18, si es menor que 18 el numero 15, y la edad es mayor o igual que 12, que también es cierta entonces se ejecuta el código entre llaves a continuación, indicándonos por consola que "Eres un adolescente."

Y así se irían evaluando cada una de las condiciones descritas asegurándonos cubrir todas las posibilidades al crear dichos condicionales. Y si lo hemos diseñado bien y lo hemos implementado correctamente, tal y como lo hemos diseñado. Nunca deben fallar nuestros condicionales. En el caso que nos ocupa, si llegase al ultimo condicional y ninguno hubiera sido cierto nos daría un mensaje de error que en este caso hemos definido como: "Tu edad no es correcta."

## CONDICIONAL SWITCH

## BUCLE WHILE

## BUCLE FOR

Ya hemos visto en funcionamiento el bucle FOR al definir las distintas formas de declarar funciones en JavaScript. Pero volvamos a él, desmenuzándolo adecuadamente para disipar cualquier duda existente sobre esta construcción.

```
let potencia = (base, exponente) => {  
  let valor=1;  
  for (let i=1; i<=exponente; i++) {  
    valor=valor*base;  
  }  
  return valor;  
}
```



En todo bucle FOR tenemos dos partes principales:

- La parte encerrada entre paréntesis. Este apartado nos muestra la configuración del bucle FOR, ya que aquí decidimos que variable usamos para iterar en cada ocasión, es el caso de nuestro ejemplo donde se define la variable con el nombre "i". También decidimos la condición que se debe cumplir y que debe ser cierta para que se continúe iterando, en nuestro ejemplo viene definido por esa condición "i <= exponente". Y para finalizar, también definimos los saltos de valor que va a dar la variable en cada una de las iteraciones, en nuestro ejemplo viene definido como "i++". Es decir, si saltamos de uno en uno en los valores asignados a la variable "i", como indica esta última instrucción "i++" analizada, en cada iteración. Las iteraciones se irán produciendo hasta que el valor en la variable "i" sea menor o igual que el valor del parámetro "exponente". Si esa condición llegase a ser falsa en esa iteración ya ejecutaría el código entre llaves y se daría por finalizado el bucle FOR. CUIDADO con los bucles que podemos definir condiciones imposibles y que siempre se cumplan y entrar en un bucle infinito que colgaría nuestra aplicación.
- La parte encerrada entre llaves. Como ya hemos comentado el código incluido entre llaves se ejecuta, si y solo si, el condicional de la declaración del bucle FOR es verdadero. Esta parte puede incluir cualquier código incluidos más bucles FOR, condicionales, o cualquier código que se te ocurra, como funciones de cualquier tipo, objetos, etc.

BUCLE

TIPOS DE DATOS BÁSICOS

CASTING DE VARIABLES

NUMBER

STRING

BOOLEAN

MANEJO DE ERRORES

## BUENAS PRACTICAS

use strict

Variables

Errores

## OBJETOS

### INTRODUCCION

En JavaScript podemos declarar un objeto de tres formas distintas:

- Objetos declarativos o literales.
- Objetos contruidos.
- Objetos usando new Object.

#### *Objetos literal o declarativo*

```
let persona = {  
  nombre : 'Jack',  
  edad : 30,  
  saludar: function () {  
    console.log('Hola');  
  }  
};
```

Se declara exactamente igual que cualquier variable con la salvedad de usar las llaves para definir las propiedades o métodos que incluya ese objeto recién creado.

#### *Objeto new Object*

El ejemplo anterior contruido de esta manera sería:

```
let persona = new Object({
```

```
nombre : 'Jack',  
edad : 30,  
saludar : function () {  
    console.log('Hola');  
}  
});
```

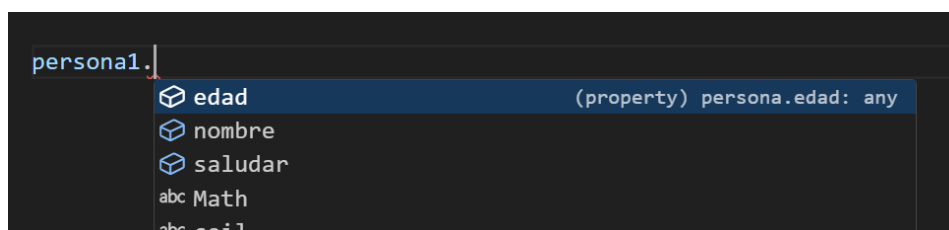
### Objeto construido

El ejemplo que vamos siguiendo construido de esta forma sería:

```
function persona(nombre, edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
  
    this.saludar = function(){  
        console.log('Hola');  
    }  
}  
  
let persona1 = new persona('Jack', 30);
```

## PROPIEDADES

La manera de acceder a las propiedades de cada objeto que creemos, se declare como se declare el objeto, es siempre con el “.”, es decir, para acceder a la propiedad “nombre” del objeto anteriormente declarado “persona”, se haría así, al pulsar el punto después de añadir el nombre del objeto referenciado, vemos que nos aparecen los métodos y propiedades en la ayuda de VS Code:



Se puede observar que aparecen las propiedades definidas: edad y nombre, así como también, el método definido: saludar.

```
console.log(persona1.edad); // nos mostraria en consola un 30.
```

Como vemos al seleccionar esa propiedad “edad”, nos indicaría por consola la instrucción anterior un 30, ya que fue el numero que asignamos a ese parámetro al definir el objeto.

## METODOS

Con los métodos ocurre igual que con las propiedades de los objetos, se referencian con el punto. Y se usa el paréntesis puesto que un método no es más que una función de un objeto.

```
console.log( persona1.saludar() );
```

Esta instrucción anterior ejecuta el código definido en el método declarado en la declaración de dicho objeto y, en nuestro ejemplo, mostraría por consola la palabra “hola”. Su funcionamiento, aunque sea un método de un objeto, es igual a cualquier otra función, la única salvedad es que solamente conoce lo que hace esa función el objeto al que pertenece. Si intentásemos ejecutar dicha función sin hacer referencia a dicho objeto y no existiera una función con ese nombre a nivel global de la aplicación obtendríamos un error.

## ARRAYS

Son un tipo de objeto y no tienen tamaño fijo, sino que, podemos añadirle elementos en cualquier momento.

Se recomienda crearlos usando notación JSON:

```
let a = []  
let b = [2,4,6]
```

aunque también podemos crearlos como instancias del objeto Array (NO recomendado):

```
let a = new Array()      // a = []  
let b = new Array(2,4,6) // b = [2, 4, 6]
```

Sus elementos pueden ser de cualquier tipo, incluso podemos tener elementos de tipos distintos en un mismo array. Si no está definido un elemento su valor será *undefined*. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a[0]) // imprime 'Lunes'
console.log(a[4]) // imprime 6
a[7] = 'Juan'      // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']
console.log(a[7])  // imprime 'Juan'
console.log(a[6])  // imprime undefined
console.log(a[10]) // imprime undefined
```

Acceder a un elemento de un array que no existe no provoca un error (devuelve *undefined*) pero sí lo provoca acceder a un elemento de algo que no es un array. Con ES2020 (ES11) se ha incluido el operador **?.** para evitar tener que comprobar nosotros que sea un array:

```
console.log(alumnos?.[0])
// si alumnos es un array muestra el valor de su primer
// elemento y si no muestra undefined pero no lanza un error
```

## Arrays de objetos

Es habitual almacenar datos en arrays en forma de objetos, por ejemplo:

```
let alumnos = [
  {
    id: 1,
    name: 'Marc Peris',
    course: '2nDAW',
    age: 21
  },
  {
    id: 2,
    name: 'Júlia Tortosa',
    course: '2nDAW',
    age: 23
  },
]
```

## Operaciones con Arrays

Vamos a ver los principales métodos y propiedades de los arrays.

length

Esta propiedad devuelve la longitud de un array:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a.length) // imprime 5
```

Podemos reducir el tamaño de un array cambiando esta propiedad, aunque es una forma poco clara de hacerlo:

```
a.length = 3 // ahora a = ['Lunes', 'Martes', 2]
```

## Añadir elementos

Podemos añadir elementos al final de un array con push o al principio con unshift:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
a.push('Juan') // ahora a = ['Lunes', 'Martes', 2, 4, 6, 'Juan']
a.unshift(7) // ahora a = [7, 'Lunes', 'Martes', 2, 4, 6, 'Juan']
```

## Eliminar elementos

Podemos borrar el elemento del final de un array con pop o el del principio con shift. Ambos métodos devuelven el elemento que hemos borrado:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let ultimo = a.pop() // ahora a = ['Lunes', 'Martes', 2, 4] y ultimo = 6
let primero = a.shift() // ahora a = ['Martes', 2, 4] y primero = 'Lunes'
```

## splice

Permite eliminar elementos de cualquier posición del array y/o insertar otros en su lugar. Devuelve un array con los elementos eliminados. Sintaxis:

```
Array.splice(posicion, num. de elementos a eliminar, 1º elemento a insertar, 2º elemento a insertar, 3º...)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let borrado = a.splice(1, 3) // ahora a = ['Lunes', 6] y borrado = ['Martes', 2, 4]
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 0, 45, 56) // ahora a = ['Lunes', 45, 56, 'Martes', 2, 4, 6] y borrado = []
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 3, 45, 56) // ahora a = ['Lunes', 45, 56, 6] y borrado = ['Martes', 2, 4]
```

EJERCICIO: Guarda en un array la lista de la compra con Peras, Manzanas, Kiwis, Plátanos y Mandarinas. Haz lo siguiente con splice:

- Elimina las manzanas (debe quedar Peras, Kiwis, Plátanos y Mandarinas)
- Añade detrás de los Plátanos Naranjas y Sandía (debe quedar Peras, Kiwis, Plátanos, Naranjas, Sandía y Mandarinas)
- Quita los Kiwis y pon en su lugar Cerezas y Nísperos (debe quedar Peras, Cerezas, Nísperos, Plátanos, Naranjas, Sandía y Mandarinas)

slice

Devuelve un subarray con los elementos indicados, pero sin modificar el array original (sería como hacer un substr pero de un array en vez de una cadena). Sintaxis:

```
Array.slice(posicion, num. de elementos a devolver)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let subArray = a.slice(1, 3)
// ahora a = ['Lunes', 'Martes', 2, 4, 6] y subArray = ['Martes', 2, 4]
```

Es muy útil para hacer una copia de un array:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()
// ahora ambos arrays contienen lo mismo pero son diferentes arrays
```

## Arrays y Strings

Cada objeto (y los arrays son un tipo de objeto) tienen definido el método `.toString()` que lo convierte en una cadena. Este método es llamado automáticamente cuando, por ejemplo, queremos mostrar un array por la consola. En realidad `console.log(a)` ejecuta `console.log(a.toString())`. En el caso de los arrays esta función devuelve una cadena con los elementos del array dentro de corchetes y separados por coma.

Además podemos convertir los elementos de un array a una cadena con `.join()` especificando el carácter separador de los elementos. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let cadena = a.join('-') // cadena = 'Lunes-Martes-2-4-6'
```

Este método es el contrario del método `.split()` que convierte una cadena en un array. Ej.:

```
let notas = '5-3.9-6-9.75-7.5-3'
let arrayNotas = notas.split('-') // arrayNotas = [5, 3.9, 6, 9.75, 7.5, 3]
let cadena = 'Que tal estás'
let arrayPalabras = cadena.split(' ') // arrayPalabras = ['Que', 'tal', 'estás']
let arrayLetras = cadena.split('') // arrayLetras = ['Q','u','e',' ','t','a','l',' ','e','s','t','á','s']
```



## sort

Ordena **alfabéticamente** los elementos del array

```
let a = ['hola','adios','Bien','Mal',2,5,13,45]
let b = a.sort()      // b = [13, 2, 45, 5, "Bien", "Mal", "adios", "hola"]
```

También podemos pasarle una función que le indique cómo ordenar que devolverá un valor negativo si el primer elemento es mayor, positivo si es mayor el segundo o 0 si son iguales. Ejemplo: ordenar un array de cadenas sin tener en cuenta si son mayúsculas o minúsculas:

```
let a = ['hola','adios','Bien','Mal']
let b = a.sort(function(elem1, elem2) {
  if (elem1.toLocaleLowerCase > elem2.toLocaleLowerCase)
    return -1
  if (elem1.toLocaleLowerCase < elem2.toLocaleLowerCase)
    return 1
  return 0
})      // b = ["adios", "Bien", "hola", "Mal"]
```

Como más se utiliza esta función es para ordenar arrays de objetos. Por ejemplo si tenemos un objeto *alumno* con los campos *name* y *age*, para ordenar un array de objetos *alumno* por su edad haremos:

```
let alumnosOrdenado = alumnos.sort(function(alumno1, alumno2) {
  return alumno1.age - alumno2.age
})
```

Usando *arrow functions* quedaría más sencillo:

```
let alumnosOrdenado = alumnos.sort((alumno1, alumno2) => alumno1.age - alumno2.age)
```

Si lo que queremos es ordenar por un campo de texto debemos usar la función *toLocaleCompare*:

```
let alumnosOrdenado = alumnos.sort((alumno1, alumno2) => alumno1.name.toLocaleCompare(alumno2.name))
```

EJERCICIO: Haz una función que ordene las notas de un array pasado como parámetro. Si le pasamos [4,8,3,10,5] debe devolver [3,4,5,8,10]. Pruébalo en la consola

## Otros métodos comunes

Otros métodos que se usan a menudo con arrays son:

- **concat()**: concatena arrays

```
let a = [2, 4, 6]
let b = ['a', 'b', 'c']
let c = a.concat(b)      // c = [2, 4, 6, 'a', 'b', 'c']
```

- **.reverse()**: invierte el orden de los elementos del array

```
let a = [2, 4, 6]
let b = a.reverse()      // b = [6, 4, 2]
```

- **.indexOf()**: devuelve la primera posición del elemento pasado como parámetro o -1 si no se encuentra en el array.
- **.lastIndexOf()**: devuelve la última posición del elemento pasado como parámetro o -1 si no se encuentra en el array.

