

## DWEC - Javascript Web Cliente.

# JavaScript - Anexo - Uso de **this** en contexto

Una método de un objeto es una función, en el ejemplo la función **getInfo()**:

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
};

alumno.getInfo = function() {
    return `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`
}

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

También se puede incluir la declaración del método en la declaración del objeto:

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: function(){
        return `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`;
    }
};

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

OJO: deberíamos poder ponerlo con sintaxis *arrow function*, pero no funciona.

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: ()=> `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`
};

console.log(alumno.getInfo()); //El alumno undefined undefined tiene undefined años
```

Si, es por algo que se llama el **alcance léxico**. Es por el **this**: con la función normal, **this** se refiere al objeto sobre el que se está invocando el método (hasta ahí bien), pero con la **función flecha** **this** se refiere a otra cosa.

Con la función flecha this se comporta de forma diferente manteniendo el valor que tenía this fuera de la función flecha.

Sí, this en JavaScript estuvo super mal diseñado y depende de la forma en que se invoque la función.

Para hacer ese método declara la función como una función normal.

En general, this se refiere al objeto actual.

Pero si no hay un objeto "actual", this depende del ambiente: en el navegador se refiere al objeto global window, en Node.js es undefined.

Así en este caso (navegador) podríamos poner window.innerHeight o this.innerHeight indistintamente. En ninguno de los dos casos conseguiremos el objetivo. Veamos:

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: ()=> `El alumno ${window.innerHeight} ${this.innerHeight} tiene
${this.edad} años`
};

console.log(alumno.getInfo()); //El alumno 758 758 tiene undefined años
```

En una función normal el valor de this puede cambiar dependiendo de cómo se llame la función. Supongamos que tenemos el siguiente método hello que utiliza this dentro de un objeto:

```
const person = {
    name: "Pedro",
    hello: function hello() {
        console.log(this.name)
    }
}
```

Si invocamos el método hello sobre person el resultado es el esperado:

```
persona.hello() // "Pedro"
```

Pero uno puede cambiar el this en hello de varias formas al invocarlo, una de ellas es con el método call que recibe como primer argumento lo que uno quiere que sea this:

```
persona.hello.call({ name: "Maria" }) // "Maria"
```

En este ejemplo estoy cambiando el this para que sea un objeto que tiene una llave name con valor "María".

Con las funciones flecha no hay forma de cambiar el this, que siempre se refiere al this que existía cuando se evalúa la función. Esto es más predecible, pero tiene la desventaja que no se puede utilizar en algunos casos, como en los métodos de objetos o funciones constructoras.

**Nota:** Al valor del this también se le conoce como el contexto.

## DWEC – Javascript Web Cliente.

# JavaScript 00 – Lenguaje JavaScript

JavaScript 00 – Lenguaje JavaScript .....	1
Introducción .....	1
Usaremos JavaScript para: .....	2
Un poco de historia .....	2
Soporte en los navegadores .....	2
Herramientas .....	3
La consola del navegador .....	3
Editores .....	3
Editores on-line .....	3
npm .....	4
Git .....	4
GitHub .....	4
Incluir JavaScript en una página web .....	4
Mostrar información .....	6

## Introducción

Para el desarrollo de las páginas web lo principal es un fichero HTML que contiene la información a mostrar en el navegador. Posteriormente surgió la posibilidad de “decorar” esa información para mejorar su apariencia, lo que dio lugar al CSS. Y también se pensó en dar dinamismo a las páginas, y apareció el lenguaje JavaScript. Que también lo escribiremos como Javascript o JS.

En un primer momento las 3 funcionalidades estaban mezcladas en el fichero HTML, pero eso complicaba bastante el poder leer esa página a la hora de mantenerla por lo que se pensó en separar los 3 elementos básicos:

**HTML:** se encarga de estructurar la página y proporciona su información, pero es una información estática

**CSS:** es lo que da forma a dicha información, permite mejorar su apariencia, permite que se adapte a distintos dispositivos, ...

**JavaScript:** es el que da vida a un sitio web y le permite reaccionar a las acciones del usuario

Por tanto, nuestras aplicaciones tendrán estos 3 elementos y lo recomendable es que estén separados en distintos ficheros:

El **HTML** lo tendremos habitualmente en un **fichero index.html**, normalmente en una **carpeta llamada public, html** (o similar).

El **CSS** lo tendremos en **uno o más ficheros con extensión .css** dentro de una **carpeta llamada styles, estilos, css** (o similar).

El **JS** estará en **ficheros con extensión .js** en un **directorio llamado js, scripts, funciones** (o similar).

Nota: Existen variedades y complementos del lenguaje JavaScript, como son TypeScript, CoffeScript, Vue, Angular, etc.

**Vanilla JavaScript** es como se conoce al lenguaje JavaScript cuando se utiliza sin ninguna librería o *framework*. La traducción más castellana sería “JavaScript a pelo”.

Las características principales de Javascript son:

- Es un lenguaje interpretado, no compilado.
- Se ejecuta en el lado cliente (en un navegador web), aunque hay implementaciones como NodeJS para el lado servidor.
- Es un lenguaje orientado a objetos (podemos crear e instanciar objetos y usar objetos predefinidos del lenguaje) pero basado en prototipos (por debajo, un objeto es un prototipo y nosotros podemos crear objetos sin instanciarlos, haciendo copias del prototipo).
- Se trata de un lenguaje débilmente tipado, con tipificación dinámica (no se indica el tipo de datos de una variable al declararla e incluso puede cambiarse).

Usaremos JavaScript para:

- Cambiar el contenido de la página
- Cambiar los atributos de un elemento
- Cambiar la apariencia de algo
- Validar datos de formularios
- ...

Sin embargo, por razones de seguridad, JavaScript no nos permite hacer cosas como:

- Acceder al sistema de ficheros del cliente
- Capturar datos de un servidor (puede pedirlo y el servidor se los servirá, o no)
- Modificar las preferencias del navegador
- Enviar e-mails de forma invisible o crear ventanas sin que el usuario lo vea
- ...

## Un poco de historia

JavaScript es una implementación del lenguaje **ECMAScript** (el estándar que define sus características).

El lenguaje surgió en 1997 y todos los navegadores a partir de 2012 soportan al menos la versión **ES5.1** completamente.

En 2015 se lanzó la 6<sup>a</sup> versión, inicialmente llamada **ES6** y posteriormente renombrada como **ES2015**, que introdujo importantes mejoras en el lenguaje y que es la versión que usaremos nosotros.

Desde entonces van saliendo nuevas versiones cada año que introducen cambios pequeños. La última es la versión 13, llamada ES2022, que ha sido aprobada en verano de 2022.

Las principales mejoras que introdujo ES2015 son: clases de objetos, let, for..of, Map, Set, Arrow functions, Promesas, spread, destructuring, ...

## Soporte en los navegadores

Los navegadores no se adaptan inmediatamente a las nuevas versiones de JavaScript, por lo que puede ser un problema usar una versión muy moderna e JS ya que puede haber partes de los programas que no funcionen en los navegadores de muchos usuarios.

En la página de [Kangax](#) podemos ver la compatibilidad de los diferentes navegadores con las distintas versiones de JavaScript.

También podemos usar [CanIUse](#) para buscar la compatibilidad de un elemento concreto de JavaScript, así como de HTML5 o CSS3.

Si queremos asegurar la máxima compatibilidad debemos usar la versión ES5 (pero nos perdemos muchas mejoras del lenguaje).

Lo mejor sería usar la ES6 (o posterior) y después *transpilar* nuestro código a la versión ES5. De esto se ocupan los *transpiladores* ([Babel](#) es el más conocido), por lo que no suponen un esfuerzo extra para el programador.

Si queremos asegurar la máxima compatibilidad debemos usar la versión ES5 (pero nos perdemos muchas mejoras del lenguaje) o mejor, usar la ES6 (o posterior) y después transpilar nuestro código a la versión ES5. De esto se ocupan los transpiladores (Babel es el más conocido) por lo que no suponen un esfuerzo extra para el programador.

## Herramientas

### La consola del navegador

Es la herramienta que más nos va a ayudar a la hora de depurar nuestro código. Abrimos las herramientas para el desarrollador (en Chrome y Firefox pulsando la tecla F12) y vamos a la pestaña Consola:

Allí vemos mensajes del navegador como errores y advertencias que genera el código y, todos los mensajes que pongamos en el código para ayudarnos a depurarlo (usando los comandos `console.log` y `console.error`).

Además, en ella podemos escribir instrucciones JavaScript que se ejecutarán mostrando su resultado. También la usaremos para mostrar el valor de nuestras variables y para probar código que, una vez que funcione correctamente, lo copiaremos a nuestro programa.

Siempre depuraremos los programas desde la consola (pondremos puntos de interrupción, veremos el valor de las variables, ...).

### Editores

Podemos usar el que más nos guste, desde editores tan simples como NotePad++ hasta complejos IDEs. La mayoría soportan las últimas versiones de la sintaxis de JavaScript (Netbeans, Eclipse, Visual Studio, Sublime Text, Atom, Kate, Notepad++, ...).

Por el momento utilizaremos el editor [Visual Studio Code](#) por su sencillez y por los plugins que incorpora para hacer más cómodo el trabajo del desarrollador. En *Visual Studio Code* instalaremos algunos plugins como:

**SonarLint:** es más que un *linter* y nos informa de todo tipo de errores, pero también del código que no cumple las recomendaciones (incluye gran número de reglas). Marca el código mientras lo escribimos y además podemos ver todas las advertencias en el panel de Problemas (Ctrl+Shift+M)

**Otros plugins:** según el documento a nuestra disposición “[Instalación de Extensiones de Visual Studio Code](#)”

### Editores on-line

Son muy útiles porque permiten ver el código y el resultado a la vez. Normalmente tienen varias pestañas o secciones de la página donde poner el código HTML, CSS y Javascript y ver su resultado.

Algunos de los más conocidos son [Codesandbox](#), [Fiddle](#), [Plunker](#), [CodePen](#), ...aunque hay muchos más.

Como ejemplo: <https://jsfiddle.net/tqobL253/>

Que se puede ver cómo queda incrustado en una página web si incrustamos el siguiente código:

```
<script async src="//jsfiddle.net/tqobL253/embed/js,html,css,result/dark/"></script>
```

## npm

**npm** es el gestor de paquetes del framework JavaScript **Node.js** y suele utilizarse en programación *frontend* como gestor de dependencias de la aplicación.

Esto significa que será la herramienta que se encargará de descargar y poner a disposición de nuestra aplicación todas las librerías JavaScript que vayamos a utilizar.

Para instalar *npm* tenemos que instalar *NodeJS*.

## Git

Usaremos repositorios git realizar el control de versiones de nuestras aplicaciones.

## GitHub

Usaremos una cuenta en GitHub.com, generalmente asociada a otra cuenta de Gmail.com, para llevar el control de las versiones y tener copia de nuestro código en la nube.

## Incluir JavaScript en una página web

El código JavaScript va entre etiquetas `<script>`. Puede ponerse en el `<head>` o en el `<body>`. Funciona como cualquier otra etiqueta.

El navegador la interpreta cuando llega a ella. Va leyendo la etiqueta y ejecutando el fichero línea a línea.

### Opción 1: en HEAD

```
<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <script>
    function saludar() {
      alert("Hola majos/as");
    }
    saludar();
  </script>
</head>
<body>
  <h1>Bienvenidos a DWEC</h1>
</body>
</html>
```

### Opción 2: en BODY (delante de otros elementos):

```
<!DOCTYPE html>
<head>
```

```

<meta charset="UTF-8">
</head>
<body>
  <script>
    function saludar() {
      alert("Hola majos/as");
    }
    saludar();
  </script>
  <h1>Bienvenidos a DWEC</h1>
</body>
</html>

```

### Opción 3: en fichero externo de extensión js (en HEAD)

```

<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <script src="js/funciones.js"></script>
</head>
<body>
  <h1>Bienvenidos a DWEC</h1>

  <h1>hasta otra</h1>
</body>
</html>

```

### Opción 4 (Recomendada): en fichero externo de extensión js (en BODY)

```

<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
  <h1>Bienvenidos a DWEC</h1>

  <h1>hasta otra</h1>
  <script src="js/funciones.js"></script>
</body>
</html>

```

Lo mejor, en cuanto a rendimiento, es ponerla al final del `<body>` para que no se detenga el renderizado de la página mientras se descarga y se ejecuta el código.

También podemos ponerlo en el `<head>` pero usando los atributos `async` y/o `defer` (en Internet encontraréis mucha información sobre esta cuestión, por ejemplo [aquí](#)).

Es posible poner el código directamente entre la etiqueta `<script>` y su etiqueta de finalización, pero lo correcto es que esté en un fichero externo (con extensión `.js`) que cargamos mediante el atributo `src` de la etiqueta. Así conseguimos que la página HTML cargue más rápido que si lo ponemos al final del BODY o usamos `async`.

Además, es preferible no mezclar HTML y JS en el mismo fichero, lo que mejora la legibilidad del código y facilita su mantenimiento.

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

## Mostrar información

JavaScript permite mostrar al usuario ventanas modales para pedirle o mostrarle información. Las funciones que lo hacen son:

- `window.alert(mensaje)`: Muestra en una ventana modal *mensaje* con un botón de *Aceptar* para cerrar la ventana.
- `window.confirm(mensaje)`: Muestra en una ventana modal *mensaje* con botones de *Aceptar* y *Cancelar*. La función devuelve **true** o **false** en función del botón pulsado por el usuario.
- `window.prompt(mensaje [, valor predeterminado])`: Muestra en una ventana modal *mensaje* y debajo tiene un campo donde el usuario puede escribir, junto con botones de *Aceptar* y *Cancelar*. La función devuelve el valor introducido por el usuario como texto (es decir que si introduce 54 lo que se obtiene es "54") o **false** si el usuario pulsa *Cancelar*.

También se pueden escribir las funciones sin *window*. (es decir `alert('Hola')` en vez de `window.alert('Hola')`) ya que en JavaScript todos los métodos y propiedades de los que no se indica de qué objeto son se ejecutan en el objeto *window*.

Si queremos mostrar una información para depurar nuestro código no utilizaremos `alert(mensaje)` si no `console.log(mensaje)` o `console.error(mensaje)`. Estas funciones muestran la información en la consola del navegador. La diferencia es que `console.error` la muestra como si fuera un error de JavaScript.

Por último, indicar que podremos usar el método `write` del objeto `document` para enviar información (en lenguaje HTML) al documento HTML donde ponemos este código. `Document.window(código_html)`. Ejemplos: `document.write("Hola Daw2")` o `document.write('<p>Hola Daw2</p>')`



## DWEC - Javascript Web Cliente.

# JavaScript 01 A – Sintaxis (I)

JavaScript 01 – Sintaxis .....	1
Variables .....	1
Use Strict .....	2
Otro ejemplo de ámbito de variables:.....	3
Variables locales y variables globales.....	4
Variables constantes (const) .....	5
Funciones.....	5
Parámetros.....	5
Funciones anónimas.....	7
Arrow functions (funciones flecha) .....	7
Estructuras y bucles.....	8
Estructura condicional: if .....	8
Estructura condicional: switch .....	8
Bucle while .....	9

## Variables

Javascript es un lenguaje débilmente tipado. Esto significa que no se indica de qué tipo es una variable al declararla, incluso puede cambiar su tipo a lo largo de la ejecución del programa. Ejemplo:

```
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**

Ni siquiera estamos obligados a declarar una variable antes de usarla, aunque es recomendable para evitar errores que nos costará depurar.

Las variables se declaran con **let** (lo recomendado desde ES2015), aunque también pueden declararse con **var**.

La diferencia es que con **let** la variable sólo existe en el bloque en que se declara.

Mientras que con **var** el **ámbito (scope)** de la variable es global, se extiende. Es decir, existe en toda la función o ámbito en el que se declara:

```

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

```

Cualquier variable que no se declara dentro de una función (o si se usa sin declarar) es **global**. Debemos siempre intentar NO usar variables globales.

Además, como podemos comprobar con el siguiente ejemplo, las variables declaradas con var pueden duplicarse sin que se produzca error, y que a la larga puede acarrear muchos problemas.

```

// CREACIÓN DE VARIABLES CON LET

/* let declara una variable limitando su ámbito (scope) al bloque,
   declaración o expresión donde se está usando. */

// SINTAXIS: let nombreVariable [= valor];

var persona = "Santi";
var persona = "Profesor";
console.log (persona);

let persona2 = "Ana";
//let persona2 = "Jefa de Estudios"; //Esta instrucción devuelve un error
console.log (persona2);

```

Se recomienda que los nombres de las variables sigan la sintaxis *camelCase* (ej.: *miPrimeraVariable*).

Desde ES2015 también podemos declarar constantes con **const**. Se les debe dar un valor al declararlas y si intentamos modificarlo posteriormente se produce un error.

NOTA: en la página de [Babel](#) podemos teclear código en ES2015 y ver cómo quedaría una vez **transpilado** a ES5.

Cuando veamos “next generation JavaScript” se refiere a la próxima versión de Javascript pendiente de ser aceptada

## Use Strict

Para evitar problemas futuros, y que no se produzca algún error si no declaramos una variable, incluiremos al principio de nuestro código la instrucción “**use strict**”, que nos obliga a declarar las variables.

```
'use strict';
```

Veamos un ejemplo:

```

// USE STRICT O MODO ESTRICTO

/* "use strict" es una línea que indica que el código debe ser usado "en modo estricto",
   es decir, no se pueden utilizar variables no declaradas.
   Fuera de una función tiene ámbito global; dentro de ella, local (el de la función).
*/

```

```
// SINTAXIS: "use strict";

// "use strict"; // si aplicamos este código da error al no declarar persona2

persona2 = "Santi";
let nacimiento;

function informar (){
    "use strict";
    let persona = "Santi";
    nacimiento = "1915";
    console.log(persona + " nacio en "+nacimiento);
}

informar();
```

Otro ejemplo de ámbito de variables:

**Caso 1º:** Usamos dos variables (a) definidas con let, cada una queda circunscrita al ámbito donde ha sido creada.

#### / ÁMBITO DE VARIABLES

```
/* El ámbito de una variable (scope) es la zona del programa en la que se define.
Javascript define dos ámbitos para variables: local y global.
Mediante var podemos definir como ámbito local el ámbito de una función.
Con let, por el contrario, podemos diferenciar también el ámbito de bloque. */
```

```
function verAmbito(){
    "use strict";
    let a = "Ámbito de función";
    if (true){
        let a = "Ámbito de bloque";
        console.log ("El ámbito de bloque a es: "+a);
    }
    console.log ("El ámbito de función a es: "+a);
}
verAmbito();
```

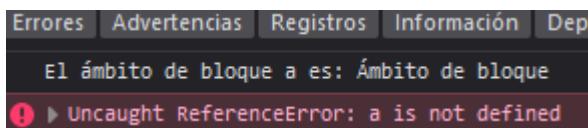
Errores | Advertencias | Registros | Información | Depura  
 El ámbito de bloque a es: Ámbito de bloque  
 El ámbito de función a es: Ámbito de función  
 »

Si en el caso 1º cambiamos el **let** (dentro del if) por **var**, da error de variable ya declarada, puesto que ahora var extiende su scope a toda la función.

Filtrar  
 ✘ Uncaught SyntaxError: Identifier 'a' has already been declared

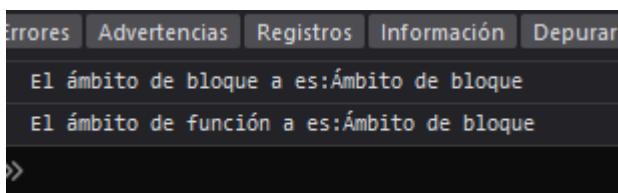
**Caso 2º:** Usamos una variable (a) definida con **let** dentro de un **if**, su ámbito queda reducido al bloque if, por tanto, fuera del bloque daría error.

```
function verAmbito(){
  "use strict";
  if (true){
    let a = "Ámbito de bloque";
    console.log ("El ámbito de bloque a es: "+a);
  }
  console.log ("El ámbito de función a es: "+a); // produce error
}
verAmbito();
```



**Caso 3º:** Usamos una variable (a) definida con **var** dentro de un **if**, su ámbito se extiende a toda la función.

```
function verAmbito(){
  "use strict";
  //let a = "Ámbito de función"; // SIMILAR AL CASO: var a = "Ámbito de función";
  if (true){
    var a = "Ámbito de bloque";
    console.log ("El ámbito de bloque a es:"+a);
  }
  console.log ("El ámbito de función a es:"+a);
}
verAmbito();
// console.log ("El ámbito de función a es: "+a); // ERROR, pues no llega el ámbito
```



Nota: si en el caso 3º cambiamos var por let (estamos igual que el caso)

**Conclusiones:** Se pueden trabajar con **var** y **let** a la vez, pero sería peligroso. En la medida de lo posible trabajaremos con **let**, e incluso con **const** cuando sepamos que esa variable nunca debe cambiar. Además, debemos pensar en que al usar **const** y **let** (su ámbito queda reducido a nuestros bloques) nunca interferirán con otras funciones o código de otro programador en nuestra aplicación.

A modo de repaso recordemos que en este lenguaje de programación no es obligado declarar las variables.

En el caso de ECMAScript 5 usamos la palabra reservada **var**.

En el caso de ECMAScript 6+ además de **var** también podemos usar **let** y **const**.

## Variables locales y variables globales

Una variable global se puede usar en cualquier lugar del script.

Una variable local sólo tiene validez dentro del ámbito de una función.

Las normas que aplican para definir el tipo de variable que estamos usando se resumen con 3 normas sencillas:

- Cualquier variable declarada o no declarada en la raíz de un script es siempre de ámbito global.
- Cualquier variable declarada dentro de una función es de ámbito local.
- Una variable NO declarada dentro de una función adquiere ámbito global.

En caso de duda, que es cuando la variable no ha sido declarada en un ámbito determinado, la variable adquiere comportamiento global.

### Casos de uso de las propiedades del ámbito de las variables

En javascript se pueden dar algunos casos curiosos:

- Dentro de una función podemos sobrescribir el valor de una variable global. Esto puede suceder por ejemplo por error si nos hemos olvidado de declarar una variable... y cargarnos un dato externo a la función sin darnos cuenta.
- Cuando declaramos una variable local con el mismo nombre que una variable global, temporalmente la variable local tiene prioridad sobre la global. Dentro de la función usaremos la variable local. Deberemos tratarlas como si temporalmente la variable global hubiera dejado de existir para nosotros.
- Hay que recordar que una variable declarada dentro de los paréntesis de declaración de la función se considera declarada de tipo local y que por cuestiones de sintaxis nunca va acompañada de la palabra reservada let.

### Variables constantes (const)

Las variables constantes en Javascript (**const**) tienen ámbito de bloque al igual que las variables definidas utilizando **let**. Es importante tener en cuenta que el valor de una constante no puede variar (reasignarse), por tanto, se asignan en el momento en que se declaran. Para diferenciarlo de las variables conviene utilizar **TODOMAYÚSCULAS**.

## Funciones

Se declaran con la palabra reservada **function** y se les pasan los parámetros entre paréntesis. La función puede devolver un valor usando **return** (si no tiene *return* es como si devolviera *undefined*).

Puede usarse una función antes de haberla declarado por el comportamiento de Javascript llamado *hoisting*: el navegador primero carga todas las funciones y mueve las declaraciones de las variables al principio y luego ejecuta el código.

**EJERCICIO mensaje:** Realiza una función que te pida que escribas algo y muestre un alert diciendo 'Has escrito...' y el valor introducido. Pruébala en la consola (pegas allí la función y luego la llamas desde la consola)

### Parámetros

Si se llama a una función con menos parámetros de los declarados el valor de los parámetros no pasados 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;
}

let resultado = potencia(4); // devolverá 1 ya que no se ejecuta el for
console.log(resultado);
//console.log(potencia(4)); // devolverá 1 ya que no se ejecuta el for

```

Podemos dar un **valor por defecto** a los parámetros por si no los pasan asignándoles el valor al definirlos:

```

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)

```

NOTA: En ES5 para dar un valor por defecto a una variable se hacía:

```

function potencia(base, exponente) {
    exponente = exponente || 2; // si exponente vale undefined se la asigna el valor 2
...