## Functional Programming

Se trata de un paradigma de programación (una forma de programar) donde se intenta que el código se centre más en qué debe hacer una función que en cómo debe hacerlo. El ejemplo más claro es que intenta evitar los bucles *for* y *while* sobre arrays o listas de elementos.

Normalmente cuando hacemos un bucle es para recorrer la lista y realizar alguna acción con cada uno de sus elementos. Lo que hace *functional programming* es que a la función que debe hacer eso se le pasa como parámetro la función que debe aplicarse a cada elemento de la lista.

Desde la versión 5.1 JavaScript incorpora métodos de *functional programming* en el lenguaje, especialmente para trabajar con arrays:

## filter

Devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica. Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array. Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar). Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo. La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado y **false** para el resto.

Ejemplo: dado un array con notas devolver un array con las notas de los aprobados. Esto usando programación *imperativa* (la que se centra en *cómo se deben hacer las cosas*) sería algo como:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = []
for (let i = 0; i++ < arrayNotas.length) {
  let nota = arrayNotas[i]
  if (nota >= 5) {
    aprobados.push(nota)
  }
} // aprobados = [5.2, 6, 9.75, 7.5]
```

Usando *functional programming* (la que se centra en *qué resultado queremos obtener*) sería:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  if (nota >= 5) {
    return true
  } else {
    return false
  }
}) // aprobados = [5.2, 6, 9.75, 7.5]
```

Podemos refactorizar esta función para que sea más compacta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  return nota >= 5 // nota >= 5 se evalúa a 'true' si es cierto o 'false' si no lo es
})
```

Y usando funciones lambda la sintaxis queda mucho más simple:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(nota => nota >= 5)
```

Vamos a construir un array con los días de la semana que usaremos en los próximos ejercicios:

```
const semana = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo'];
```

Las 7 líneas del código usando programación *imperativa* quedan reducidas a sólo una.

EJERCICIO: Dado un array con los días de la semana obtén todos los días que empiezan por 'M'

## find

Como *filter* pero NO devuelve un **array** sino el primer **elemento** que cumpla la condición (o *undefined* si no la cumple nadie). Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let primerAprobado = arrayNotas.find(nota => nota >= 5) // primerAprobado = 5.2
```

Este método tiene más sentido con objetos. Por ejemplo, si queremos encontrar la persona con DNI '21345678Z' dentro de un array llamado personas cuyos elementos son objetos con un campo 'dni' haremos:

```
let personaBuscada = personas.find(persona => persona.dni === '21345678Z') // devolverá el
objeto completo
```

EJERCICIO: Dado un array con los días de la semana obtén el primer día que empieza por 'M'

```
let primerDiaConM = semana.find(dia => (dia.startsWith('M')));
console.log(primerDiaConM);
```

## findIndex

Como *find* pero en vez de devolver el elemento devuelve su posición (o -1 si nadie cumple la condición). En el ejemplo anterior el valor devuelto sería 0 (ya que el primer elemento cumple la condición). Al igual que el anterior tiene más sentido con arrays de objetos.

EJERCICIO: Dado un array con los días de la semana obtén la posición en el array del primer día que empieza por 'M'

## every / some

La primera devuelve **true** si **TODOS** los elementos del array cumplen la condición y **false** en caso contrario. La segunda devuelve **true** si **ALGÚN** elemento del array cumple la condición. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let todosAprobados = arrayNotas.every(nota => nota >= 5) // false
let algunAprobado = arrayNotas.some(nota => nota >= 5) // true
```

## map

Permite modificar cada elemento de un array y devuelve un nuevo array con los elementos del original modificados. Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = arrayNotas.map(nota => nota + nota * 10%)
```

EJERCICIO: Dado un array con los días de la semana devuelve otro array con los días en mayúsculas

## reduce

Devuelve un valor calculado a partir de los elementos del array. En este caso la función recibe como primer parámetro el valor calculado hasta ahora y el método tiene como 1º parámetro la función y como 2º parámetro al valor calculado inicial (si no se indica será el primer elemento del array).

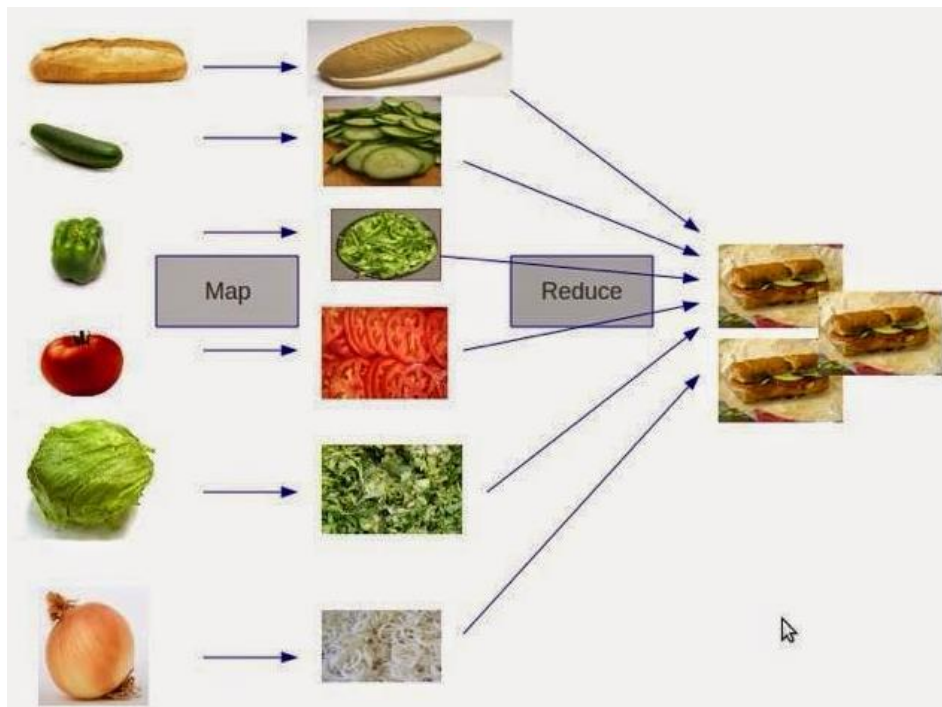
Ejemplo: queremos obtener la suma de las notas:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let sumaNotas = arrayNotas.reduce((total,nota) => total + = nota, 0) // total = 35.35
// podríamos haber omitido el valor inicial 0 para total
let sumaNotas = arrayNotas.reduce((total,nota) => total + = nota) // total = 35.35
```

Ejemplo: queremos obtener la nota más alta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota) => nota > max ? nota : max) // max = 9.75
```

En el siguiente ejemplo gráfico tenemos un “array” de verduras al que le aplicamos una función *map* para que las corte y al resultado le aplicamos un *reduce* para que obtenga un valor (el sándwich) con todas ellas:



EJERCICIO: Dado el array de notas anterior devuelve la nota media

## forEach

Es el método más general de los que hemos visto. No devuelve nada sino que permite realizar algo con cada elemento del array.

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.forEach((nota, indice) => {
  console.log('El elemento de la posición ' + indice + ' es: ' + nota)
})
```

## includes

Devuelve **true** si el array incluye el elemento pasado como parámetro. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.includes(7.5)    // true
```

EJERCICIO: Dado un array con los días de la semana indica si algún día es el 'Martes'

## Array.from

Devuelve un array a partir de otro al que se puede aplicar una función de transformación (es similar a *map*). Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = Array.from(arrayNotas, nota => nota + nota * 10%)
```

Puede usarse para hacer una copia de un array, igual que *slice*:

```
let arrayA = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayB = Array.from(arrayA)
```

También se utiliza mucho para convertir colecciones en arrays y así poder usar los métodos de arrays que hemos visto. Por ejemplo, si queremos mostrar por consola cada párrafo de la página que comience por la palabra 'If' en primer lugar obtenemos todos los párrafos con:



```
let parrafos = document.getElementsByTagName('p')
```

Esto nos devuelve una colección con todos los párrafos de la página (lo veremos más adelante al ver DOM). Podríamos hacer un **for** para recorrer la colección y mirar los que empiecen por lo indicado pero no podemos aplicarle los métodos vistos aquí porque son sólo para arrays así que hacemos:

```
let arrayParrafos = Array.from(parrafos)
// y ya podemos usar los métodos que queramos:
arrayParrafos.filter(parrafo => parrafo.textContent.startsWith('If'))
.forEach(parrafo => alert(parrafo.textContent))
```

**IMPORTANTE:** desde este momento se han acabado los bucles *for* en nuestro código, siempre que sea posible, para trabajar con arrays. ¡¡¡Usaremos siempre estas funciones!!!

## Referencia vs Copia

Cuando copiamos una variable de tipo *boolean*, *string* o *number* o se pasa como parámetro a una función se hace una copia de la misma y si se modifica la variable original no es modificada. Ej.:

```
let a = 54
let b = a      // a = 54 b = 54
b = 86        // a = 54 b = 86
```

Sin embargo, al copiar objetos (y los arrays son un tipo de objeto) la nueva variable apunta a la misma posición de memoria que la antigua por lo que los datos de ambas son los mismos:

```
let a = [54, 23, 12]
let b = a      // a = [54, 23, 12] b = [54, 23, 12]
b[0] = 3       // a = [3, 23, 12] b = [3, 23, 12]
let fecha1 = new Date('2018-09-23')
```

```
let fecha2 = fecha1          // fecha1 = '2018-09-23'   fecha2 = '2018-09-23'
fecha2.setFullYear(1999)    // fecha1 = '1999-09-23'   fecha2 = '1999-09-23'
```

Si queremos obtener una copia de un array que sea independiente del original podemos obtenerla con `slice` o con `Array.from`:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()      // ahora ambos arrays contienen lo mismo pero son diferentes
let otraCopiaDeA = Array.from(a)
```

En el caso de objetos es algo más complejo. ES6 incluye `Object.assign` que hace una copia de un objeto:

```
let a = {id:2, name: 'object 2'}
let copiaDeA = Object.assign({}, a)    // ahora ambos objetos contienen lo mismo pero son diferentes
```

Sin embargo, si el objeto tiene como propiedades otros objetos estos se continúan pasando por referencia. Es ese caso lo más sencillo sería hacer:

```
let a = {id: 2, name: 'object 2', address: {street: 'C/ Ajo', num: 3} }
let copiaDeA = JSON.parse(JSON.stringify(a))    // ahora ambos objetos contienen lo mismo pero son diferentes
```

EJERCICIO: Dado el array `arr1` con los días de la semana haz un array `arr2` que sea igual al `arr1`. Elimina de `arr2` el último día y comprueba qué ha pasado con `arr1`. Repita la operación con un array llamado `arr3` pero que crearás haciendo una copia de `arr1`.

También podemos copiar objetos usando *rest* y *spread*.

## Rest y Spread

Permiten extraer a parámetros los elementos de un array o string (*spread*) o convertir en un array un grupo de parámetros (*rest*). El operador de ambos es ... (3 puntos).

Para usar *rest* como parámetro de una función debe ser siempre el último parámetro.

Ejemplo: queremos hacer una función que calcule la media de las notas que se le pasen como parámetro y que no sabemos cuántas són. Para llamar a la función haremos:

```
console.log(notaMedia(3.6, 6.8))
console.log(notaMedia(5.2, 3.9, 6, 9.75, 7.5, 3))
```

La función convertirá los parámetros recibidos en un array usando *rest*:

```
function notaMedia(...notas) {
  let total = notas.reduce((total,nota) => total + = nota)
  return total/notas.length
}
```

Si lo que queremos es convertir un array en un grupo de elementos haremos *spread*. Por ejemplo, el objeto *Math* proporciona métodos para trabajar con números como *.max* que devuelve el máximo de los números pasados como parámetro. Para saber la nota máxima en vez de *.reduce* como hicimos en el ejemplo anterior podemos hacer:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]

let maxNota = Math.max(...arrayNotas)    // maxNota = 9.75
// si hacemos Math.max(arrayNotas) devuelve NaN porque arrayNotas es un array y no un número
```

Estas funcionalidades nos ofrecen otra manera de copiar objetos (pero sólo a partir de ES-2018):

```
let a = { id: 2, name: 'object 2' }
let copiaDeA = { ...a }      // ahora ambos objetos contienen lo mismo pero son diferentes

let b = [2, 8, 4, 6]
let copiaDeB = [ ...b ]     // ahora ambos objetos contienen lo mismo pero son diferentes
```

### Desestructuración de arrays

Similar a *rest* y *spread*, permiten extraer los elementos del array directamente a variables y viceversa. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let [primera, segunda, tercera] = arrayNotas // primera = 5.2, segunda = 3.9, tercera = 6
let [primera, , , cuarta] = arrayNotas      // primera = 5.2, cuarta = 9.75
let [primera, ...resto] = arrayNotas        // primera = 5.2, resto = [3.9, 6, 9.75, 3]
```

También se pueden asignar valores por defecto:

```
let preferencias = ['Javascript', 'NodeJS']
let [lenguaje, backend = 'Laravel', frontend = 'VueJS'] = preferencias // lenguaje = 'Javascript',
backend = 'NodeJS', frontend = 'VueJS'
```

La desestructuración también funciona con objetos. Es normal pasar un objeto como parámetro para una función, pero si sólo nos interesan algunas propiedades del mismo podemos desestructurarlo:

```
const miProducto = {
  id: 5,
```

```

    name: 'TV Samsung',
    units: 3,
    price: 395.95
  }

  muestraNombre(miProducto)

  function miProducto({name, units}) {
    console.log('Del producto ' + name + ' hay ' + units + ' unidades')
  }

```

También podemos asignar valores por defecto:

```

function miProducto({name, units = 0}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre({name: 'USB Kingston'})
// mostraría: Del producto USB Kingston hay 0 unidades

```

## Map

Es una colección de parejas de [clave,valor]. Un objeto en Javascript es un tipo particular de *Map* en que las claves sólo pueden ser texto o números. Se puede acceder a una propiedad con `.` o **[propiedad]**. Ejemplo:

```

let persona = {
  nombre: 'John',
  apellido: 'Doe',
  edad: 39
}

console.log(persona.nombre) // John
console.log(persona['nombre']) // John

```

Un *Map* permite que la clave sea cualquier cosa (array, objeto, ...). No vamos a ver en profundidad estos objetos pero podéis saber más en [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) o cualquier otra página.

## Set

Es como un *Map* pero que no almacena los valores sino sólo la clave. Podemos verlo como una colección que no permite duplicados. Tiene la propiedad **size** que devuelve su tamaño y los métodos **.add** (añade un elemento), **.delete** (lo elimina) o **.has** (indica si el elemento pasado se encuentra o no en la colección) y también podemos recorrerlo con **.forEach**.

Una forma sencilla de eliminar los duplicados de un array es crear con él un *Set*:

```
let ganadores = ['Márquez', 'Rossi', 'Márquez', 'Lorenzo', 'Rossi', 'Márquez', 'Márquez']
let ganadoresNoDuplicados = new Set(ganadores) // {'Márquez', 'Rossi', 'Lorenzo'}
// o si lo queremos en un array:
let ganadoresNoDuplicados = Array.from(new Set(ganadores)) // ['Márquez', 'Rossi', 'Lorenzo']
```

## Programación orientada a Objetos en Javascript

### Introducción

La **Programación Orientada a Objetos (POO)** es un [paradigma de programación](#), es decir, un modelo o un estilo de programación que nos da unas guías sobre cómo trabajar con él. Se basa en el **concepto de clases y objetos**. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos.

Desde ES2015 la POO en Javascript es similar a como se hace en otros lenguajes, con clases, herencia, ...:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre
    this.apellidos = apellidos
    this.edad = edad
  }
  getInfo() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
  }
}

let alumno1 = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(alumno1.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

EJERCICIO: Crea una clase `Productos` con las propiedades *name*, *category*, *units* y *price* y los métodos *total* que devuelve el importe del producto y *getInfo* que devolverá: *'Name (category): units uds x price € = total €'*. Crea 3 productos diferentes.

## Herencia

Una clase puede heredar de otra utilizando la palabra reservada **extends** y heredará todas sus propiedades y métodos. Podemos sobrescribirlos en la clase hija (seguimos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super** -es lo que haremos si creamos un constructor en la clase hija-).

```
class AlumnInf extends Alumno{
  constructor(nombre, apellidos, edad, ciclo) {
    super(nombre, apellidos, edad)
    this.ciclo = ciclo
  }
  getInfo() {
    return super.getInfo() + ' y estudia el Grado ' + (this.getGradoMedio() ? 'Medio' : 'Superior') +
    ' de ' + this.ciclo
  }
  getGradoMedio() {
    if (this.ciclo.toUpperCase() === 'SMX')
      return true
    return false
  }
}

let cpo = new AlumnInf('Carlos', 'Pérez Ortiz', 19, 'DAW')
console.log(cpo.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años y estudia el
Grado Superior de DAW'
```

EJERCICIO: crea una clase `Televisores` que hereda de `Productos` y que tiene una nueva propiedad llamada *tamaño*. El método *getInfo* mostrará el tamaño junto al nombre

## Métodos estáticos

Desde ES2015 podemos declarar métodos estáticos, pero no propiedades estáticas. Estos métodos se llaman directamente utilizando el nombre de la clase y no tienen acceso al objeto *this* (ya que no hay objeto instanciado).

```

class User {
  ...
  static getRoles() {
    return ["user", "guest", "admin"]
  }
}

console.log(User.getRoles()) // ["user", "guest", "admin"]
let user = new User("john")
console.log(user.getRoles()) // Uncaught TypeError: user.getRoles is not a function

```

Suelen usarse para crear funciones de la aplicación.

## Método toString()

Al convertir un objeto a string (por ejemplo, al concatenarlo con un String) se llama al método `toString()` del mismo, que por defecto devuelve la cadena `[object object]`. Podemos sobrecargar este método para que devuelva lo que queramos:

```

class Alumno {
  ...
  toString() {
    return this.apellidos + ', ' + this.nombre
  }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
console.log('Alumno: ' + cpo) // imprime 'Alumno: Pérez Ortiz, Carlos'
// en vez de 'Alumno: [object Object]'

```

Este método también es el que se usará si queremos ordenar un array de objetos (recordad que `sort()` ordena alfabéticamente para lo que llama al método `.toString()` del objeto a ordenar). Por ejemplo, tenemos el array de alumnos *misAlumnos* que queremos ordenar alfabéticamente. Si la clase *Alumno* no tiene un método *toString* habría que hacer como vimos en el tema de Arrays:

```

misAlumnos.sort(function(alum1, alum2) {
  if (alum1.apellidos > alum2.apellidos)
    return -1

```



```
    if (alum1.apellidos < alum2.apellidos)
        return 1
    return alum1.nombre < alum2.nombre
});
```

Pero con el método *toString* que hemos definido antes podemos hacer directamente:

```
misAlumnos.sort()
```

**NOTA:** si las cadenas a comparar pueden tener acentos u otros caracteres propios del idioma ese código no funcionará bien. La forma correcta de comparar cadenas es usando el método *.localeCompare()*. El código anterior debería ser:

```
misAlumnos.sort(function(alum1, alum2) {
    return alum1.apellidos.localeCompare(alum2.apellidos)
});
```

que con *arrow function* quedaría:

```
misAlumnos.sort((alum1, alum2) => alum1.apellidos.localeCompare(alum2.apellidos))
```

o si queremos comparar por 2 campos ('apellidos' y 'nombre')

```
misAlumnos.sort((alum1, alum2) =>
    (alum1.apellidos+alum1.nombre).localeCompare(alum2.apellidos+alum2.nombre) )
```

**NOTA:** si queremos ordenar un array de objetos por un campo numérico lo mas sencillo es restar dicho campo:

```
misAlumnos.sort((alum1, alum2) => alum1.edad - alum2.edad)
```

EJERCICIO: modifica las clases Productos y Televisores para que el método que muestra los datos del producto se llame de la manera más adecuada

EJERCICIO: Crea 5 productos y guárdalos en un array. Crea las siguientes funciones (todas reciben ese array como parámetro):

- prodsSortByName: devuelve un array con los productos ordenados alfabéticamente.
- prodsSortByPrice: devuelve un array con los productos ordenados por importe.
- prodsTotalPrice: devuelve el importe total de los productos del array, con 2 decimales.
- prodsWithLowUnits: además del array recibe como segundo parámetro un nº y devuelve un array con todos los productos de los que quedan menos de las unidades indicadas.
- prodsList: devuelve una cadena que dice 'Listado de productos:' y en cada línea un guión y la información de un producto del array.

## Método valueOf()

Al comparar objetos (con >, <, ...) se usa el valor devuelto por el método **.valueOf()** para realizar la comparación. Si este método no existiera sería **.toString()** el que se usaría.:

```
class Alumno {  
  ...  
  valueOf() {  
    return this.edad  
  }  
}  
  
let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)  
let aat = new Alumno('Ana', 'Abad Tudela', 23)  
console.log(cpo < aat) // imprime true ya que 19<23
```

## Organizar el código

Lo más conveniente es guardar cada clase en su propio fichero, que llamaremos como la clase con la extensión `.class.js`. Por ejemplo el fichero de la clase *Users* sería `users.class.js`.

En dicho fichero exportamos la clase (con `export` o `export default` porque sólo hay 1) y donde queramos usarla la importamos (`import { Users } from 'users.class'` o `import Users from 'users.class'`, según cómo la hayamos exportado).

## Ojo con this

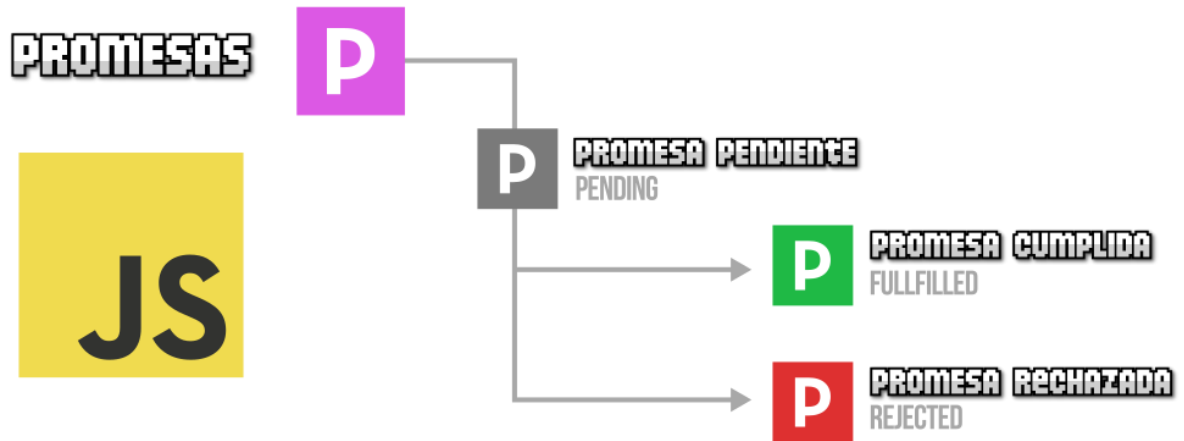
Dentro de una función se crea un nuevo contexto y la variable *this* pasa a hacer referencia a dicho contexto. Si en el ejemplo anterior hiciéramos algo como esto:

```
class Alumno {  
  ...  
  getInfo() {  
    return 'El alumno ' + nomAlum() + ' tiene ' + this.edad + ' años'  
    function nomAlum() {  
      return this.nombre + ' ' + this.apellidos // Aquí this no es el objeto Alumno  
    }  
  }  
}
```

Este código fallaría porque dentro de la función *nomAlum* la variable *this* ya no hace referencia al objeto *Alumno* sino al contexto de la función. Este ejemplo no tiene mucho sentido pero a veces nos pasará en manejadores de eventos.

Si debemos llamar a una función dentro de un método (o de un manejador de eventos) tenemos varias formas de pasarle el valor de *this*:

## PROMESAS



### ¿Qué es una promesa en JavaScript?

Las promesas en JavaScript no solo representan el resultado de una operación asíncronica, sino que también proporcionan métodos que facilitan el manejo y la manipulación de los datos una vez que la promesa se resuelve.

Una promesa es un objeto que representa un valor que puede que esté disponible «ahora», en un «futuro» o que «nunca» lo esté. Como no se sabe cuándo va a estar disponible, todas las operaciones dependientes de ese valor tendrán que posponerse en el tiempo.

### Métodos de las promesas en JavaScript

Aquí están algunos de los métodos más comunes que puedes utilizar:

- **.then():** Este método se utiliza para manejar el resultado exitoso de una promesa. Recibe una función que se ejecutará cuando la promesa se resuelva con éxito y puede recibir el resultado como argumento.
- **.catch():** Se utiliza para manejar errores que puedan ocurrir durante la ejecución de la promesa. Puedes encadenar `.catch()` después de `.then()` para manejar errores específicos.
- **.finally():** Este método se utiliza para ejecutar una función después de que la promesa se resuelva o se rechace, independientemente del resultado. Es útil para realizar tareas de limpieza o acciones que deben ocurrir sin importar el resultado de la promesa.
- **Promise.all(iterable):** Este método permite manejar múltiples promesas al mismo tiempo y resuelve una promesa una vez que todas las promesas del iterable se hayan resuelto o alguna de ellas se haya rechazado.

- **Promise.race(iterable):** Este método resuelve una promesa tan pronto como una de las promesas en el iterable se resuelva o se rechace. Es útil cuando deseas obtener el resultado más rápido de múltiples promesas.

## Estados de las promesas en JavaScript

Una promesa puede estar en los siguientes tres estados:

- Pendiente (**pending**). Es el estado inicial al crear una promesa.
- Resuelta con éxito (**fulfilled**). Estará resuelta en el momento que llamemos a `resolve` y, a continuación, se ejecutará la función que pasamos al método `.then`. Debemos de tener en cuenta que, una vez resuelta, no podremos modificar el valor de la promesa, aunque sí podríamos correr la misma instrucción para obtener un valor distinto y hacerlo las veces que deseemos.
- Rechazada (**rejected**). También puede ocurrir que se complete pero sea rechazada por un error, pasando a continuación a ejecutar la función que pasamos a `.catch`.

## Cómo se crea una promesa en JavaScript

Se puede crear una promesa con el constructor `promise` y pasándole una función con dos parámetros: `resolve` y `reject`, que nos deja decirle si ha sido resuelta o rechazada.

```
const promise = new Promise((resolve, reject) => {  
  const number = Math.floor(Math.random() * 12);  
  setTimeout(  
    () => (number > 4 ? resolve(number) : reject(new Error("Menor a 4"))),  
    2000  
  );  
});  
promise  
  .then((number) => console.log(number))  
  .catch((error) => console.error(error));
```

En el ejemplo, hemos creado una promesa que se completará en **2 segundos**. Si el número aleatorio que hemos generado es **mayor a 4**, se resolverá; en caso contrario, se rechaza y se muestra un error.

Declaramos otra promesa:

---

```
const myPromise = new Promise(function (resolve, reject) {
  let sampleData = [2, 4, 6, 8];
  let randomNumber = Math.floor(Math.random() * (sampleData.length + 1));
  console.log("sampledata.length + 1: " + (sampleData.length+1))
  console.log("randomnumber: "+randomNumber)
  if (sampleData[randomNumber]) {
    resolve(sampleData[randomNumber]);
  } else {
    reject('An error occurred!');
  }
});
```

En este caso la promesa crea un array de cuatro elementos numéricos que se imprimen si el número generado aleatoriamente en la propia promesa coincide con algún índice del array sampleData, si no coincide imprime un error.

```
myPromise
  .then(function (e) {
    console.log(e);
  })
  .catch(function (error) {
    throw new Error(error);
  })
  .finally(function () {
    console.log('Promise completed');
  });
```

## Cómo implementar las promesas en JavaScript: más ejemplos

### Ejemplo de transacción de pago en línea

Supongamos que deseamos simular una transacción de pago en línea y queremos asegurarnos de que se manejen correctamente tanto los pagos exitosos como los fallidos, utilizando promesas.

```
// Simulación de una función que procesa un pago
function procesarPago(total) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const exito = Math.random() < 0.8; // Simulamos que el 80 % de las veces el pago es exitoso

      if (exito) {
        resolve("Pago exitoso");
      }
    }, 1000);
  });
}
```

```

    } else {
      reject(new Error("Error en el pago"));
    }
  }, 1500); // Simulamos un retraso de 1.5 segundos para el proceso de pago
});
}

// Ejemplo de uso de la función de procesamiento de pagos
const totalAPagar = 100; // Monto total del pago

procesarPago(totalAPagar)
  .then(resultado => {
    console.log(resultado); // Se ejecutará en caso de pago exitoso
    // Aquí podríamos realizar acciones adicionales, como actualizar la base de datos, enviar un
    recibo por correo, etc.
  })
  .catch(error => {
    console.error(error.message); // Se ejecutará en caso de pago fallido
    // Aquí podemos ofrecer al usuario la opción de intentar nuevamente o mostrar un mensaje de error
  });

```

En este ejemplo, la función *procesarPago()* simula una transacción de pago en línea. Dependiendo de un valor aleatorio, la función decide si el pago es exitoso o fallido. La promesa que se devuelve se cumple si el pago es exitoso (el 80 % de las veces) y se rechaza si el pago falla.

Luego, utilizamos *.then()* para manejar el caso de pago exitoso, donde podemos realizar acciones adicionales, como actualizar la base de datos o enviar un recibo por correo electrónico. Empleamos *.catch()* para manejar el caso de pago fallido, donde es posible ofrecer al usuario opciones para resolver el problema o mostrar un mensaje de error.

## Ejemplo de base de datos

En este caso, supondremos que estamos trabajando con datos de usuarios y queremos simular la adición de múltiples usuarios a una base de datos, teniendo en cuenta casos exitosos y fallidos.

```

// Simulación de una función que agrega usuarios a una base de datos
function agregarUsuarios(usuarios) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const exito = Math.random() < 0.7; // Simulamos que el 70 % de las veces la operación es exitosa

      if (exito) {

```

```

        resolve("Usuarios agregados exitosamente");
    } else {
        reject(new Error("Error al agregar usuarios"));
    }
}, 2000); // Simulamos un retraso de 2 segundos para la operación de almacenamiento
});
}

// Datos de ejemplo de usuarios a agregar a la base de datos
const nuevosUsuarios = [
    { id: 1, nombre: "Usuario 1" },
    { id: 2, nombre: "Usuario 2" },
    { id: 3, nombre: "Usuario 3" }
];

// Uso de la función de agregación de usuarios
agregarUsuarios(nuevosUsuarios)
    .then(resultado => {
        console.log(resultado); // Se ejecutará en caso de operación exitosa
        // Aquí podríamos realizar acciones adicionales, como actualizar la interfaz de usuario o
        // notificar al usuario
    })
    .catch(error => {
        console.error(error.message); // Se ejecutará en caso de operación fallida
        // Aquí podríamos ofrecer al usuario la opción de intentar nuevamente o mostrar un mensaje de
        // error
    });

```

- **Definición de la función agregarUsuarios():** esta función simula el proceso de agregar usuarios a una base de datos. Devuelve una promesa que se cumple si la operación es exitosa (el 70 % de las veces) y se rechaza si la operación falla. La función toma un arreglo de usuarios como argumento.
- **Simulación de retardo:** para simular el tiempo que puede llevar almacenar datos en una base de datos, utilizamos setTimeout() para introducir un retraso de 2 segundos.
- **Uso de la función de agregación de usuarios:** pasamos el arreglo nuevosUsuarios a la función agregarUsuarios(). Luego utilizamos .then() para manejar el caso de éxito, donde podríamos realizar acciones adicionales, y .catch() para manejar el caso de falla, donde podríamos mostrar un mensaje de error.
- **Datos de ejemplo de usuarios:** creamos un arreglo de objetos que representan a los usuarios que deseamos agregar a la base de datos.

## DOCUMENT OBJECT MODEL (DOM)



## Introducción

La mayoría de las veces que programamos con JavaScript es para que se ejecute en una página web mostrada por el navegador. En este contexto tenemos acceso a ciertos objetos que nos permiten interactuar con la página (DOM) y con el navegador (Browser Object Model, BOM).

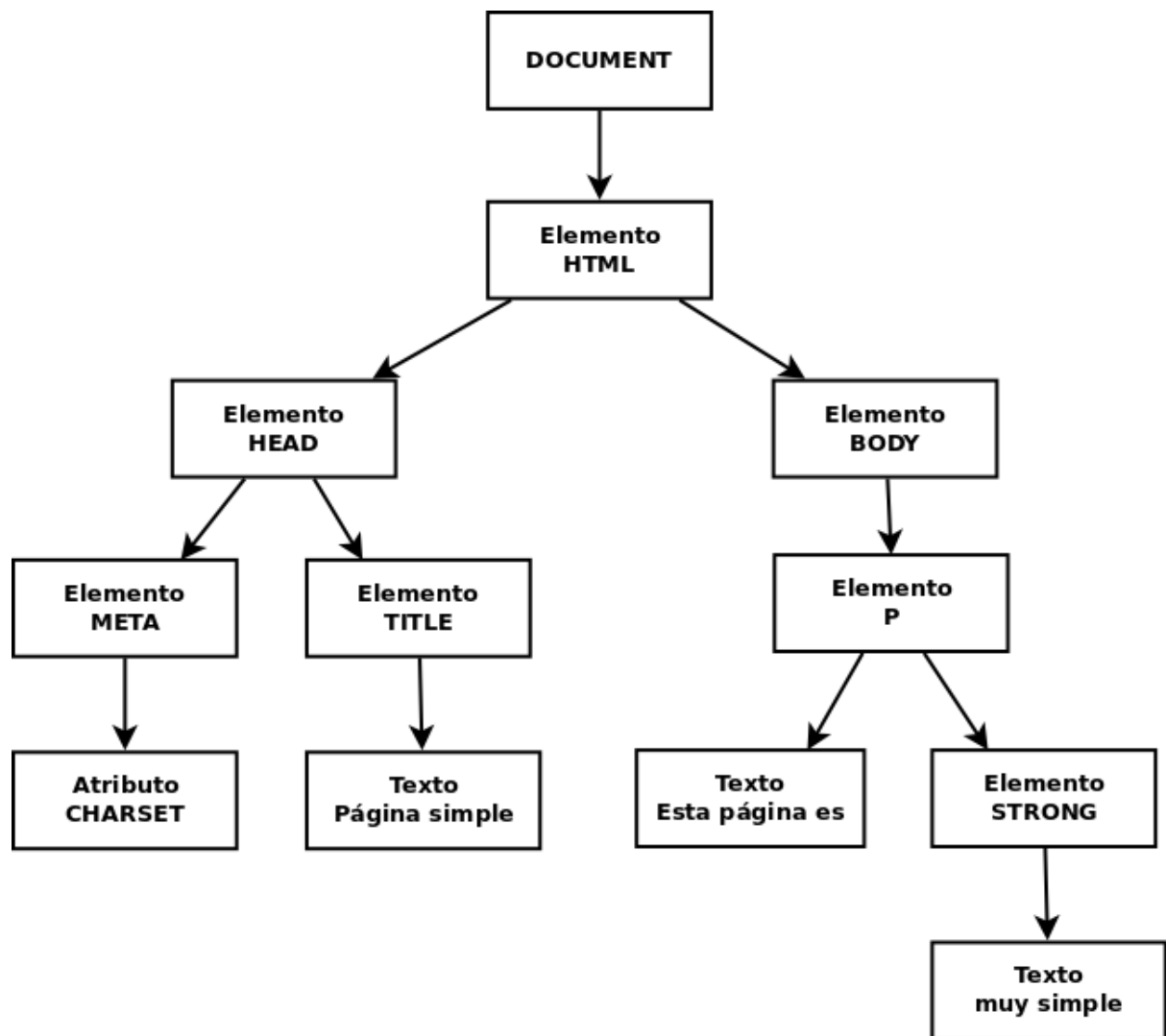
El **DOM** es una estructura en árbol que representa todos los elementos HTML de la página y sus atributos. Todo lo que contiene la página se representa como nodos del árbol y mediante el DOM podemos acceder a cada nodo, modificarlo, eliminarlo o añadir nuevos nodos de forma que cambiamos dinámicamente la página mostrada al usuario.

La raíz del árbol DOM es ***document*** y de este nodo cuelgan el resto de elementos HTML. Cada uno constituye su propio nodo y tiene subnodos con sus *atributos*, *estilos* y elementos HTML que contiene.

Por ejemplo, la página HTML:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Página simple</title>
</head>
<body>
  <p>Esta página es <strong>muy simple</strong></p>
</body>
</html>
```

se convierte en el siguiente árbol DOM:



Cada etiqueta HTML suele originar 2 nodos:

- **Element**: correspondiente a la etiqueta
- **Text**: correspondiente a su contenido (lo que hay entre la etiqueta y su par de cierre)

Cada nodo es un objeto con sus propiedades y métodos.