```

También es posible acceder a los parámetros desde el array **arguments[]** si no sabemos cuántos parámetros recibiremos:

```

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

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

```

En Javascript las funciones son un tipo de datos más, por lo que podemos hacer cosas como pasárlas como argumento o asignárlas a una variable:

```

const cuadrado=function(value){
    return value * value;
}
function aplicarFuncion(dato, funcion_a_aplicar){

```

```

        return funcion_a_aplicar(dato);
    }
aplicarFuncion(3, cuadrado); // devolverá 9 (3^2)

```

A este tipo de funciones se llama *funciones de primera clase* y son típicas de lenguajes funcionales.

## Funciones anónimas

Como acabamos de ver podemos definir una función sin darle un nombre. Dicha función puede asignarse a una variable, autoejecutarse o asignarsrse a un manejador de eventos. Ejemplo:

```

let holaMundo = function() {
    alert('Hola mundo!');
}

holaMundo();           // se ejecuta la función

```

Como vemos, asignamos una función a una variable de forma que podamos “ejecutar” dicha variable.

## Arrow functions (funciones flecha)

ES2015 permite declarar una función anónima de forma más corta. Ejemplo sin *arrow function*:

```

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

```

Al escribirla con la sintaxis de una *arrow function* lo que hacemos es:

- Eliminamos la palabra *function*
- Si sólo tiene 1 parámetro podemos eliminar los paréntesis de los parámetros
- Ponemos el símbolo =>
- Si la función sólo tiene 1 línea podemos eliminarnr las {} y la palabra *return*

El ejemplo con *arrow function*:

```

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

```

Otro ejemplo, sin *arrow function*:

```

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

```

conn arrow function:

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

EJERCICIO: Haz una *arrow function* 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 *arrow function*.

## Estructuras y bucles

### Estructura condicional: if

El **if** es como en la mayoría de los lenguajes. Puede tener asociado un **else** y pueden anidarse varios con **else if**.

```
if (condicion) {
    ...
} else if (condicion2) {
    ...
} else if (condicion3) {
    ...
} else {
    ...
}
```

Ejemplo:

```
if (edad < 18) {
    console.log('Es menor de edad');
} else if (edad > 65) {
    console.log('Está jubilado');
} else {
    console.log('Edad correcta');
}
```

Se puede usar el operador **? :** que es como un *if* que devuelve un valor:

```
let esMayorDeEdad = edad > 18 ? true : false;
```

### Estructura condicional: switch

El **switch** también es como en la mayoría de lenguajes. Hay que poner *break* al final de cada bloque para que no continúe evaluando:

```
switch(color) {
    case 'blanco':
    case 'amarillo': // Ambos colores entran aquí
        colorFondo='azul';
        break;
    case 'azul':
        colorFondo='amarillo';
        break;
    default:           // Para cualquier otro valor
        colorFondo='negro';
```

}

Javascript permite que el *switch* en vez de evaluar valores pueda evaluar expresiones. En este caso se pone como condición *true*:

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

## Bucle while

Podemos usar el bucle *while...do*

```
while (condicion) {
    // sentencias
}
```

que se ejecutará 0 o más veces. Ejemplo:

```
let nota = prompt("Introduce una nota (o cancela para finalizar)");
while (nota) {
    console.log("La nota introducida es: " + nota);
    nota = prompt("Introduce una nota (o cancela para finalizar");
}
```

O el bucle *do...while*:

```
do {
    // sentencias
} while (condicion)
```

que al menos se ejecutará 1 vez. Ejemplo:

```
let nota;
do {
    nota=prompt('Introduce una nota (o cancela para finalizar)');
    console.log('La nota introducida es: '+nota);
} while (nota)
```

**EJERCICIO adivina:** Haz un programa para que el usuario juegue a adivinar un número. Obtén un número al azar (busca por internet cómo se hace o simplemente guarda el número que quieras en una variable) y ve pidiendo al usuario que introduzca un número. Si es el que busca, le dices que lo ha encontrado y si no te lo mostrarás si el número que busca es mayor o menor que el introducido. El juego acaba cuando el usuario encuentra el número o cuando pulsa en 'Cancelar' (en ese caso te mostraremos un mensaje de que ha cancelado el juego).



## DWEC - Javascript Web Cliente.

# JavaScript 01 B – Sintaxis (II)

JavaScript 01 B – Sintaxis (II).....	1
Funciones.....	1
Parámetros.....	1
Funciones anónimas.....	3
Arrow functions (funciones flecha).....	3
Estructuras y bucles .....	4
Estructura condicional: if .....	4
Operador condicional (ternario) .....	4
Estructura condicional: switch .....	4
Bucle while .....	5
Podemos usar el bucle while...do .....	5
O el bucle do...while: .....	5
Bucle: for .....	6
Bucle: for con contador .....	6
Bucle: for...in .....	6
Bucle: for...of .....	7

## Funciones

Se declaran con la palabra reservada **function** y se les pasan los parámetros entre paréntesis. La función puede devolver un valor usando **return** (si no tiene *return* es como si devolviera *undefined*).

Puede usarse una función antes de haberla declarado por el comportamiento de Javascript llamado **hoisting**: el navegador primero carga todas las funciones y mueve las declaraciones de las variables al principio y luego ejecuta el código.

**EJERCICIO mensaje:** Realiza una función que te pida que escribas algo y muestre un alert diciendo 'Has escrito...' y el valor introducido. Pruébala en la consola (pegas allí la función y luego la llamas desde la consola)

## Parámetros

Si se llama a una función con menos parámetros de los declarados el valor de los parámetros no pasados 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;
    }

let resultado = potencia(4); // devolverá 1 ya que no se ejecuta el for
console.log(resultado);
//console.log(potencia(4));   // devolverá 1 ya que no se ejecuta el for

```

Podemos dar un **valor por defecto** a los parámetros por si no los pasan asignándoles el valor al definirlos:

```

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)

```

NOTA: En ES5 para dar un valor por defecto a una variable se hacía:

```

function potencia(base, exponente) {
    exponente = exponente || 2; // si exponente vale undefined se la asigna el valor 2
    ...

```

También es posible acceder a los parámetros desde el array **arguments[]** si no sabemos cuántos parámetros recibiremos:

```

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

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

```

En Javascript las funciones son un tipo de datos más, por lo que podemos hacer cosas como pasarlas como argumento o asignarlas a una variable:

```
const cuadrado=function(value){
```

```

        return value * value;
    }
function aplicarFuncion(dato, funcion_a_aplicar){
    return funcion_a_aplicar(dato);
}
aplicarFuncion(3,cuadrado); // devolverá 9 (3^2)

```

A este tipo de funciones, que son tratadas como cualquier otra variable, se llaman *funciones de primera clase* y son típicas de lenguajes funcionales.

## Funciones anónimas

Podemos definir una función sin darle un nombre. Dicha función puede asignarse a una variable, autoejecutarse o asignarse a un manejador de eventos. Ejemplo:

```

let holaMundo = function() {
    alert('Hola mundo!');
}

holaMundo();           // se ejecuta la función

```

Como vemos, asignamos una función a una variable de forma que podamos “ejecutar” dicha variable.

```

let nuevaVariable = holaMundo; // asigno el valor de holaMundo a otra variable
nuevaVariable(); // Ejecutamos la nueva variable => la función holaMundo.
                  //Para la ejecución se añaden los paréntesis

```

## Arrow functions (funciones flecha)

ES2015 permite declarar una función anónima de forma más corta. Ejemplo sin *arrow function*:

```

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

```

Al escribirla con la sintaxis de una *arrow function* lo que hacemos es:

- Eliminamos la palabra *function*
- Si sólo tiene 1 parámetro podemos eliminar los paréntesis de los parámetros
- Ponemos el símbolo =>
- Si la función sólo tiene 1 línea podemos eliminarnr las {} y la palabra *return*

El ejemplo con *arrow function*:

```

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

```

}

Otro ejemplo, sin *arrow function*:

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

con *arrow function*:

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

EJERCICIO: Haz una *arrow function* 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 *arrow function*.

## Estructuras y bucles

### Estructura condicional: if

El **if** es como en la mayoría de los lenguajes. Puede tener asociado un **else** y pueden anidarse varios con **else if**.

```
if (condicion) {
    ...
} else if (condicion2) {
    ...
} else if (condicion3) {
    ...
} else {
    ...
}
```

Ejemplo:

```
if (edad < 18) {
    console.log('Es menor de edad');
} else if (edad > 65) {
    console.log('Está jubilado');
} else {
    console.log('Edad correcta');
}
```

### Operador condicional (ternario)

Se puede usar el operador **? :** que es como un *if* que devuelve un valor:

```
let esMayorDeEdad = edad > 18 ? true : false;
```

### Estructura condicional: switch

El **switch** también es como en la mayoría de lenguajes. Hay que poner **break** al final de cada bloque para que no continúe evaluando:

```
switch(color) {
    case 'blanco':
    case 'amarillo': // Ambos colores entran aquí
        colorFondo='azul';
        break;
    case 'azul':
        colorFondo='amarillo';
        break;
    default:           // Para cualquier otro valor
        colorFondo='negro';
}
```

Javascript permite que el `switch` en vez de evaluar valores pueda evaluar expresiones. En este caso se pone como condición `true`:

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

## Bucle while

Podemos usar el bucle `while...do`

```
while (condicion) {
    // sentencias
}
```

que se ejecutará 0 o más veces. Ejemplo:

```
let nota = prompt("Introduce una nota (o cancela para finalizar)");
while (nota) {
    console.log("La nota introducida es: " + nota);
    nota = prompt("Introduce una nota (o cancela para finalizar)");
}
```

O el bucle `do...while`:

```
do {
    // sentencias
} while (condicion)
```

que al menos se ejecutará 1 vez. Ejemplo:

```
let nota;
do {
    nota=prompt('Introduce una nota (o cancela para finalizar)');
    console.log('La nota introducida es: '+nota);
```

```
}
```

EJERCICIO: Haz un programa para que el usuario juegue a adivinar un número. Obtén un número al azar (busca por internet cómo se hace o simplemente guarda el número que quieras en una variable) y ve pidiendo al usuario que introduzca un número. Si es el que busca le dices que lo ha encontrado y si no te mostrarás si el número que busca es mayor o menor que el introducido. El juego acaba cuando el usuario encuentra el número o cuando pulsa en ‘Cancelar’ (en ese caso le mostraremos un mensaje de que ha cancelado el juego).

## Bucle: for

Tenemos muchos *for* que podemos usar.

### Bucle: for con contador

Creamos una variable contador que controla las veces que se ejecuta el *for*:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let i=0; i<datos.length; i++) {
    sumaDatos += datos[i];
}
// El valor de sumaDatos será 125
```

EJERCICIO: El factorial de un número entero n es una operación matemática que consiste en multiplicar ese número por todos los enteros menores que él:  $n \times (n-1) \times (n-2) \times \dots \times 1$ . Así, el factorial de 5 (se escribe 5!) vale  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ . Haz un script que calcule el factorial de un número entero positivo.