El ejemplo anterior está simplificado porque sólo aparecen los nodos de tipo **elemento** pero en realidad también generan nodos los saltos de línea, tabuladores, espacios, comentarios, etc. En el siguiente ejemplo podemos ver TODOS los nodos que realmente se generan. La página:

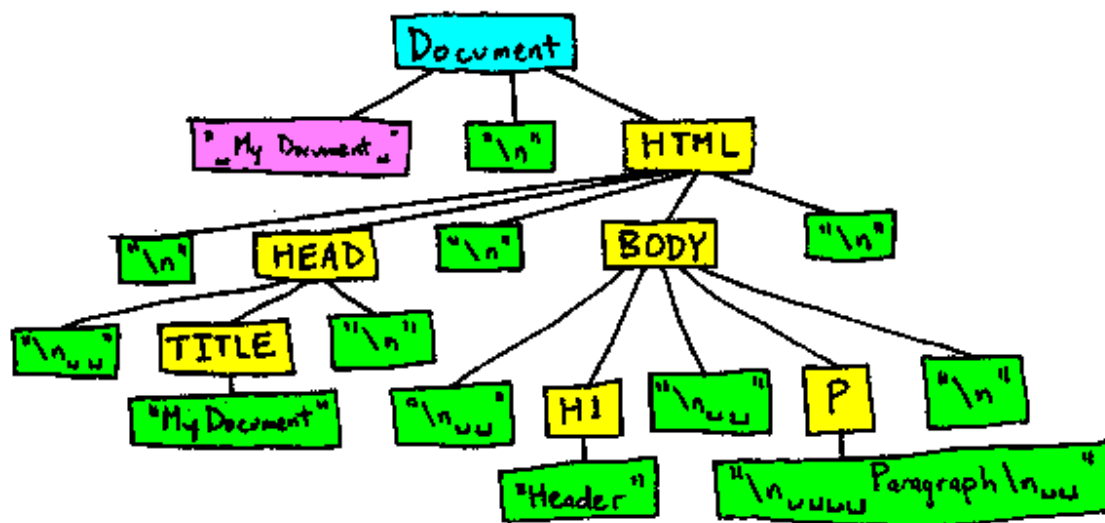
```
<!DOCTYPE html>
<html>
<head>
  <title>My Document</title>
</head>
<body>
```

```

<h1>Header</h1>
<p>
  Paragraph
</p>
</body>
</html>

```

se convierte en el siguiente árbol DOM:



## Acceso a los nodos

Los principales métodos para acceder a los diferentes nodos son:

- **.getElementById(id)**: devuelve el nodo con la id pasada. Ej.:

```

let nodo = document.getElementById('main'); // nodo contendrá el nodo cuya id es _main_

```

- **.getElementsByClassName(clase)**: devuelve una colección (similar a un array) con todos los nodos de la *clase* indicada. Ej.:

```

let nodos = document.getElementsByClassName('error'); // nodos contendrá todos los nodos cuya
clase es _error_

```

NOTA: las colecciones son similares a arrays (se accede a sus elementos con *[índice]*) pero no se les pueden aplicar sus métodos *filter*, *map*, ... a menos que se conviertan a arrays con *Array.from()*

- **.getElementsByTagName(etiqueta)**: devuelve una colección con todos los nodos de la *etiqueta* HTML indicada. Ej.:

```
let nodos = document.getElementsByTagName('p'); // nodos contendrá todos los nodos de tipo _<p>_
```

- **.getElementsByName(name)**: devuelve una colección con todos los nodos que contengan un atributo name con el valor indicado. Ej.:

```
let radiosSexo = document.getElementsByName('sexo'); // radiosSexo contendrá todos los nodos con ese atributo (seguramente radiobuttons con name="sexo")
```

- **.querySelector(selector)**: devuelve el primer nodo seleccionad por el *selector* CSS indicado. Ej.:

```
let nodo = document.querySelector('p.error'); // nodo contendrá el primer párrafo de clase _error_
```

- **.querySelectorAll(selector)**: devuelve una colección con todos los nodos seleccionados por el *selector* CSS indicado. Ej.:

```
let nodos = document.querySelectorAll('p.error'); // nodos contendrá todos los párrafos de clase _error_
```

NOTA: al aplicar estos métodos sobre **document** se seleccionará sobre la página pero podrían también aplicarse a cualquier nodo y en ese caso la búsqueda se realizaría sólo entre los descendientes de dicho nodo.

También tenemos 'atajos' para obtener algunos elementos comunes:

- **document.documentElement**: devuelve el nodo del elemento `<html>`

- **document.head:** devuelve el nodo del elemento `<head>`
- **document.body:** devuelve el nodo del elemento `<body>`
- **document.title:** devuelve el nodo del elemento `<title>`
- **document.link:** devuelve una colección con todos los hipervínculos del documento
- **document.anchor:** devuelve una colección con todas las anclas del documento
- **document.forms:** devuelve una colección con todos los formularios del documento
- **document.images:** devuelve una colección con todas las imágenes del documento
- **document.scripts:** devuelve una colección con todos los scripts del documento

EJERCICIO: Para hacer los ejercicios de este tema descárgate [esta página de ejemplo](#) y ábrela en tu navegador. Obtén por consola, al menos de 2 formas diferentes:

- El elemento con id 'input2'
- La colección de párrafos
- La colección de párrafos pero sólo de los párrafos que hay dentro del div 'lipsum'
- El formulario (ojo, no la colección con el formulario sino sólo el formulario)
- Todos los inputs
- Sólo los inputs con nombre 'sexo'
- Los items de lista de la clase 'important' (sólo los LI)

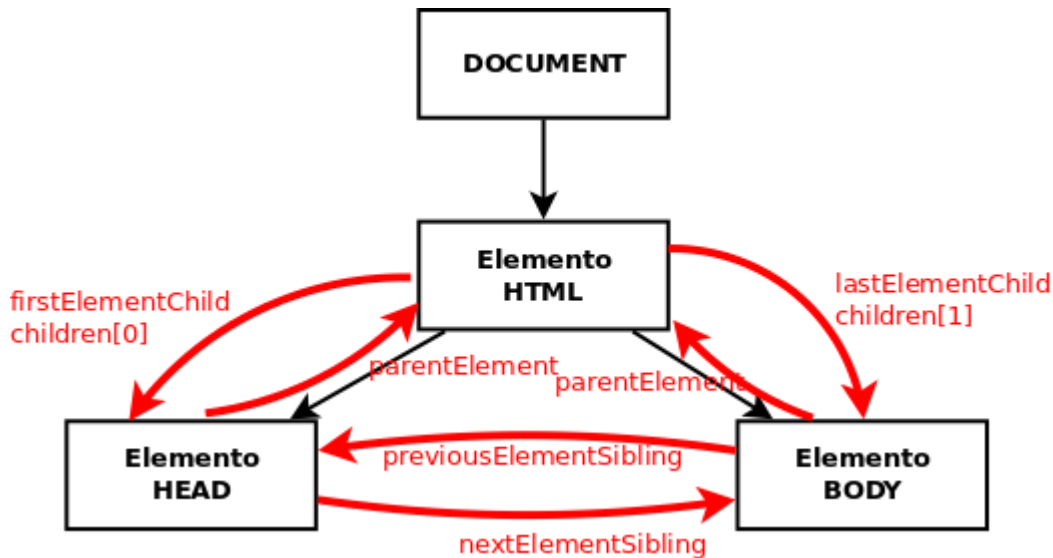
## Acceso a nodos a partir de otros

En muchas ocasiones queremos acceder a cierto nodo a partir de uno dado. Para ello tenemos los siguientes métodos que se aplican sobre un elemento del árbol DOM:

- **elemento.parentElement:** devuelve el elemento padre de elemento.
- **elemento.children:** devuelve la colección con todos los elementos hijo de elemento (sólo elementos HTML, no comentarios ni nodos de tipo texto)
- **elemento.childNodes:** devuelve la colección con todos los hijos de elemento, incluyendo comentarios y nodos de tipo texto por lo que no suele utilizarse.
- **elemento.firstElementChild:** devuelve el elemento HTML que es el primer hijo de elemento.
- **elemento.firstChild:** devuelve el nodo que es el primer hijo de elemento (incluyendo nodos de tipo texto o comentarios)
- **elemento.lastElementChild, elemento.lastChild:** igual pero con el último hijo.
- **elemento.nextElementSibling:** devuelve el elemento HTML que es el siguiente hermano de elemento.
- **elemento.nextSibling:** devuelve el nodo que es el siguiente hermano de elemento (incluyendo nodos de tipo texto o comentarios)
- **elemento.previousElementSibling, elemento.previousSibling:** igual pero con el hermano anterior.

- ***elemento.hasChildNodes***: indica si elemento tiene o no nodos hijos.
- ***elemento.childElementCount***: devuelve el número de nodos hijo de elemento.

IMPORTANTE: a menos que me interesen comentarios, saltos de página, etc. siempre debo usar los métodos que sólo devuelven elementos HTML, no todos los nodos.



EJERCICIO: Siguiendo con la [página de ejemplo](#) obtén desde la consola, al menos de 2 formas diferentes:

- El primer párrafo que hay dentro del div 'lipsum'
- El segundo párrafo de 'lipsum'
- El último ítem de la lista
- La label de 'Escoge sexo'

## Propiedades de un nodo

Las principales propiedades de un nodo son:

- ***elemento.innerHTML***: todo lo que hay entre la etiqueta que abre elemento y la que lo cierra, incluyendo otras etiquetas HTML. Por ejemplo si elemento es el nodo `<p>Esta página es <strong>muy simple</strong></p>`

```
let contenido = elemento.innerHTML; // contenido='Esta página es <strong>muy simple</strong>'
```

- **elemento.textContent**: todo lo que hay entre la etiqueta que abre elemento y la que lo cierra, pero ignorando otras etiquetas HTML. Siguiendo con el ejemplo anterior:

```
let contenido = elemento.textContent; // contenido='Esta página es muy simple'
```

- **elemento.value**: devuelve la propiedad 'value' de un <input> (en el caso de un <input> de tipo text devuelve lo que hay escrito en él). Como los <inputs> no tienen etiqueta de cierre (</input>) no podemos usar .innerHTML ni .textContent. Por ejemplo si elem1 es el nodo <input name="nombre"> y elem2 es el nodo <input type="radio" value="H">Hombre

```
let cont1 = elem1.value; // cont1 valdría lo que haya escrito en el <input> en ese momento
let cont2 = elem2.value; // cont2="H"
```

Otras propiedades:

- **elemento.innerText**: igual que *textContent*
- **elemento.focus**: da el foco a elemento (para inputs, etc.). Para quitarle el foco *elemento.blur*
- **elemento.clientHeight** / **elemento.clientWidth**: devuelve el alto / ancho visible del elemento
- **elemento.offsetHeight** / **elemento.offsetWidth**: devuelve el alto / ancho total del elemento
- **elemento.clientLeft** / **elemento.clientTop**: devuelve la distancia de elemento al borde izquierdo / superior
- **elemento.offsetLeft** / **elemento.offsetTop**: devuelve los píxeles que hemos desplazado elemento a la izquierda / abajo

EJERCICIO: Obtén desde la consola, al menos de 2 formas:

- El innerHTML de la etiqueta de 'Escoge sexo'
- El textContent de esa etiqueta
- El valor del primer input de sexo
- El valor del sexo que esté seleccionado (difícil, búscalo por Internet)

## Manipular el árbol DOM

Vamos a ver qué métodos nos permiten cambiar el árbol DOM, y por tanto modificar la página:

- **document.createElement('etiqueta')**: crea un nuevo elemento HTML con la etiqueta indicada, pero aún no se añade a la página. Ej.:

```
let nuevoLi = document.createElement('li');
```

- ***document.createTextNode('texto')***: crea un nuevo nodo de texto con el texto indicado, que luego tendremos que añadir a un nodo HTML. Ej.:

```
let textoLi = document.createTextNode('Nuevo elemento de lista');
```

- ***elemento.appendChild(nuevoNodo)***: añade *nuevoNodo* como último hijo de elemento. Ahora ya se ha añadido a la página. Ej.:

```
nuevoLi.appendChild(textoLi); // añade el texto creado al elemento LI creado
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
miPrimeraLista.appendChild(nuevoLi); // añade LI como último hijo de UL, es decir al final de la lista
```

- ***elemento.insertBefore(nuevoNodo, nodo)***: añade *nuevoNodo* como hijo de elemento antes del hijo *nodo*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona el 1º LI de miPrimeraLista
miPrimeraLista.insertBefore(nuevoLi, primerElementoDeLista); // añade LI al principio de la lista
```

- ***elemento.removeChild(nodo)***: borra nodo de elemento y por tanto se elimina de la página. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona el 1º LI de miPrimeraLista
miPrimeraLista.removeChild(primerElementoDeLista); // borra el primer elemento de la lista
// También podríamos haberlo borrado sin tener el padre con:
primerElementoDeLista.parentElement.removeChild(primerElementoDeLista);
```

- ***elemento.replaceChild(nuevoNodo, viejoNodo)***: reemplaza *viejoNodo* con *nuevoNodo* como hijo de elemento. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona el 1º LI de miPrimeraLista
```



```
miPrimeraLista.replaceChild(nuevoLi, primerElementoDeLista); // reemplaza el 1º elemento de la lista con nuevoLi
```

- ***elementoAClonar.cloneNode(boolean)***: devuelve un clon de *elementoAClonar* o de *elementoAClonar* con todos sus descendientes según le pasemos como parámetro false o true. Luego podremos insertarlo donde queramos.

OJO: Si añado con el método *appendChild* un nodo que estaba en otro sitio se elimina de donde estaba para añadirse a su nueva posición. Si quiero que esté en los 2 sitios deberé clonar el nodo y luego añadir el clon y no el nodo original.

**Ejemplo de creación de nuevos nodos:** tenemos un código HTML con un DIV que contiene 3 párrafos y vamos a añadir un nuevo párrafo al final del div con el texto 'Párrafo añadido al final' y otro que sea el 2º del div con el texto 'Este es el nuevo segundo párrafo':

HTML inicial:

```
<div id="articulos">
  <p>
    Este es el primer párrafo que tiene <strong>algo en negrita</strong>.
  </p>
  <p>Este era el segundo párrafo pero será desplazado hacia abajo.</p>
  <p>Y este es el último párrafo pero luego añadiremos otro después</p>
</div>
```

Ejemplo:

```
let miDiv=document.getElementById('articulos');

let ultimoParrafo=document.createElement('p');
let ultimoParrafoTexto=document.createTextNode('Párrafo añadido al final');
ultimoParrafo.appendChild(ultimoParrafoTexto);
miDiv.appendChild(ultimoParrafo);

let nuevaNegrita=document.createElement('strong');
nuevaNegritaTexto=document.createTextNode('nuevo');
nuevaNegrita.appendChild(nuevaNegritaTexto);
let nuevoSegundoParrafo=document.createElement('p');
let nuevoSegundoParrafoTexto1=document.createTextNode('Este es el ');
let nuevoSegundoParrafoTexto2=document.createTextNode(' segundo párrafo');
nuevoSegundoParrafo.appendChild(nuevoSegundoParrafoTexto1);
nuevoSegundoParrafo.appendChild(nuevaNegrita);
nuevoSegundoParrafo.appendChild(nuevoSegundoParrafoTexto2);

let segundoParrafo=miDiv.children[1];
```

```
miDiv.insertBefore(nuevoSegundoParrafo, segundoParrafo);
```

Si utilizamos la propiedad **innerHTML** el código a usar es mucho más simple:

```
let miDiv=document.getElementById('articulos');

miDiv.innerHTML+='<p>Párrafo añadido al final</p>';

let nuevoSegundoParrafo=document.createElement('p');
nuevoSegundoParrafo.innerHTML='Este es el <strong>nuevo</strong> segundo párrafo';

let segundoParrafo=miDiv.children[1];
miDiv.insertBefore(nuevoSegundoParrafo, segundoParrafo);
```

OJO: La forma de añadir el último párrafo (línea #3: `miDiv.innerHTML+='<p>Párrafo añadido al final</p>';`) aunque es válida no es muy eficiente ya que obliga al navegador a volver a pintar TODO el contenido de `miDiv`. La forma correcta de hacerlo sería:

```
let ultimoParrafo = document.createElement('p');
ultimoParrafo.innerHTML = 'Párrafo añadido al final';
miDiv.appendChild(ultimoParrafo);
```

Así sólo debe repintar el párrafo añadido, conservando todo lo demás que tenga `miDiv`.

Podemos ver más ejemplos de creación y eliminación de nodos en [W3Schools](https://www.w3schools.com/js/default.asp).

EJERCICIO: Añade a la página de ejemplo de manejo del DOM:

- Un nuevo párrafo al final del DIV *'lipsum'* con el texto "Nuevo párrafo **añadido** por javascript" (fíjate que una palabra está en negrita)
- Un nuevo elemento al formulario tras el *'Dato 1'* con la etiqueta *'Dato 1 bis'* y el INPUT con id *'input1bis'* que al cargar la página tendrá escrito "Hola"

## Modificar el DOM con ChildNode

Childnode es una interfaz que permite manipular del DOM de forma más sencilla pero no está soportada en los navegadores Safari de IOS. Incluye los métodos:

- **elemento.before(nuevoNodo)**: añade el *nuevoNodo* pasado antes del nodo elemento
- **elemento.after(nuevoNodo)**: añade el *nuevoNodo* pasado después del nodo elemento

- ***elemento.replaceWith(nuevoNodo)***: reemplaza el nodo elemento con el nuevoNodo pasado
- ***elemento.remove()***: elimina el nodo elemento

## Atributos de los nodos

Podemos ver y modificar los valores de los atributos de cada elemento HTML y también añadir o eliminar atributos:

- ***elemento.attributes***: devuelve un array con todos los atributos de elemento
- ***elemento.hasAttribute('nombreAtributo')***: indica si elemento tiene o no definido el atributo nombreAtributo
- ***elemento.getAttribute('nombreAtributo')***: devuelve el valor del atributo *nombreAtributo* de elemento. Para muchos elementos este valor puede directamente con *elemento.atributo*.
- ***elemento.setAttribute('nombreAtributo', 'valor')***: establece valor como nuevo valor del atributo nombreAtributo de elemento. También puede cambiarse el valor directamente con *elemento.atributo=valor*.
- ***elemento.removeAttribute('nombreAtributo')***: elimina el atributo *nombreAtributo* de elemento

A algunos atributos comunes como *id*, *title* o *className* (para el atributo *class*) se puede acceder y cambiar como si fueran una propiedad del elemento (*elemento.atributo*). Ejemplos:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
miPrimeraLista.id = 'primera-lista';
// es equivalente a hacer:
miPrimeraLista.setAttribute('id', 'primera-lista');
```

## Estilos de los nodos

Los estilos están accesibles como el atributo **style**. Cualquier estilo es una propiedad de dicho atributo pero con la sintaxis *camelCase* en vez de *kebab-case*. Por ejemplo, para cambiar el color de fondo (propiedad *background-color*) y ponerle el color rojo al elemento *miPrimeraLista* haremos:

```
miPrimeraLista.style.backgroundColor = 'red';
```

De todas formas, normalmente **NO CAMBIAREMOS ESTILOS** a los elementos, sino que les pondremos o quitaremos clases que harán que se le apliquen o no los estilos definidos para ellas en el CSS.

## Atributos de clase

Ya sabemos que el aspecto de la página debe configurarse en el CSS por lo que no debemos aplicar atributos *style* al HTML. En lugar de ello les ponemos clases a los elementos que harán que se les aplique el estilo definido para dicha clase.

Como es algo muy común en lugar de utilizar las instrucciones de *elemento.setAttribute('className', 'destacado')* o directamente *elemento.className='destacado'* podemos usar la propiedad *classList* que devuelve la colección de todas las clases que tiene el elemento. Por ejemplo, si elemento es `<p class="destacado direccion">...`:

```
let clases=elemento.classList; // clases=['destacado', 'direccion'], OJO es una colección, no un Array
```

Además, dispone de los métodos:

- **.add(clase)**: añade al elemento la clase pasada (si ya la tiene no hace nada). Ej.:

```
elemento.classList.add('primero'); // ahora elemento será <p class="destacado direccion primero">...
```

- **.remove(clase)**: elimina del elemento la clase pasada (si no la tiene no hace nada). Ej.:

```
elemento.classList.remove('direccion'); // ahora elemento será <p class="destacado primero">...
```

- **.toggle(clase)**: añade la clase pasada si no la tiene o la elimina si la tiene ya. Ej.:

```
elemento.classList.toggle('destacado'); // ahora elemento será <p class="primero">...  
elemento.classList.toggle('direccion'); // ahora elemento será <p class="primero direccion">...
```

- **.contains(clase)**: dice si el elemento tiene o no la clase pasada. Ej.:

```
elemento.classList.contains('direccion'); // devuelve true
```

- **.replace(oldClass, newClass):** reemplaza del elemento una clase existente por una nueva.  
Ej.:

```
elemento.classList.replace('primero', 'ultimo'); // ahora elemento será <p class="ultimo
direccion">...
```

Tened en cuenta que NO todos los navegadores soportan classList por lo que si queremos añadir o quitar clases en navegadores que no lo soportan debemos hacerlo con los métodos estándar, por ejemplo, para añadir la clase 'rojo':

```
let clases = elemento.className.split(" ");
if (clases.indexOf('rojo') == -1) {
    elemento.className += ' ' + 'rojo';
}
```

## EVENTOS

### Introducción

Nos permiten detectar acciones que realiza el usuario o cambios que suceden en la página y reaccionar en respuesta a ellas. Existen muchos eventos diferentes (podéis ver la lista en [w3schools](https://www.w3schools.com/js/default_events.asp)) aunque nosotros nos centraremos en los más comunes.

JavaScript nos permite ejecutar código cuando se produce un evento (por ejemplo, el evento "click" del ratón) asociando al mismo una función. Hay varias formas de hacerlo.

### Cómo escuchar un evento

La primera manera "estándar" de asociar código a un evento era añadiendo un atributo con el nombre del evento a escuchar (con 'on' delante) en el elemento HTML. Por ejemplo, para ejecutar código al producirse el evento 'click' sobre un botón se escribía:

```
<input type="button" id="boton1" onclick="alert('Se ha pulsado');" />
```

Una mejora era llamar a una función que contenía el código:

```
<input type="button" id="boton1" onclick="clicked()" />
function clicked() {
  alert('Se ha pulsado');
}
```

Esto “ensuciaba” con código la página HTML por lo que se creó el modelo de registro de eventos tradicional que permitía asociar a un elemento HTML una propiedad con el nombre del evento a escuchar (con ‘on’ delante). En el caso anterior:

```
document.getElementById('boton1').onclick = function () {
  alert('Se ha pulsado');
}
```

NOTA: hay que tener cuidado porque si se ejecuta el código antes de que se haya creado el botón estaremos asociando la función al evento click de un elemento que aún no existe así que no hará nada. Para evitarlo siempre es conveniente poner el código que atiende a los eventos dentro de una función que se ejecute al producirse el evento load de la ventana. Este evento se produce cuando se han cargado todos los elementos HTML de la página y se ha creado el árbol DOM. Lo mismo habría que hacer con cualquier código que modifique el árbol DOM. El código correcto sería:

```
window.onload = function() {
  document.getElementById('boton1').onclick = function() {
    alert('Se ha pulsado');
  }
}
```

La forma recomendada de hacerlo es usando el modelo avanzado de registro de eventos del W3C. Se usa el método `addEventListener` que recibe como primer parámetro el nombre del evento a escuchar (sin ‘on’) y como segundo parámetro la función a ejecutar (OJO, sin paréntesis) cuando se produzca:

```
document.getElementById('boton1').addEventListener('click', pulsado);
```

```
...  
function pulsado() {  
    alert('Se ha pulsado');  
}
```

Habitualmente se usan funciones anónimas ya que no necesitan ser llamadas desde fuera del escuchador:

```
document.getElementById('boton1').addEventListener('click', function() {  
    alert('Se ha pulsado');  
});
```

Si queremos pasarle algún parámetro a la función escuchadora (cosa bastante poco usual) debemos usar funciones anónimas como escuchadores de eventos:

## Event listeners

### Archivo JavaScript

```
window.addEventListener('load', function() {  
    document.getElementById('acepto').addEventListener('click', function() {  
        alert('Se ha aceptado');  
    })  
});
```

### Archivo HTML

```
<button id="acepto">Aceptar</button>
```

Una ventaja de este método es que podemos poner varios escuchadores para el mismo evento y se ejecutarán todos ellos. Para eliminar un escuchador se usa el método `removeEventListener`.

```
document.getElementById('acepto').removeEventListener('click', aceptado);
```

NOTA: no se puede quitar un escuchador si hemos usado una función anónima, para quitarlo debemos usar como escuchador una función con nombre.