### Bucle: for...in

El bucle se ejecuta una vez para cada elemento del array (o propiedad del objeto) y se crea una variable contador que toma como valores la posición del elemento en el array:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let indice in datos) {
    sumaDatos += datos[indice];      // Los valores que toma indice son 0, 1, 2, 3
}
// El valor de sumaDatos será 125
```

También sirve para recorrer las propiedades de un objeto:

```
let profe={
    nom:'Santi',
    ape1:'Blanco',
    ape2:'Arenal'
}

let nombre='';

for (var campo in profe) {          // no declarar campo con let
    nombre += profe.campo + ' ';    // o profe[campo];
```

```
}
```

```
// El valor de nombre será 'Santiago Blanco Arenal'
```

### Bucle: `for...of`

Es similar al `for...in` pero la variable contador en vez de tomar como valor cada índice toma cada elemento. Es nuevo en ES2015:

```
let sumaDatos = 0;
```

```
for (let valor of datos) {
```

```
    sumaDatos += valor;           // los valores que toma valor son 5, 23, 12, 85
```

```
}
```

```
// El valor de sumaDatos será 125
```

También sirve para recorrer los caracteres de una cadena de texto:

```
let cadena = 'Hola';
```

```
for (let letra of cadena) {
```

```
    console.log(letra);         // los valores de letra son 'H', 'o', 'l', 'a'
```

```
}
```

EJERCICIO: Haz 3 funciones a las que se le pasa como parámetro un array de notas y devuelve la nota media. Cada una usará un `for` de una de las 3 formas vistas. Pruébalas en la consola.



## DWEC – Javascript Web Cliente.

# JavaScript 01 – Sintaxis (III)

JavaScript 01 – Sintaxis .....	1
Tipos de datos básicos.....	1
Casting de variables .....	1
Number.....	2
Veamos algunos ejemplos de código y la salida en consola: .....	2
Otras funciones útiles son: .....	3
String.....	4
Template literals.....	5
Boolean.....	5

## Tipos de datos básicos

Para saber de qué tipo es el valor de una variable tenemos el operador **typeof**. Ej.:

- `typeof 3` devuelve *number*
- `typeof 'Hola'` devuelve *string*

En Javascript hay 2 valores especiales:

- **undefined**: es lo que vale una variable a la que no se ha asignado ningún valor.
- **null**: es un tipo de valor especial que podemos asignar a una variable. Es como un objeto vacío (`typeof null` devuelve *object*)

También hay otros valores especiales relacionados con operaciones numéricas (o con números):

- **NaN (Not a Number)**: indica que el resultado de la operación no puede ser convertido a un número (ej. `'Hola'*2`, aunque `'2'*2` daría 4 ya que se convierte la cadena '2' al número 2)
- **Infinity y -Infinity**: indica que el resultado es demasiado grande o demasiado pequeño (ej. `1/0` o `-1/0`)

## Casting de variables

Como hemos dicho, las variables pueden contener cualquier tipo de valor.

En las operaciones, Javascript realiza **automáticamente** las conversiones necesarias para, si es posible, realizar la operación.

Ejemplos:

- `'4' / 2` devuelve 2 (convierte '4' en 4 y realiza la operación)

- `'23' - null` devuelve 0 (hace  $23 - 0$ )
- `'23' - undefined` devuelve `Nan` (no puede convertir undefined a nada así que no puede hacer la operación)
- `'23' * true` devuelve 23 ( $23 * 1$ )
- `'23' * 'Hello'` devuelve `Nan` (no puede convertir 'Hello')
- `23 + 'Hello'` devuelve '23Hello' (+ es el operador de concatenación así que convierte 23 a '23' y los concatena)
- `23 + '23'` devuelve 2323 (OJO, convierte 23 a '23', no al revés)

Ten en cuenta que en Javascript todo son objetos, por lo que todo tiene métodos y propiedades. Veamos brevemente los tipos de datos básicos.

EJERCICIO: Prueba en la consola las operaciones anteriores y alguna más con la que tengas dudas de qué devolverá.

## Number

Sólo hay 1 tipo de números, no existen enteros y decimales. El tipo de dato para cualquier número es `number`. El carácter para la coma decimal es el `,` (como en inglés, así que 23,12 debemos escribirlo como 23.12).

Tenemos los operadores aritméticos `+, -, *, /` y `%` y los unarios `++` y `--`

Existen los valores especiales `Infinity` y `-Infinity` ( $23/0$  no produce un error sino que devuelve `Infinity`).

Podemos usar los operadores aritméticos junto al operador de asignación `= (+=, -=, *=, /= y %=)`.

Algunos métodos útiles de los números son:

- `.toFixed(num)`: redondea el número a los decimales indicados. Ej. `23.2376.toFixed(2)` devuelve 23.24
- `.toLocaleString()`: devuelve el número convertido al formato local. Ej. `23.76.toLocaleString()` devuelve '23,76' (Los navegadores en español convierten el punto decimal en coma y ponen el punto separador de miles)

Podemos forzar la conversión a número con la función `Number(valor)`. Ejemplo `Number('23.12')` devuelve 23.12

Veamos algunos ejemplos de código y la salida en consola:

```

let num=23.3333333333;
console.log(typeof(num));
console.log(num);

```

```

num= 45212.444444
console.log(typeof(num));
console.log(num.toLocaleString());

```

```

let num2 = 36.2222;
console.log(typeof(num2));
num2=num2.toLocaleString();
var console: Console 2);
console.log(num2);

```

```
let num2 = 36.2222;
console.log(typeof(num2));
num2=num2.toLocaleString();
console.log(typeof(num2));
console.log(num2);
```

```
num2=888; //ojo
console.log(num2);
num2=num2.toLocaleString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
>
```

```
num2=888.8888; //ojo
console.log(num2);
num2=num2.toLocaleString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
>
```

En el caso anterior, no es numérico porque después de convertir `toLocaleString()`, no puede volver a numérico.

Si se convierte con el método `toString()`, sí que puede retornar a numérico, como vemos en el siguiente ejemplo:

```
num2=888.8888; //ojo
console.log(num2);
num2=num2.toString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
>
```

Otras funciones útiles son:

- **`isNaN(valor)`**: nos dice si el valor pasado es un número (false), o no (true)
- **`isFinite(valor)`**: devuelve true si el valor es finito (no es `Infinity` ni `-Infinity`).
- **`parseInt(valor)`**: convierte el valor pasado a un número entero. Siempre que comience por un número la conversión se podrá hacer. Ej.:

```
parseInt(3.65)      // Devuelve 3
parseInt('3.65')    // Devuelve 3
parseInt('3 manzanas') // Devuelve 3, Number devolvería NaN
```

- **`parseFloat(valor)`**: como la anterior, pero conserva los decimales

OJO: al sumar números decimales (*floats*) podemos tener problemas:

```
console.log(0.1 + 0.2) // imprime 0.3000000000000004
```

Para evitarlo:

- Redondead los resultados.
- También se puede hacer una operación similar a  $(0.1*10 + 0.2*10) / 10$ .

console.log((0.1 + 0.2));	0.3000000000000004
console.log((0.1 + 0.2).toFixed(1));	0.3
console.log( (0.1*10 + 0.2*10) / 10);	0.3

EJERCICIO: Modifica la función de calcular la nota media para que devuelva la media con 1 decimal

EJERCICIO: Modifica la función que devuelve el cubo de un número para que compruebe si el parámetro pasado es un número entero. Si no es un entero o no es un número mostrará un alert indicando cuál es el problema y devolverá false.

## String

Las cadenas de texto van entre comillas simples o dobles, es indiferente. Podemos escapar un carácter con \ para poder usarlo dentro de la cadena. (Ejemplo: 'Hola \'Mundo\' ' devuelve Hola 'Mundo').

console.log("hola \"majo\"");	hola "majo"
console.log('Hola "majo"');	Hola "majo"

Para forzar la conversión a cadena se usa la función **String(valor)** (ej. String(23) devuelve '23')

El operador de concatenación de cadenas es +. Ojo porque si pedimos un dato con **prompt** siempre devuelve una cadena así que si le pedimos la edad al usuario (por ejemplo 20) y se sumamos 10 tendremos 2010 ('20'+10).

Algunos métodos y propiedades de las cadenas son:

- .length**: devuelve la longitud de una cadena. Ej.: 'Hola mundo'.length devuelve 10
- .charAt(posición)**: 'Hola mundo'.charAt(0) devuelve 'H'
- .indexOf(carácter)**: 'Hola mundo'.indexOf('o') devuelve 1. Si no se encuentra devuelve -1
- .lastIndexOf(carácter)**: 'Hola mundo'.lastIndexOf('o') devuelve 9
- .substring(desde, hasta)**: 'Hola mundo'.substring(2,4) devuelve 'la'
- .substr(desde, num caracteres)**: 'Hola mundo'.substr(2,4) devuelve 'la m'
- .replace(busco, reemplaza)**: 'Hola mundo'.replace('Hola', 'Adiós') devuelve 'Adiós mundo'
- .toLocaleLowerCase()**: 'Hola mundo'.toLocaleLowerCase() devuelve 'hola mundo'
- .toLocaleUpperCase()**: 'Hola mundo'.toLocaleUpperCase() devuelve 'HOLA MUNDO'
- .localeCompare(cadena)**: devuelve -1 si la cadena a que se aplica el método es anterior alfabéticamente a 'cadena', 1 si es posterior y 0 si ambas son iguales. Tiene en cuenta caracteres locales como acentos ñ, ç, etc
- .trim(cadena)**: ' Hola mundo '.trim() devuelve 'Hola mundo'
- .startsWith(cadena)**: 'Hola mundo'.startsWith('Hol') devuelve true
- .endsWith(cadena)**: 'Hola mundo'.endsWith('Hol') devuelve false
- .includes(cadena)**: 'Hola mundo'.includes('mun') devuelve true
- .repeat(veces)**: 'Hola mundo'.repeat(3) devuelve 'Hola mundoHola mundoHola mundo'

- .split(separador):** 'Hola mundo'.split(' ') devuelve el array ['Hola', 'mundo']. 'Hola mundo'.split('') devuelve el array ['H', 'o', 'l', 'a', '', 'm', 'u', 'n', 'd', 'o']

Podemos probar los diferentes métodos en la página de [w3schools](#).

EJERCICIO: Haz una función a la que se le pasa un DNI (ej. 12345678w o 87654321T) y devolverá si es correcto o no. La letra que debe corresponder a un DNI correcto se obtiene dividiendo la parte numérica entre 23 y cogiendo de la cadena 'TRWAGMYFPDXBNJZSQVHLCKE' la letra correspondiente al resto de la división. Por ejemplo, si el resto es 0 la letra será la T y si es 4 será la G. Prueba la función en la consola con tu DNI

### Template literals

Desde ES2015 también podemos poner una cadena entre ` (acento grave) y en ese caso podemos poner dentro variables y expresiones que serán evaluadas al ponerlas dentro de \${}. También se respetan los saltos de línea, tabuladores, etc que haya dentro. Ejemplo:

<pre>let edad=25; console.log(`El usuario tiene: \${edad} años`);</pre>	<pre>El usuario tiene: 25 años</pre>
---	--------------------------------------

### Boolean

Los valores booleanos son **true** y **false**. Para convertir algo a booleano se usar **Boolean(valor)** aunque también puede hacerse con la doble negación (!!). Cualquier valor se evaluará a **true** excepto 0, NaN, null, undefined o una cadena vacía ("") que se evaluarán a **false**.

console.log(Boolean("pepe"));	→ true
console.log(!!( "pepe" ));	→ true
console.log(!!( "pepe" ));	→ false
console.log(!null);	→ true
console.log(!!null);	→ false
console.log(!!" ");	→ true
console.log(!!"");	→ false
console.log(!!undefined);	→ false

Los operadores lógicos son ! (negación), && (and), || (or).

Para comparar valores tenemos == y ===. La triple igualdad devuelve **true** si son igual valor y del mismo tipo.

Como Javascript hace conversiones de tipos automáticas conviene usar la === para evitar cosas como:

- '3' == 3 true
- 3 == 3.0 true
- 0 == false true
- '' == false true
- '' == false true
- [] == false true
- null == false false
- undefined == false false
- undefined == null true

También tenemos 2 operadores lógicos para *diferente*: `!=` y `!==` que se comportan como hemos dicho antes.

Los operadores relacionales son `>`, `>=`, `<`, `<=`. Cuando se compara un número y una cadena ésta se convierte a número y no al revés (`23 > '5'` devuelve `true`, aunque `'23' > '5'` devuelve `false`)

<code>console.log( 23 &gt; 5 );</code>	<code>true</code>
<code>console.log( '23' &gt; '5' );</code>	<code>false</code>
<code>console.log( '23' &gt; 5 );</code>	<code>true</code>
<code>console.log( 23 &gt; '5' );</code>	<code>true</code>
	.



## DWEC - Javascript Web Cliente.

# JavaScript 01 – Sintaxis (IV)

JavaScript 01 – Sintaxis (IV).....	1
Manejo de errores .....	1
Buenas prácticas .....	4
'use strict' .....	4
Variables .....	4
Otras .....	4
Clean Code.....	5

## Manejo de errores

Si sucede un error en nuestro código el programa dejará de ejecutarse, por lo que el usuario tendrá la sensación de que no hace nada (el error sólo se muestra en la consola y el usuario no suele abrirla nunca). Para evitarlo debemos intentar capturar los posibles errores de nuestro código antes de que se produzcan.

Pero hay una construcción sintáctica `try...catch` que nos permite "atrapar" errores para que el script pueda, en lugar de morir, hacer algo más razonable.

```
try {
  // código...
} catch (err) {
  // manipulación de error
}
```

Dentro del bloque `try` ponemos el código que queremos proteger y cualquier error producido en él será pasado al bloque `catch` donde es tratado. Opcionalmente podemos tener al final un bloque `finally` que se ejecuta tanto si se produce un error como si no.

Funciona así:

- Primero, se ejecuta el código en `try { ... }`.
- Si no hubo errores, se ignora `catch (err)`: la ejecución llega al final de `try` y continúa, omitiendo `catch`.
- Si se produce un error, la ejecución de `try` se detiene y el control fluye al comienzo de `catch (err)`.
- La variable `err` (podemos usar cualquier nombre para ella) contendrá un objeto de error con detalles sobre lo que sucedió.

El parámetro que recibe `catch` es un objeto con las propiedades `name`, que indica el tipo de error (`SyntaxError, RangeError, ...` o el genérico `Error`), y `message`, que indica el texto del error producido.

The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there is some JavaScript code:

```
try {
  jaja
} catch (err) {
  console.log(err);
  console.log(err.name);
  console.log(err.message);
}
```

On the right, the console output shows three error messages:

- ReferenceError: jaja is not defined at funciones.js:10:3
- ReferenceError
- jaja is not defined

Para que `try..catch` funcione, el código debe ser ejecutable. En otras palabras, debería ser JavaScript válido.

No funcionará si el código es sintácticamente incorrecto, por ejemplo, si hay llaves sin cerrar.

El motor de JavaScript primero lee el código y luego lo ejecuta. Los errores que ocurren en la fase de lectura se denominan errores de "tiempo de análisis" y son irrecuperables (desde dentro de ese código). Eso es porque el motor no puede entender el código.

Entonces, `try...catch` solo puede manejar errores que ocurren en un código válido. Dichos errores se denominan "errores de tiempo de ejecución" o, a veces, "excepciones".

En ocasiones podemos querer que nuestro código genere un error. Esto evita que tengamos que comprobar si el valor devuelto por una función es el adecuado o es un código de error.

### Ejemplo:

Tenemos una función para retirar dinero de una cuenta que recibe:

- el saldo de la cuenta
- y la cantidad de dinero a retirar

La función devuelve:

- el nuevo saldo, pero si no hay suficiente saldo no debería restar nada sino mostrar un mensaje al usuario.

Sin gestión de errores haríamos:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    return false;
  }
  return saldo - cantidad;
}

let saldo = 30;
cantidad = 200;
let resultado = retirar(saldo, cantidad);

if (resultado === false) {
  console.log("Saldo insuficiente");
} else {
  saldo = resultado;
}
```

Se trata de un código poco claro que podemos mejorar lanzando un error en la función. Para ello se utiliza la instrucción `throw`:

```
if (saldo < cantidad) {
  throw 'Saldo insuficiente'
}
```

Por defecto al lanzar un error, este será de clase `Error` pero (el código anterior es equivalente a `throw new Error('Valor no válido')`) aunque podemos lanzarlo de cualquier otra clase (`throw new RangeError('Saldo insuficiente')`) o personalizarlo.

Siempre que vayamos a ejecutar código que pueda generar un error debemos ponerlo dentro de un bloque `try`.

Por lo que la llamada a la función que contiene el código anterior debería estar dentro de un `try`. El código del ejemplo anterior quedaría:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    throw "Saldo insuficiente";
  }
  return saldo - cantidad;
}
// Siempre debemos llamar a esa función desde un bloque _try_

let saldo = 30;
let importe = 200;

try {
  saldo = console.log(`Nuevo saldo: ${retirar(saldo, importe)}`);
} catch (err) {
  console.log(err); // muestra "Saldo Insuficiente"
}

try {
  saldo = console.log(`Nuevo saldo: ${retirar(200, 5)}`); //Muestra "Nuevo Saldo: 195"
} catch (err) {
  console.log(err);
}
```

Podemos ver en detalle cómo funcionan en la página de [MDN web docs](#) de Mozilla.

Try...catch trabaja sincrónicamente

Si ocurre una excepción en el código “programado”, como en `setTimeout`, entonces `try..catch` no lo detectará.

Más información en: <https://es.javascript.info/try-catch>

**EJERCICIO:** Escribe una función que devuelva la nota media de un array de notas. Si el array está vacío debe avisar del error. Prueba la función con `try catch`:

## Buenas prácticas

Javascript nos permite hacer muchas cosas que otros lenguajes no nos dejan por lo que debemos ser cuidadosos para no cometer errores de los que no se nos va a avisar.

### 'use strict'

Si ponemos siempre esta sentencia al principio de nuestro código el intérprete nos avisará si usamos una variable sin declarar (muchas veces por equivocarnos al escribir su nombre). En concreto fuerza al navegador a no permitir:

- Usar una variable sin declarar
- Definir más de 1 vez una propiedad de un objeto
- Duplicar un parámetro en una función
- Usar números en octal
- Modificar una propiedad de sólo lectura

### Variables

Algunas de las prácticas que deberíamos seguir respecto a las variables son:

- Elegir un buen nombre es fundamental. Evitar abreviaturas o nombres sin significado (a, b, c, ...)
- Evitar en lo posible variables globales
- Usar `let` para declararlas
- Usar `const` siempre que una variable no deba cambiar su valor
- Declarar todas las variables al principio
- Inicializar las variables al declararlas
- Evitar conversiones de tipo automáticas
- Usar, para nombrarlas, la notación `camelCase`

También es conveniente, por motivos de eficiencia no usar objetos `Number`, `String` o `Boolean`, sino los tipos primitivos (no usar `let numero = new Number(5)`, sino `let numero = 5`) y lo mismo al crear arrays, objetos o expresiones regulares (no usar `let miArray = new Array()`, sino `let miArray = []`).

### Otras

Algunas reglas más que deberíamos seguir son:

- Debemos ser coherentes a la hora de escribir código: por ejemplo, podemos poner (recomendado) o no espacios antes y después del `=` en una asignación, pero debemos hacerlo siempre igual. Existen muchas guías de estilo y muy buenas: [Airbnb](#), [Google](#), [Idiomatic](#), etc. Para obligarnos a seguir las reglas podemos usar alguna herramienta [linter](#).
- También es conveniente para mejorar la legibilidad de nuestro código separar las líneas de más de 80 caracteres.
- Usar `==` en las comparaciones
- Si un parámetro puede faltar al llamar a una función, darle un valor por defecto
- Y para acabar: comentar el código cuando sea necesario, pero mejor que el código sea lo suficientemente claro como para no necesitar comentarios

## Clean Code

Estas y otras muchas recomendaciones se recogen en el libro [Clean Code](#) de *Robert C. Martin* y en muchos otros libros y artículos. Aquí se puede ver un pequeño resumen traducido al castellano:

<https://github.com/devictoribero/clean-code-javascript>



## DWEC – Javascript Web Cliente.

JavaScript 02 – Arrays (I).....	2
Introducción.....	2
Objetos JavaScript.....	2
Los arrays en JavaScript .....	2
Operaciones con Arrays .....	3
length.....	3
Añadir elementos.....	3
Eliminar elementos .....	3
splice .....	3
slice .....	4
Arrays y Strings .....	4
sort.....	5
Otros métodos comunes.....	6
Functional Programming.....	7
filter.....	7
find.....	8
findIndex .....	9
every / some .....	9
map .....	9
reduce .....	9
forEach.....	12
includes .....	12
Array.from.....	12
Referencia vs Copia.....	13
Rest y Spread .....	14
Desestructuración de arrays .....	15
Los objetos Map y Set .....	16
Map.....	16
Set .....	17

# JavaScript 02 – Arrays (I)

## Introducción

### Objetos JavaScript

Los objetos son uno de los tipos de datos de JavaScript.

Los objetos se utilizan para almacenar colecciones de clave/valor (nombre/valor).

Un objeto JavaScript es una colección de valores con nombre.

El siguiente ejemplo crea un objeto JavaScript con cuatro propiedades clave/valor:

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

### Métodos y propiedades de objetos de JavaScript

- **Constructor** Returns the function that created an object's prototype
- **keys()** Returns an Array Iterator object with the keys of an object
- **prototype** Let you to add properties and methods to JavaScript objects
- **toString()** Converts an object to a string and returns the result
- **valueOf()** Returns the primitive value of an object

## Los arrays en JavaScript

Los arrays, también llamados arreglos, vectores, matrices, listas..., son un tipo de objeto que no tiene tamaño fijo, podemos añadirle elementos en cualquier momento.

Para hacer referencia a (referenciar) los elementos se hace con un índice numérico. A diferencia de los objetos que se referencian con un nombre.

Podemos crearlos como instancias del objeto Array:

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

Pero lo recomendado es crearlos usando notación JSON:

```
let a = []
let 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
```

## Operaciones con Arrays

Los arrays tienen las mismas propiedades y métodos que los objetos, y muchos más que son propios de los 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:

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

### Añadir elementos

Podemos añadir elementos al final de un array con el método `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]
```

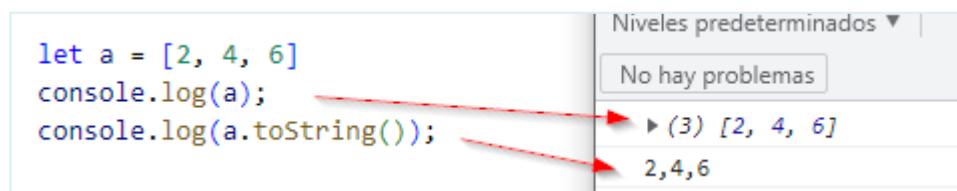
Es muy útil para hacer una copia de un array:

```
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 el caso de los arrays esta función devuelve una cadena con los elementos del array 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 `.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.

OJO!! El array original queda modificado con el nuevo orden.

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

Si no se especifica otra cosa, el orden que se sigue es el de los **códigos ascii**, por lo que los dígitos numéricos van antes que las letras, y las mayúsculas antes que las minúsculas.

También podemos pasarle una función que le indique cómo ordenar. Esta función debe devolver un valor negativo si el primer elemento es menor, positivo si es mayor, 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;
  if (elem1.toLocaleLowerCase() = elem2.toLocaleLowerCase()) return 0;
});
// b = ["adios", "Bien", "hola", "Mal"]
```

También se utiliza para ordenar números, tanto de forma ascendente como descendente:

```
let a=[20,4,87,2];
let b= a.sort(function(elem1,elem2){return elem1-elem2}) //ascendente
console.log(b);

b= a.sort(function(elem1,elem2){return elem2 - elem1}) //descendente
console.log(b);
```

Es frecuente utilizar esta función es para ordenar arrays de objetos. Por ejemplo, si tenemos un objeto *persona* con los campos *nombre* y *edad*, para ordenar un array de objetos persona por su edad haremos:

```
let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];
```

```
let personasOrdenado = personas.sort(function(persona1, persona2) {
    return persona1.edad - persona2.edad
})
```

```
let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort(function(persona1, persona2) {
    return persona1.edad - persona2.edad
});
console.log(personas);
```

No hay problemas

▼ (3) [{} , {} , {} ] ↴  
 ► 0: {nombre: 'Juan', edad: 25}  
 ► 1: {nombre: 'Ana', edad: 33}  
 ► 2: {nombre: 'Benito', edad: 52}  
 length: 3  
 ► [[Prototype]]: Array(0)

Usando arrow functions quedaría más sencillo:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.edad - persona2.edad)
```

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

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
```

```
let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
console.log(personas);
```

No hay problemas

▼ (3) [{} , {} , {} ] ↴  
 ► 0: {nombre: 'Ana', edad: 33}  
 ► 1: {nombre: 'Benito', edad: 52}  
 ► 2: {nombre: 'Juan', edad: 25}  
 length: 3  
 ► [[Prototype]]: Array(0)

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

```
let a = [2, 4, 6, 4]
console.log(a.indexOf(4)); // 1
```

```
console.log(a.lastIndexOf(4));      // 3
console.log(a.lastIndexOf('4')));   // -1
```

## 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, además de pasarle como parámetro la lista sobre la que debe actuar, se le pasa como segundo 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.