## Tipos de eventos

Según qué o dónde se produce un evento, estos se clasifican en:

### Eventos de página

Se producen en el documento HTML, normalmente en el BODY:

- **load**: se produce cuando termina de cargarse la página (cuando ya está construido el árbol DOM). Es útil para hacer acciones que requieran que el DOM esté cargado como modificar la página o poner escuchadores de eventos
- **unload**: al destruirse el documento (ej. cerrar)
- **beforeUnload**: antes de destruirse (podríamos mostrar un mensaje de confirmación)
- **resize**: si cambia el tamaño del documento (porque se redimensiona la ventana)

### Eventos de ratón

Los produce el usuario con el ratón:

- **click / dblclick**: cuando se hace click/doble click sobre un elemento
- **mousedown / mouseup**: al pulsar/soltar cualquier botón del ratón
- **mouseenter / mouseleave**: cuando el puntero del ratón entra/sale del elemento (también podemos usar *mouseover/mouseout*)
- **mousemove**: se produce continuamente mientras el puntero se mueva dentro del elemento

NOTA: si hacemos doble clic sobre un elemento la secuencia de eventos que se produciría es: *mousedown -> mouseup -> click -> mousedown -> mouseup -> click -> dblclick*.

EJERCICIO: Pon un escuchador desde la consola al botón 1 de la [página de ejemplo de DOM](#) para que al hacer clic sobre él se muestre una ventana modal tipo "alert" con el texto que muestre: 'Clic sobre botón 1'. Ponle otro escuchador para que al pasar el ratón sobre él se muestre una ventana modal tipo "alert" que indique: 'Entrando en botón 1'.



## Eventos de teclado

Los produce el usuario al usar el teclado:

- **keydown**: se produce al presionar una tecla y se repite continuamente si la tecla se mantiene pulsada
- **keyup**: cuando se deja de presionar la tecla
- **keypress**: acción de pulsar y soltar (sólo se produce en las teclas alfanuméricas)

NOTA: el orden de secuencia de los eventos cuando se produce un evento *keyUp* es: *keyDown* -> *keyPress* -> *keyUp*

## Eventos de toque

Se producen al usar una pantalla táctil:

- **touchstart**: se produce cuando se detecta un toque en la pantalla táctil.
- **touchend**: cuando se deja de pulsar la pantalla táctil.
- **touchmove**: cuando un dedo es desplazado a través de la pantalla.
- **touchcancel**: cuando se interrumpe un evento táctil.

## Eventos de formulario

Se producen en los formularios:

- **focus / blur**: al obtener/perder el foco el elemento
- **change**: al perder el foco un `<input>` o `<textarea>` si ha cambiado su contenido o al cambiar de valor un `<select>` o un `<checkbox>`
- **input**: al cambiar el valor de un `<input>` o `<textarea>` (se produce cada vez que escribimos una letra en estos elementos)
- **select**: al cambiar el valor de un `<select>` o al seleccionar texto de un `<input>` o `<textarea>`
- **submit / reset**: al enviar/recargar un formulario

## Los objetos this y event

Al producirse un evento se generan automáticamente en su función manejadora 2 objetos:

- **this**: siempre hace referencia al elemento que contiene el código en donde se encuentra la variable this. En el caso de una función escuchadora será el elemento que tiene el escuchador que ha recibido el evento
- **event**: es un objeto y la función escuchadora lo recibe como parámetro. Tiene propiedades y métodos que nos dan información sobre el evento, como:
  - **type**: qué evento se ha producido (click, submit, keyDown, ...)
  - **.target**: el elemento donde se produjo el evento (puede ser this o un descendiente de this, como en el ejemplo siguiente)
  - **.currentTarget**: el elemento que contiene el escuchador del evento lanzado (normalmente el mismo que this). Por ejemplo si tenemos un <p> al que le ponemos un escuchador de 'click' que dentro tiene un elemento \_\_, si hacemos \_\_.click sobre el \_\_ event.target será el \_\_ que es donde hemos hecho click (está dentro de <p>) pero tanto \_\_ como \_\_.event.currentTarget será \_\_<p> (que es quien tiene el escuchador que se está ejecutando).
  - **.relatedTarget**: en un evento 'mouseover' event.target es el elemento donde ha entrado el puntero del ratón y event.relatedTarget el elemento del que ha salido. En un evento 'mouseout' sería al revés.
  - **cancelable**: si el evento puede cancelarse. En caso afirmativo se puede llamar a event.preventDefault() para cancelarlo
  - **.preventDefault()**: si un evento tiene un escuchador asociado se ejecuta el código de dicho escuchador y después el navegador realiza la acción que correspondería por defecto al evento si no tuviera escuchador (por ejemplo un escuchador del evento click sobre un hipervínculo hará que se ejecute su código y después saltará a la página indicada en el href del hipervínculo). Este método cancela la acción por defecto del navegador para el evento. Por ejemplo si el evento era el submit de un formulario éste no se enviará o si era un click sobre un hipervínculo no se irá a la página indicada en él.
  - **.stopPropagation**: un evento se produce sobre un elemento y todos sus padres. Por ejemplo si hacemos click en un <span> que está en un <p> que está en un <div> que está en el BODY el evento se va propagando por todos estos elementos y saltarían los escuchadores asociados a todos ellos (si los hubiera). Si alguno llama a este método el evento no se propagará a los demás elementos padre.
  - dependiendo del tipo de evento tendrá más propiedades:
    - eventos de ratón:
      - **.button**: qué botón del ratón se ha pulsado (0: izq, 1: rueda; 2: dcho).
      - **.screenX / .screenY**: las coordenadas del ratón respecto a la pantalla
      - **.clientX / .clientY**: las coordenadas del ratón respecto a la ventana cuando se produjo el evento
      - **.pageX / .pageY**: las coordenadas del ratón respecto al documento (si se ha hecho un scroll será el clientX/Y más el scroll)
      - **.offsetX / .offsetY**: las coordenadas del ratón respecto al elemento sobre el que se produce el evento
      - **.detail**: si se ha hecho click, doble click o triple click
    - eventos de teclado: son los más incompatibles entre diferentes navegadores. En el teclado hay teclas normales y especiales (Alt, Ctrl, Shift, Enter, Tab, flechas, Supr, ...). En la información del teclado hay que distinguir

entre el código del carácter pulsado (e=101, E=69, €=8364) y el código de la tecla pulsada (para los 3 caracteres es el 69 ya que se pulsa la misma tecla). Las principales propiedades de event son:

- **.key**: devuelve el nombre de la tecla pulsada
- **.which**: devuelve el código de la tecla pulsada
- **.keyCode / .charCode**: código de la tecla pulsada y del carácter pulsado (según navegadores)
- **.shiftKey / .ctrlKey / .altKey / .metaKey**: si está o no pulsada la tecla SHIFT / CTRL / ALT / META. Esta propiedad también la tienen los eventos de ratón

NOTA: a la hora de saber qué tecla ha pulsado el usuario es conveniente tener en cuenta:

- para saber qué carácter se ha pulsado lo mejor usar la propiedad *key* o *charCode* de *keyPress*, pero varía entre navegadores
- para saber la tecla especial pulsada mejor usar el *key* o el *keyCode* de *keyUp*
- captura sólo lo que sea necesario, se producen muchos eventos de teclado
- para obtener el carácter a partir del código: `String.fromCharCode(codigo);`

Lo mejor para familiarizarse con los diferentes eventos es consultar los ejemplos de [w3schools](https://www.w3schools.com/js/default.asp).

EJERCICIO: Implementa desde la consola un escuchador al BODY de la página de ejemplo para que, al mover el ratón en cualquier punto de la ventana del navegador, se muestre en algún sitio (añade un DIV o un P al HTML para colocar este mensaje que te pide el ejercicio) la posición del puntero respecto del navegador y respecto de la página.

EJERCICIO: Desde la consola del navegador, coloca un escuchador al BODY de la página de ejemplo para que al pulsar cualquier tecla nos muestre en un alert el *key* y el *keyCode* de la tecla pulsada. Pruébalo con diferentes teclas

Bindeo del objeto *this*

En ocasiones no queremos que *this* sea el elemento sobre quien se produce el evento, sino que, queremos conservar el valor que tenía antes de entrar a la función escuchadora. Por ejemplo, la función escuchadora es un método de una clase en *this* y tenemos el objeto de la clase sobre el que estamos actuando pero al entrar en la función perdemos esa referencia.

El método `.bind()` nos permite pasarle a una función el valor que queremos darle a la variable `this` dentro de dicha función. Por defecto a una función escuchadora de eventos se le bindea el valor de **`event.currentTarget`**. Si queremos que tenga otro valor se lo indicamos con **`.bind()`**:

```
document.getElementById('acepto').removeEventListener('click', aceptado.bind(variable));
```

En este ejemplo el valor de *this* dentro de la función `aceptado` será `variable`. En el ejemplo que habíamos comentado de un escuchador dentro de una clase, para mantener el valor de `this` y que haga referencia al objeto sobre el que estamos actuando haríamos:

```
document.getElementById('acepto').removeEventListener('click', aceptado.bind(this));
```

por lo que el valor de *this* dentro de la función `aceptado` será el mismo que tenía fuera, es decir, el objeto.

Podemos *bindear*, es decir, pasarle a la función escuchadora más variables declarándolas como parámetros de *bind*. El primer parámetro será el valor de *this* y los demás serán parámetros que recibirá la función antes de recibir el parámetro *event* que será el último. Por ejemplo:

```
document.getElementById('acepto').removeEventListener('click', aceptado.bind(var1, var2, var3));  
...  
function aceptado(param1, param2, event) {  
    // Aquí dentro tendremos los valores  
    // this = var1  
    // param1 = var2  
    // param2 = var3  
    // event es el objeto con la información del evento producido  
}
```

## Propagación de eventos (bubbling)

Normalmente en una página web los elementos HTML se solapan unos con otros, por ejemplo, un `<span>` está en un `<p>` que está en un `<div>` que está en el `<body>`. Si ponemos un escuchador del evento *click* a todos ellos se ejecutarán todos ellos, pero ¿en qué orden?

Pues el W3C estableció un modelo en el que primero se disparan los eventos de fuera hacia dentro (primero el `<body>`) y al llegar al más interno (el `<span>`) se vuelven a disparar de nuevo pero de dentro hacia afuera. La primera fase se conoce como **fase de captura** y la segunda como **fase de burbujeo**. Cuando ponemos un escuchador con `addEventListener` el tercer parámetro indica en qué fase debe dispararse:

- **true**: en fase de captura.
- **false** (valor por defecto): en fase de burbujeo.

Podéis ver un ejemplo en:

Fichero JavaScript:

```
let divClick = function(event) {  
    // eventPhase: 1 -> capture, 2 -> target (objetivo), 3 -> bubble  
    document.getElementById('info').innerHTML+="Has pulsado: " + this.id + ". Fase: " +  
event.eventPhase + ", this: " + this.id + ", event.target: " + event.target.id + ",  
event.currentTarget: " + event.currentTarget.id + "<br>";  
};  
  
let divVerde = document.getElementById("divVerde");  
let divRojo = document.getElementById("divRojo");  
let divAzul = document.getElementById("divAzul");  
divVerde.addEventListener('click', divClick);  
divRojo.addEventListener('click', divClick);  
divAzul.addEventListener('click', divClick);
```

Fichero HTML:

```
<div id="divVerde" style="background-color: green; width: 150px; height: 150px;">  
  <div id="divRojo" style="background-color: red; width: 100px; height: 100px;">  
    <div id="divAzul" style="background-color: blue; width: 50px; height: 50px;"></div>  
  </div>  
</div>  
<p id="info"></p>
```

Sin embargo si al método `.addEventListener` le pasamos un tercer parámetro con el valor `true` el comportamiento será el contrario, lo que se conoce como *captura* y el primer escuchador que se ejecutará es el del `<body>` y el último el del `<span>` (podéis probarlo añadiendo ese parámetro a los escuchadores del ejemplo anterior).