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; i++) {
  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
})
// aprobados = [5.2, 6, 9.75, 7.5]
```

Y usando funciones tipo flecha 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)
// aprobados = [5.2, 6, 9.75, 7.5]
```

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 ninguno). Ejemplo:

```
let arrayNotas = [4, 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 un coche de color rojo dentro de un array llamado *coches* cuyos elementos son objetos con un campo 'color', haremos:

```
let coches = [
  {
    "color": "morado",
    "tipo": "berlina",
    "capacidad": 7
  },
  {
    "color": "rojo",
    "tipo": "camioneta",
    "capacidad": 5
  },
  {
    "color": "rojo",
    "tipo": "furgón",
    "capacidad": 7
  }
]

let cocheBuscado = coches.find(coche => coche.color === 'rojo') // devolverá el objeto completo del primer elemento que cumpla la condición.
```

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

## findIndex

Como `find` pero, en vez de devolver el elemento, devuelve su posición (-1 si ningún elemento cumple la condición).

```
let cocheBuscado = coches.findIndex(coche => coche.color === 'rojo') // devolverá 1
```

En el ejemplo de los coches el valor devuelto sería 1, ya que el segundo elemento cumple la condición.

Al igual que el anterior, `findIndex` 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

- `every` devuelve **true** si **TODOS** los elementos del array cumplen la condición y **false** en caso contrario.
- `some` 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
```

EJERCICIO: Dado un array con los días de la semana indica si algún día empieza por 'S'. Dado un array con los días de la semana indica si todos los días acaban por 's'

## map

`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 * 0.1);
```

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

## reduce

El método `reduce`:

- Ejecuta una función tipo callback para cada elemento del array, y devuelve un único valor calculado a partir de los elementos del array, este valor es el resultado acumulado de las llamadas a la función.
- No se ejecuta la función para elementos vacíos del array.
- No se cambia el contenido del array original.

Sintaxis: `array.reduce(function(total, currentValue, currentIndex, vector), initialValue)`

Hay dos parámetros: una función e initialValue (este segundo parámetro es opcional).

La función admite 4 parámetros:

- `total` o `valorAnterior` (obligatorio), contiene el valor devuelto por la función en cada llamada.
- `currentValue` (obligatorio), contiene el valor del elemento actual.

- **currentIndex** (opcional), contiene el índice del elemento actual.
- **vector** (opcional), es el array al que pertenece el elemento actual.

La primera vez que se llama la función, `valorAnterior` y `valorActual` pueden tener uno de dos posibles valores:

- Si se proveyó un `valorInicial` al llamar a `reduce`, entonces `valorAnterior` será igual al `valorInicial` y `valorActual` será igual al primer elemento del array.
- Si no se proveyó un `valorInicial`, entonces `valorAnterior` será igual al primer valor en el array y `valorActual` será el segundo.

Ejemplo: queremos obtener la suma de las notas de un array:

```
let arrayNotas = [4,7,5];
let suma=0;

suma= arrayNotas.reduce((total, valor) => total+= valor, 0);
console.log(suma); // obtiene 16

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor) => {
    return total+= valor
}, 0);
console.log(suma); // obtiene 16
```

La función segunda permite añadir fácilmente otras funcionalidades

Para observar cómo funciona `reduce()`, estudiaremos este caso para sumar el array de notas. Se ha añadido el parámetro del índice a la función callback.

Si añadimos un `valorInicial`, hace un recorrido por el índice cero. Si este valor es cero, el resultado es el mismo que sin no ponemos `valorInicial`.

<pre>let arrayNotas = [4,7,5]; let suma=0;  // Misma función para sumar con el parámetro índice suma= arrayNotas.reduce((total, valor, i) =&gt; {     console.log('Valor: ' + valor);     console.log('Índice: ' + i);     return total+= valor; }, 0); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Niveles predeterminados No hay problemas</p> <table border="1"> <tr><td>Valor: 4</td></tr> <tr><td>Índice: 0</td></tr> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 4	Índice: 0	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16	<pre>let arrayNotas = [4,7,5]; let suma=0;  // Misma función para sumar con el parámetro índice suma= arrayNotas.reduce((total, valor, i) =&gt; {     console.log('Valor: ' + valor);     console.log('Índice: ' + i);     return total+= valor; }, ); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Filtrar Niveles predeterminados No hay problemas</p> <table border="1"> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16
Valor: 4															
Índice: 0															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															

En el siguiente ejemplo hay dos casos. Entre el caso de la izda y el de la derecha solo cambia el valor del parámetro opcional `initialValue` que en este caso vale 3, por lo que el resultado cambia.

<pre>let arrayNotas = [4,7,5]; let suma=0;  // Ahora la misma función sin ser tipo flecha suma= arrayNotas.reduce((total, valor, i) =&gt; {     console.log('Valor: ' + valor);     console.log('Índice: ' + i);     return total+= valor; },); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Filtrar Niveles predeterminados No hay problemas</p> <table border="1"> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16	<pre>let arrayNotas = [4,7,5]; let suma=0;  // Ahora la misma función sin ser tipo flecha suma= arrayNotas.reduce((total, valor, i) =&gt; {     console.log('Valor: ' + valor);     console.log('Índice: ' + i);     return total+= valor; }, 3); console.log('Resultado: ' + suma); // obtiene 19</pre>	<p>Filtrar Niveles predeterminados No hay problemas</p> <table border="1"> <tr><td>Valor: 4</td></tr> <tr><td>Índice: 0</td></tr> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 19</td></tr> </table>	Valor: 4	Índice: 0	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 19
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															
Valor: 4															
Índice: 0															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 19															

Otros ejemplos similares de suma de notas:

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

```
let sumaNotas = arrayNotas.reduce((total,nota) => total += nota, 30) // total = 65.35
sumaNotas = arrayNotas.reduce((total,nota) => total += nota) // total = 35.35
```

Podemos tener la función declarada y utilizarla con reduce(). Ejemplo:

```
function suma(a, b) {
    return a + b;
}

const numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90];
const resultado = numeros.reduce(suma);
console.log(resultado); // 450
```

Otro 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
```

Ejemplo muy útil: Integrar un array a partir de varios arrays

```
var integrado = [[0,1], [2,3], [4,5]].reduce(function(a,b) {
    return a.concat(b);
});
// integrado es [0, 1, 2, 3, 4, 5]
```

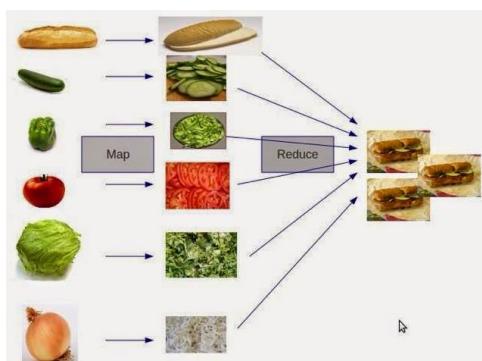
Ejemplo clarificador de la idea: Cadena de montaje, se van añadiendo piezas a la cadena de texto inicial.

```
const partesDelCoche = ["asientos", "volante", "puertas", "ruedas", "pintura metalizada"];

const coche = partesDelCoche.reduce(function (valorAnterior, valorActual) {
    return `${valorAnterior} ${valorActual},`;
}, "Mi coche tiene: ");

console.log(coche);
//Mi coche tiene: asientos, volante, puertas, ruedas, pintura metalizada,
```

Una idea del funcionamiento de map y reduce sería: 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 sandwich) con todas ellas.



EJERCICIO: Dado el array de notas visto anteriormente, usar un método para que devuelva 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 * 1.1)
```

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 con las colecciones.

Por ejemplo, si queremos mostrar por consola cada párrafo de la página que comience por la palabra ‘Si’ 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 parrafos = document.getElementsByTagName('p');
let arrayParrafos = Array.from(parrafos);
// y ya podemos usar los métodos que queramos:
arrayParrafos.filter(parrafo => parrafo.textContent.startsWith('Si'))
  .forEach(parrafo => alert(parrafo.textContent))
```

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

Existen otros métodos para utilizar con arrays, se pueden ver en:

[https://www.w3schools.com/jsref/jsref\\_obj\\_array.asp](https://www.w3schools.com/jsref/jsref_obj_array.asp)

## Referencia vs Copia

Cuando copiamos una variable de tipo `boolean`, `string` o `number`, o se pasa esa variable como parámetro a una función, se hace una copia de la misma. Por lo que, 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('2022-09-23')
let fecha2 = fecha1          // fecha1 = '2022-09-23'    fecha2 = '2022-09-23'
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, éstos se continúan pasando por referencia. En ese caso lo más sencillo sería utilizar `objetoCopia = JSON.stringify(objetoOriginal);`

**JSON** es un es un formato de texto para almacenar y transportar datos.

JavaScript tiene una función integrada para convertir cadenas JSON en objetos JavaScript: `JSON.parse()`

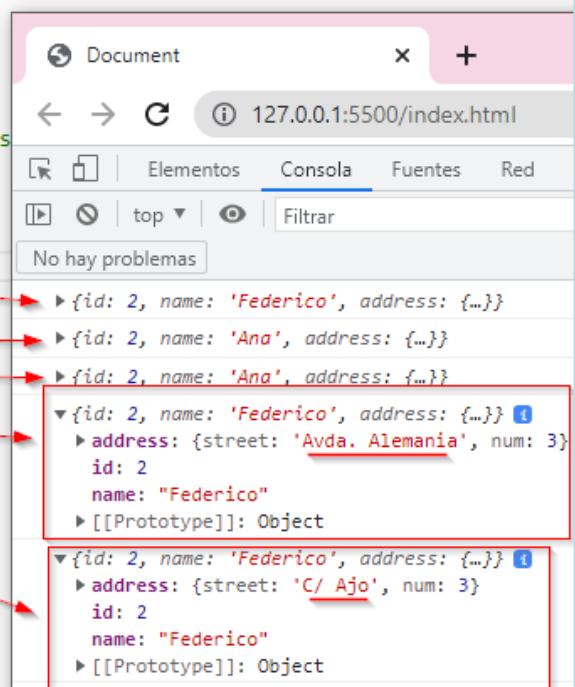
JavaScript tiene una función integrada para convertir un objeto en una cadena JSON: `JSON.stringify()`

```
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
```

```
'use strict';
let a = {id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3} }
let b = a;
let copia1= Object.assign({}, a);

//solución:
// ahora ambos objetos contienen lo mismo pero son diferentes
let copiaDeA = JSON.parse(JSON.stringify(a))

console.log((b));
a.name= 'Ana';
a.address.street = 'Avda. Alemania'
console.log(a);
console.log(b);
console.log(copia1);
console.log(copiaDeA);
```



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

**Rest** se utiliza para convertir en un array un grupo de parámetros (*rest*). El operador es ... (3 puntos).

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

```
function transformaRest(...datos) {
    return datos;
}

console.log(transformaRest(1,2,3,));
```

Ejemplo para utilizar con *reduce()*:

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

// le pasamos un número variable de parámetros
console.log(notaMedia(5.2, 3.9, 6, 9.75, 7.5, 3));
```

**Spread** permite pasar como parámetros independientes los elementos de un array o string. El operador también es ... (3 puntos).

Si lo que queremos es convertir un array en un grupo de elementos haremos *spread*.

```

let array = [1,2,3];
console.log(array);
// hacemos la conversión de array a parámetros
console.log(...array);

```

The screenshot shows a browser's developer tools console. The first line of code `let array = [1,2,3];` is executed, followed by `console.log(array);`. The output is `(3) [1, 2, 3]` with arrows pointing to the array elements. Then, the second line of code `// hacemos la conversión de array a parámetros` is executed, followed by `console.log(...array);`. The output is `1 2 3` with arrows pointing to each individual number.

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 maximo = Math.max(arrayNotas); //maximo = NaN
// si hacemos Math.max(arrayNotas) devuelve NaN porque arrayNotas es un array y no un número

// hacemos la conversión de array a parámetros
let maximoSpread = Math.max(...arrayNotas); // maxNota = 9.75

```

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

```

## 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 desestructurararlo:

```

const miProducto = {
  id: 5,
  name: 'TV Samsung',
  units: 3,
  price: 395.95
};

// Se puede abreviar: function muestraNombre({name, units}) {

```

```
function muestraNombre({name: name, units: units}) {
    console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}
muestraNombre(miProducto); //Del producto TV Samsung hay 3 unidades
```

También podemos asignar valores por defecto:

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

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

## Los objetos Map y Set

En el ámbito de JavaScript, a menudo los desarrolladores pasan mucho tiempo decidiendo la estructura de datos correcta que se usará. Esto se debe a que elegir la estructura de datos correcta puede facilitar la manipulación de esos datos posteriormente, con lo cual se puede ahorrar tiempo y facilitar la comprensión del código.

Las dos estructuras de datos predominantes para almacenar conjuntos de datos son los **Object (Objetos)** y **Array** (un tipo de objeto).

Los desarrolladores utilizan **Object** para almacenar pares clave-valor y **Array** para almacenar listas indexadas.

Sin embargo, para dar más flexibilidad a los desarrolladores, en la especificación ECMAScript 2015 se introdujeron dos nuevos tipos de objetos iterables:

- los **Map**, que son grupos ordenados de pares clave-valor.
- los **Set**, que son grupos de valores únicos.

### Map

El objeto **Map** es una colección de parejas de [clave, valor].

En cambio, un objeto en Javascript es un tipo particular de *Map* en el 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, ...).

Más información en [MDN](#) o cualquier otra página.

## Set

El objeto **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:
ganadoresNoDuplicados = Array.from(new Set(ganadores)) // ['Márquez', 'Rossi', 'Lorenzo']
```



## DWEC – Javascript Web Cliente.

JavaScript 02 – Objetos en Javascript .....	1
Introducción.....	1
Propiedades de un objeto.....	1

# JavaScript 03 – Objetos en Javascript

## Introducción

En Javascript podemos definir cualquier variable como un objeto, existen dos formas para hacerlo:

- Declarándola con **new** (NO se recomienda)
- Forma recomendada: creando un *literal object* usando notación **JSON**.

Ejemplo con *new*:

```
let alumno = new Object();
alumno.nombre = 'Carlos';      // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz';    // se crea la propiedad 'apellidos'
alumno.edad = 19;
```

Creando un *literal object* según la forma recomendada, el ejemplo anterior sería:

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  edad: 19,
};
```

## Propiedades de un objeto

Podemos acceder a las propiedades con **.** (punto) o **[ ]**:

```
console.log(alumno.nombre);      // imprime 'Carlos'
console.log(alumno['nombre']);    // imprime 'Carlos'

let prop = 'nombre';
console.log(alumno[prop]);       // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen no se produce un error, se devuelve **undefined**:

```
console.log(alumno.ciclo);      // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo);           // muestra undefined
console.log(alumno.ciclo.descrip);    // se genera un ERROR
```

En versiones anteriores de JavaScript, para evitar este error se comprobaba que existían las propiedades previamente. Veamos un ejemplo:

```
console.log(alumno.ciclo && alumno.ciclo.descrip);
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip y si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador `?.` para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip);
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip y si no muestra undefined
```

Este nuevo operador también puede aplicarse a **arrays**:

```
let alumnos = ['Juan', 'Ana'];
console.log(alumnos?.[0]);
// si alumnos es un array y existe el primer elemento muestra el valor
// si ese elemento no existe muestra undefined
// si no existe el objeto con el nombre alumnos da ERROR
```

Podremos recorrer las propiedades de un objeto con `for..in`:

```
for (let prop in alumno) {
  console.log(prop + ': ' + alumno[prop])
}
```

Resultado:

<code>for (let prop in alumno) {   console.log(prop + ': ' + alumno[prop]) }</code>	nombre: Carlos apellidos: Pérez Ortiz edad: 19
---	--

Una propiedad de un objeto puede ser una función:

```
alumno.getInfo = function() {
  return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad +
' años'
}

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19años
```

También se puede incluir la declaración del método en la declaración del objeto:

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  edad: 19,
```

```

    getInfo: function(){
        return `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`;
    }
};

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años

```

OJO: deberíamos poder ponerlo con sintaxis *arrow function*, pero no funciona.

```

let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: ()=> `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`
};

console.log(alumno.getInfo()); //El alumno undefined undefined tiene undefined años

```

No funciona bien porque `this` tiene distinto valor dependiendo del contexto, y no se puede usar en estos casos con función flecha. Tienes un documento titulado “JavaScript - Anexo - Uso de `this` en contexto” que lo explica.

Si el valor de una propiedad es el valor de una variable que se llama como ella, desde ES2015 no es necesario ponerlo:

```

let nombre = 'Carlos'

let alumno = {
    nombre,           // es equivalente a nombre: nombre
    apellidos: 'Pérez Ortiz',
    ...
}

```

EJERCICIO: Crea un objeto llamado `tvSamsung` con las propiedades `nombre` (TV Samsung 42’’), `categoria` (Televisores), `unidades` (4), `precio` (345.95) y con un método llamado `importe` que devuelve el valor total de las unidades (`nº de unidades * precio`).

Prueba el uso del método con un ejemplo.

**Solución:**

```

let tvSamsung = {
    nombre: 'TV Samsung 42''',
    categoria: 'televisores',
    unidades: 4,
    precio: 345.95,
    importe: function(){
        return this.unidades * this.precio;
    }
};

console.log(` ${tvSamsung.nombre}: ${tvSamsung.importe()} €`);
// TV Samsung 42'': 1383.8 €

```



## DWEC – Javascript Web Cliente.

JavaScript 04 – DOM - Document Object Model .....	1
Introducción.....	1
Acceso a los nodos.....	3
getElementById(id) .....	3
getElementsByClassName(clase).....	4
getElementsByTagName(elemento) .....	4
querySelector(selector).....	4
querySelectorAll(selector).....	4
Atajos .....	5
Acceso a nodos a partir de otros .....	5
Propiedades de un nodo.....	6
innerHTML .....	6
textContent .....	6
value.....	6
Manipular el árbol DOM .....	7
createElement.....	7
createTextNode .....	7
appendChild .....	7
insertBefore .....	8
removeChild.....	8
replaceChild .....	8
cloneNode .....	8
Ejemplo de creación de nuevos nodos: .....	9
Modificar el DOM con ChildNode .....	10
Atributos de los nodos .....	10
Estilos de los nodos .....	10
Atributos de clase .....	11

# JavaScript 04 – DOM - Document Object Model

## Introducción

La mayoría de las veces que se programa con Javascript es para que se ejecute en una página web mostrada por el navegador. En este contexto, y para facilitar el desarrollo de la aplicación, 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 elemento constituye su propio nodo y tiene subnodos con sus *atributos*, *estilos* y otros elementos HTML que contiene.

Nota: Un **elemento HTML** consiste en:

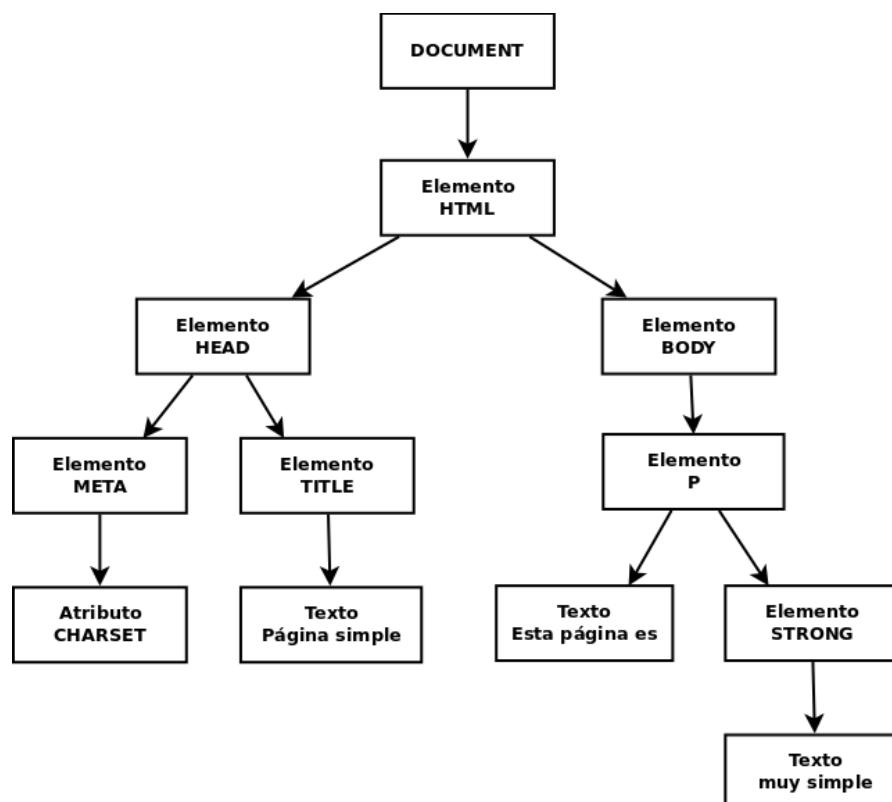
- Una **etiqueta inicial**. Opcionalmente contiene pares “atributo: valor”.
- **Contenido** del elemento (no siempre aparece).
- **Etiqueta final** o de cierre (no siempre aparece).

Es frecuente que alguien se refiera a un ‘elemento HTML’ como ‘etiqueta HTML’. Por lo que debes interpretar el significado de **etiqueta** en cada momento según el contexto.

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 elemento HTML suele originar 2 nodos:

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

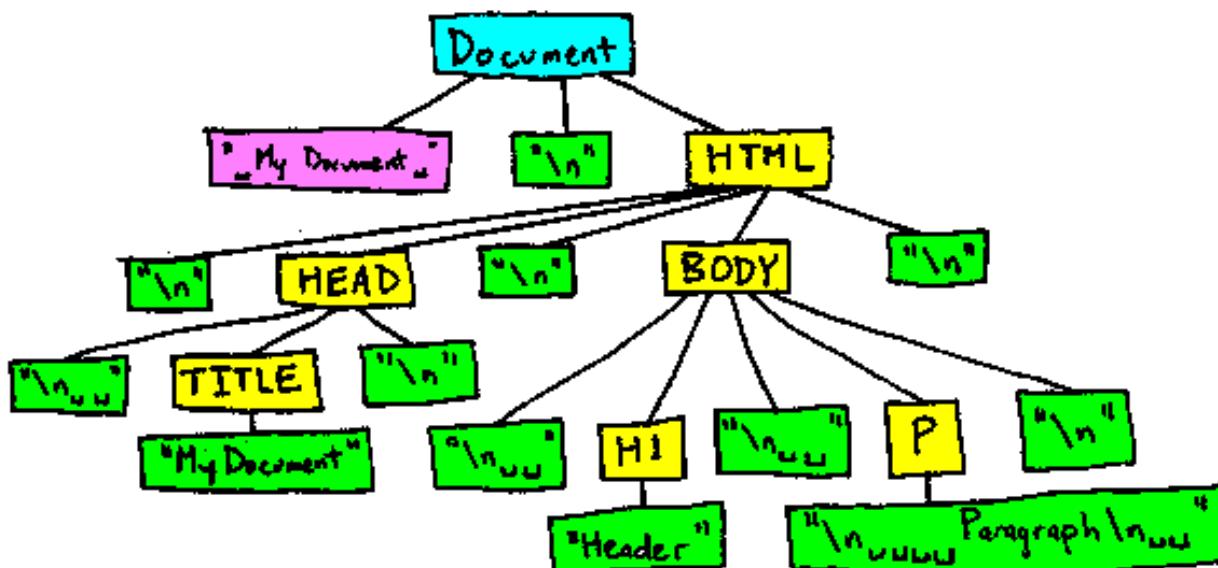
Cada nodo es un objeto con sus propiedades y métodos.

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)**

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

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

Cuidado de no confundir con el método `.getElementsByName(nombre)`, que devuelve los elementos con el atributo *name* especificado.

### getElementsByClassName(clase)

**.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 `[indice]`) pero no se les pueden aplicar los métodos *filter*, *map*, ... a menos que se conviertan a arrays con `Array.from()`

### getElementsByTagName(elemento)

**.getElementsByTagName(elemento):** devuelve una colección con todos los nodos del tipo *elemento HTML* indicado. Ej.:

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

### querySelector(selector)

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

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

### querySelectorAll(selector)

**.querySelectorAll(selector):** devuelve una NodeList 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_
```

```
let nodo = document.querySelectorAll('#text'); // nodo contendrá el elemento con ID=text
```

Más información en: <https://developer.mozilla.org/es/docs/Web/API/Document/querySelectorAll>

**NOTA:** Los objetos `NodeList` son colecciones de nodos como los devueltos por propiedades como `Node.childNodes` y el método `document.querySelectorAll()`.