En cualquier momento podemos evitar que se siga propagando el evento ejecutando el método `.stopPropagation()` en el código de cualquiera de los escuchadores.

Podéis ver las distintas fases de un evento en la página [domevents.dev](https://domevents.dev).

## innerHTML y escuchadores de eventos

Si cambiamos la propiedad innerHTML de un elemento del árbol DOM todos sus escuchadores de eventos desaparecen ya que es como si se volviera a crear ese elemento (y los escuchadores deben ponerse después de crearse).

Por ejemplo, tenemos una tabla de datos y queremos que al hacer doble clic en cada fila se muestre su id. La función que añade una nueva fila podría ser:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = `<tr id="${data.id}"><td>${data.dato1}</td><td>${data.dato2}...</td></tr>`;
  miTabla.innerHTML += nuevaFila;
  document.getElementById(data.id).addEventListener('dblclick', event => alert('Id: ' +
    event.target.id));
}
```

Sin embargo, esto sólo funcionaría para la última fila añadida ya que la línea `miTabla.innerHTML += nuevaFila` equivale a `miTabla.innerHTML = miTabla.innerHTML + nuevaFila`. Por tanto, estamos asignando a *miTabla* un código HTML que ya no contiene escuchadores, excepto el de *nuevaFila* que lo ponemos después de hacer la asignación.

La forma correcta de hacerlo sería:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = document.createElement('tr');
  nuevaFila.id = data.id;
  nuevaFila.innerHTML = `<td>${data.dato1}</td><td>${data.dato2}...</td>`;
  nuevaFila.addEventListener('dblclick', event => alert('Id: ' + event.target.id));
  miTabla.appendChild(nuevaFila);
}
```

De esta forma además mejoramos el rendimiento ya que el navegador sólo tiene que renderizar el nodo correspondiente a la *nuevaFila*. Si lo hacemos como estaba al principio se deben volver a crear y a renderizar todas las filas de la tabla (todo lo que hay dentro de *miTabla*).

## Eventos personalizados

También podemos mediante código lanzar manualmente cualquier evento sobre un elemento con el método `dispatchEvent()` e incluso crear eventos personalizados, por ejemplo:

```
const event = new Event('build');

// Listen for the event.
elem.addEventListener('build', (e) => { /* ... */ }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

Incluso podemos añadir datos al objeto *event* si creamos el evento con `new CustomEvent()`. Podéis obtener más información en la [página de MDN](#).

## WEB API

Acronimo ingles que hace referencia a Application Program Interface, o, Interfaz de Aplicaciones del Programa. No es más que una interfaz, o herramienta, que hace de intermediario entre un sistema y otro.

Que dos sistemas intervienen en este caso y uno este Web API, por un lado, el motor de interpretación de código del navegador y por el otro lado el propio lenguaje JavaScript.

Para entenderlo podemos pensar que un API, en este caso, Web API, es como un traductor en una conferencia internacional. Es decir, JavaScript se intenta comunicar con el motor del navegador, pero el motor del navegador no lo entiende por qué no sabe que quiere decir cuando dice JavaScript que quiere recargar una web.

Entonces, Web API interviene para poder mediante su propia mediación recibir la orden desde JavaScript y decir al motor del navegador que es lo que quiere conseguir.

Para ello, Web API tiene un montón de Interfaces que ofrecer a JavaScript para que se comunique fácil y eficientemente con el navegador que toque, pero dada la enorme cantidad y la dificultad, por la falta de tiempo, nos centramos en 4 elementos de la API de todas las que nos ofrece Web API:

- **Window:** representa una ventana del navegador que contiene un DOM.
- **Document:** Representa el propio DOM que se encuentra en una ventana del navegador.
- **Event:** representa los eventos que ocurren en ese DOM.
- **Element:** representa un nodo (elemento) del DOM.

Todas las APIs se pueden consultar en la dirección web de la WEB API:

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

Por ejemplo, si queremos ver distintas maneras de usar la WEB API de Window podemos probar los siguientes comandos por consola:

Instrucción	Resultado
<b>API window</b>	
<code>window.console.log("Hola");</code>	Muestra en la consola el mensaje "hola"
<code>window.alert("Hola...");</code>	Muestra una ventana modal con el mensaje "hola"
<code>window.print();</code>	Muestra el menu "Imprimir" del navegador.
<code>window.scrollTo(0, 100);</code>	Baja 100 pixeles verticalmente.
<b>API document</b>	
<code>document.querySelector(".o3j99.n1xJcf");</code>	Selecciona un objeto del DOM
<code>document.querySelector(".o3j99.n1xJcf").remove();</code>	Borra un objeto del DOM
<code>document.querySelectorAll("input");</code>	Selecciona todos los elementos del DOM de tipo input.
<code>document.getSelection();</code>	Selecciona el elemento del DOM que el usuario seleccione.
<code>document.location</code>	Propiedad que nos devuelve un objeto con muchos datos que nos muestran, entre otros, los datos del host, protocolos, href, puerto, origin, servidor, hash, etc...
<code>document.title</code>	Propiedad que nos devuelve el valor de la etiqueta title que se define dentro de la etiqueta Head.
<code>document.cookie</code>	Propiedad que devuelve las cookies del documento.
<b>RECORDATORIO: UN ID se selecciona con el símbolo hashtag.</b>	<b>#</b>
<b>RECORDATORIO: UNA CLASE se selecciona con el punto.</b>	<b>.</b>
<code>document.querySelector("#SivCob");</code>	Nos devuelve un objeto que es un elemento del DOM.
<code>document.querySelector("#SivCob").innerHTML;</code>	Nos devuelve el código HTML incluido dentro de un elemento del DOM.
<code>document.querySelector("#SivCob").innerText;</code>	Nos devuelve el texto incluido dentro de un elemento del DOM.



## API LocalStorage

Antes de HTML5 la única manera que tenían los programadores de guardar algo en el navegador del cliente (como sus preferencias, su idioma predeterminado para nuestra web, etc) era utilizando cookies. Las cookies tienen muchas limitaciones y es engorroso trabajar con ellas.

HTML5 incorpora la API de Storage para subsanar esto. Además, existen otros métodos de almacenamiento en el cliente más avanzados como *IndexedDB* (es un estándar del W3C pero aún con poco soporte entre los navegadores).

El funcionamiento de la API Storage es muy sencillo: dentro del objeto window tendremos los objetos *localStorage* y *sessionStorage* donde podremos almacenar información en el espacio de almacenamiento local (5 o 10 MB por sitio web según el navegador, que es mucho más de lo que teníamos con las cookies). La principal diferencia entre ellos es que la información almacenada en *localStorage* nunca expira, permanece allí hasta que la borremos (nosotros o el usuario) mientras que la almacenada en *sessionStorage* se elimina automáticamente al cerrar la sesión el usuario.

Sólo los navegadores muy antiguos (Internet Explorer 7 y anteriores) no soportan esta característica. Puedo saber si el navegador soporta esta API simplemente mirando su `typeof`:

```
if (typeof(Storage) === 'undefined') // NO está soportado
```

Tanto *localStorage* como *sessionStorage* son como un objeto global al que tengo acceso desde el código. Lo que puedo hacer con ellos es:

- Guardar un dato: ***localStorage.setItem('dato', 'valor')*** o también ***localStorage.dato = 'valor'***
- Recuperar un dato: ***let miDato=localStorage.getItem('dato')*** o también ***let miDato = localStorage.dato***
- Borrar datos: ***localStorage.removeItem('dato')*** para borrar 'dato'. Si quiero borrar TODO lo que tengo ***localStorage.clear()***
- Saber cuántos datos tenemos: ***localStorage.length***

Sólo podemos guardar objetos primitivos (cadenas, números, ...) por lo que si queremos guardar un objeto o un array hay que convertirlo a una cadena JSON con ***localStorage.setItem('dato', JSON.stringify('objeto'))***. Para recuperar el objeto haremos ***let miObjeto = JSON.parse(localStorage.getItem('dato'))***.

Cada vez que cambia la información que tenemos en nuestro *localStorage* se produce un evento *storage*. Si, por ejemplo, queremos que una ventana actualice su información si otra cambia algún dato del *storage* haremos:

```
window.addEventListener("storage", actualizaDatos);
```

y la función 'actualizaDatos' podrá leer de nuevo lo que hay y actuar en consecuencia.

EJERCICIO: comprueba qué tienes almacenado en el `localStorage` y el `sessionStorage` de tu navegador. Guarda y recupera algunas variables. Luego cierra el navegador y vuelve a abrir la página. ¿Están las variables guardadas en `localStorage`? ¿Y las de `sessionStorage`?

Puedes ver un ejemplo [en este vídeo](#) de cómo almacenar en el *Storage* datos del usuario.

A tener en cuenta

*localStorage*, *sessionStorage* y cookies almacenan información en un navegador específico del cliente, y por tanto:

- No podemos asegurar que permanece ahí
- Puede ser borrada/manipulada
- Puede ser leída, por lo que NO es adecuada para almacenar información sensible pero sí para preferencias del usuario, marcadores de juegos, etc

Podríamos usar *localStorage* para almacenar localmente los datos con los que trabaja una aplicación web. Así conseguiríamos minimizar los accesos al servidor y que la velocidad de la aplicación sea mucho mayor al trabajar con datos locales. Pero periódicamente debemos sincronizar la información con el servidor.

Storage vs cookies

Ventajas de *localStorage*:

- 5 o 10 MB de almacenamiento frente a 4 KB de las cookies
- Todas las cookies del dominio se envían al servidor con cada petición al mismo lo que aumenta el tráfico innecesariamente

Ventajas de las *cookies*:

- Soportadas por navegadores muy antiguos

- Las cookies ofrecen algo de protección frente a XSS (Cross-Site Scripting) y Script injection

## Cookies

Son pequeños ficheros de texto y tienen las siguientes limitaciones:

- Máximo 300 cookies, si hay más se borran las antiguas
- Máximo 4 KB por cookie, si nos pasamos se truncará
- Máximo 20 cookies por dominio

Cada cookie almacena los siguientes datos:

- Nombre de la cookie (obligatorio)
- Valor de la misma.
- *expires*: timestamp en que se borrará (si no pone nada se borra al salir del dominio)
- *max-age*: en lugar de expires podemos indicar aquí los segundos que durará la cookie antes de expirar
- *path*: ruta desde dónde es accesible (/: todo el dominio, /xxx: esa carpeta y subcarpetas). Si no se pone nada sólo lo será desde la carpeta actual
- *domain*: dominio desde el que es accesible. Si no ponemos nada lo será desde este dominio y sus subdominios
- *secure*: si aparece indica que sólo se enviará esta cookie con https

Un ejemplo de cookie sería:

```
username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC;
```

Se puede acceder a las cookies desde `document.cookie` que es una cadena con las cookies de nuestras páginas. Para trabajar con ellas conviene que creamos funciones para guardar, leer o borrar cookies, por ejemplo:

- Crear una nueva cookie:

```
function setCookie(cname, cvalue, cexpires, cpath, cdomain, csecure) {  
    document.cookie = cname + '=' + cvalue +  
        (cexpires?';expires='+cexpires.toUTCString():'') +  
        (cpath?';path='+cpath:'') +  
        (cdomain?';domain='+cdomain:'') +  
        (csecure?';secure':'')  
}
```

- Leer una cookie:

```
function getCookie(cname) {
  if(document.cookie.length > 0){
    start = document.cookie.indexOf(cname + '=')
    if (start != -1) { // Existe la cookie, busquemos dónde acaba su valor
      //El inicio de la cookie, el nombre de la cookie mas les simbolo '='
      start = start + nombre.length + 1
      //Buscamos el final de la cookie (es el simbolo ';')
      end = document.cookie.indexOf(';', start + cname.length + 1)
      if (end === -1) { // si no encuentra el ; es que es la última cookie
        end = document.cookie.length;
      }
      return document.cookie.substring(start + cname.length + 1, end)
    }
  }
  return '' // Si estamos aquí es que no hemos encontrado la cookie
}
```

- Borrar una cookie:

```
function delCookie(cname) {
  return document.cookie = cname + ';;expires=Thu, 01 Jan 1970 00:00:01 GMT;';
}
```

Podéis ver en [este vídeo](#) un ejemplo de cómo trabajar con cookies, aunque como ya hemos dicho lo recomendable es trabajar con *Storage*.

## REACT