**NOTA:** al aplicar estos métodos sobre `document` se seleccionarán sobre la página (objeto `document`). Pero podrían también aplicarse a cualquier nodo, en ese caso la búsqueda se realizaría sólo entre los descendientes de dicho nodo.

## Atajos

También tenemos 'atajos' para obtener algunos elementos comunes:

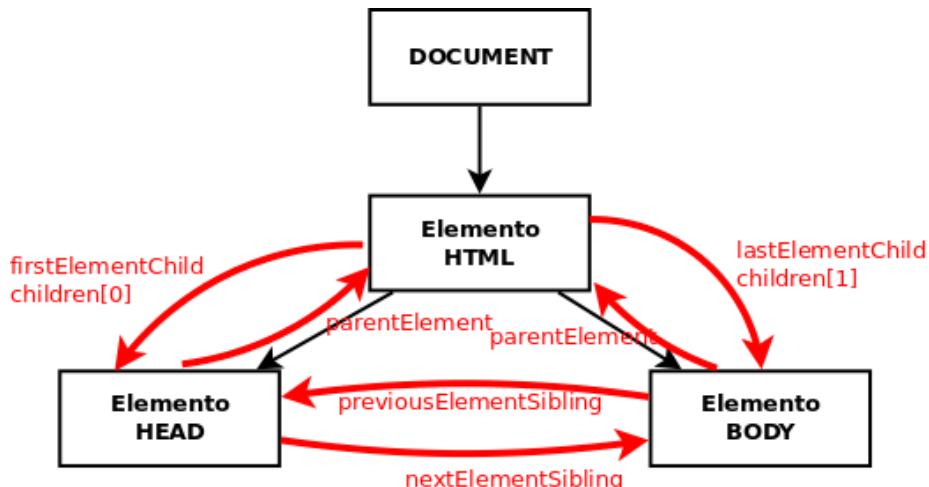
- `document.documentElement`: devuelve el nodo del elemento `<html>`
- `document.head`: devuelve el nodo del elemento `<head>`
- `document.body`: devuelve el nodo del elemento `<body>`
- `document.title`: devuelve el nodo del elemento `<title>`
- `document.link`: devuelve una colección con todos los hiperenlaces 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 ábrelo en tu navegador. Obtén por consola, al menos de 2 formas diferentes lo que se pide:

1. El elemento con id 'input2'
2. La colección de párrafos
3. Lo mismo pero sólo de los párrafos que hay dentro del div 'lipsum'
4. El formulario (ojo, no la colección con el formulario sino sólo el formulario)
5. Todos los inputs
6. Sólo los inputs con nombre 'sexo'
7. 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 nº de nodos hijo de `elemento`

**IMPORTANTE:** a menos que interesen comentarios, saltos de página, etc ..., **siempre** hay que 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:

1. El primer párrafo que hay dentro del div 'lipsum'
2. El segundo párrafo de 'lipsum'
3. El último item de la lista
4. El elemento `label` de 'Escoge sexo'

## Propiedades de un nodo

Las principales propiedades de un nodo son:

### innerHTML

`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>'
```

### textContent

`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'
```

### value

`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`: pone (sitúa) el foco en `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 / borde 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:

1. El `innerHTML` de la etiqueta de 'Escoge sexo'
2. El `textContent` de esa etiqueta
3. El valor del primer input de sexo
4. 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:

### createElement

`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');
```

### createTextNode

`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');
```

### appendChild

`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
```

## insertBefore

`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
```

## removeChild

`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 parent con:
primerElementoDeLista.parentElement.removeChild(primerElementoDeLista);
```

## replaceChild

`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
```

## cloneNode

`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 deberá 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':

```
<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>
```

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

miDiv.innerHTML+="

Párrafo añadido al final

";

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

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

Resultado:

```
Este es el primer párrafo que tiene algo en negrita.
Este es el nuevo segundo párrafo
Este era el segundo párrafo pero será desplazado hacia abajo.
Y este es el último párrafo pero luego añadiremos otro después
Párrafo añadido al final
```

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

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

**OJO:** La forma de añadir el último párrafo (línea #3: `miDiv.innerHTML+="

Párrafo añadido al final

";`) 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](#).

EJERCICIO: Añade a la página:

1. Un nuevo párrafo al final del DIV '*lipsum*' con el texto "Nuevo párrafo **añadido** por javascript" (fíjate que una palabra está en negrita)
2. 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.getAttribute('nombreAtributo')`.
- `elemento.setAttribute('nombreAtributo', 'valor')`: establece *valor* como nuevo valor del atributo *nombreAtributo* de *elemento*. También puede cambiarse el valor directamente con `elemento.setAttribute('nombreAtributo', 'nuevoValor')`.
- `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.getAttribute('atributo')`). Ejemplos:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
miPrimeraLista.id = 'primera-lista';
// es equivalente 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">...</p>`:

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

Además, dispone de los métodos:

`add`

`.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`

`.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">...
```

`toogle`

`.toogle(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`

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

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

### replace

**.replace(oldClase, newClase):** reemplaza del elemento una clase existente por una nueva. Ej.:

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

Ten 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';  
}
```

## JavaScript - El DOM

### Introducción

Para renderizar una página web el navegador crea dos árboles, el primero de ellos **DOM** (*Document Object Model*) y el segundo **CSSOM**, que contiene el árbol de estilos CSS. En ambos casos se dice árbol porque esa es la estructura de datos que representa, donde cada elemento es un nodo.

DOM permite a los programadores web acceder y manipular las páginas XHTML como si fueran documentos XML. De hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.

El DOM se ha convertido en una utilidad disponible para la mayoría de los lenguajes de programación (Java, PHP, JavaScript).

### Árbol de nodos

Una de las tareas habituales en la programación de aplicaciones web con JavaScript consiste en la manipulación de las páginas web. De esta forma, es habitual obtener el valor almacenado por algunos elementos (por ejemplo: los elementos de un formulario), crear un elemento (párrafos, `<div>`, etc.) de forma dinámica y añadirlo a la página, aplicar una animación a un elemento (que aparezca/desaparezca, que se desplace, etc.).

Todas estas tareas habituales son muy sencillas de realizar gracias a DOM. Sin embargo, para poder utilizar las utilidades de DOM, es necesario "transformar" la página original. Una página HTML normal no es más que una sucesión de caracteres, por lo que es un formato muy difícil de manipular. Por ello, los navegadores web transforman automáticamente todas las páginas web en una estructura más eficiente de manipular.

Esta transformación la realizan todos los navegadores de forma automática y nos permite utilizar las herramientas de DOM de forma muy sencilla. El motivo por el que se muestra el funcionamiento de esta transformación interna es que condiciona el comportamiento de DOM y por tanto, la forma en la que se manipulan las páginas.

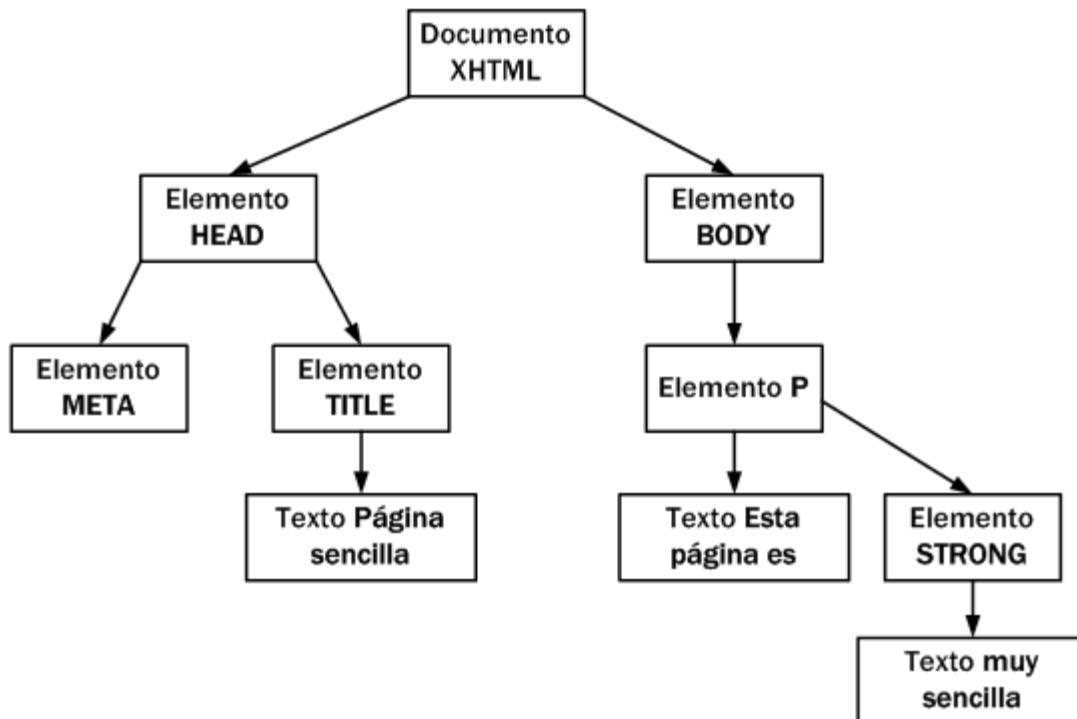
DOM transforma todos los documentos XHTML en un conjunto de elementos llamados **nodos**, que están interconectados y que representan los contenidos de las páginas web y las relaciones entre ellos. Por su aspecto, la unión de todos los nodos se llama "**árbol de nodos**".

La siguiente página XHTML sencilla:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>Página sencilla</title>
  </head>
```

```
<body>
  <p>Esta página es <strong>muy sencilla</strong></p>
</body>
</html>
```

Se transforma en el siguiente árbol de nodos:



En el esquema anterior, cada rectángulo representa un nodo DOM y las flechas indican las relaciones entre nodos.

Dentro de cada nodo, se ha incluido su tipo (que se verá más adelante) y su contenido.

La raíz del árbol de nodos de cualquier página XHTML siempre es la misma: un nodo de tipo especial denominado "*Documento*".

A partir de ese nodo raíz, cada etiqueta XHTML se transforma en un nodo de tipo "*Elemento*". La conversión de etiquetas en nodos se realiza de forma jerárquica. De esta forma, del nodo raíz solamente pueden derivar los nodos HEAD y BODY. A partir de esta derivación inicial, cada etiqueta XHTML se transforma en un nodo que deriva del nodo correspondiente a su "*etiqueta padre*".

La transformación de las etiquetas XHTML habituales genera dos nodos: el primero es el nodo de tipo "*Elemento*" (correspondiente a la propia etiqueta XHTML) y el segundo es un nodo de tipo "*Texto*" que contiene el texto encerrado por esa etiqueta XHTML.

Así, la siguiente etiqueta XHTML:

```
<title>Página sencilla</title>
```

Genera los siguientes dos nodos:

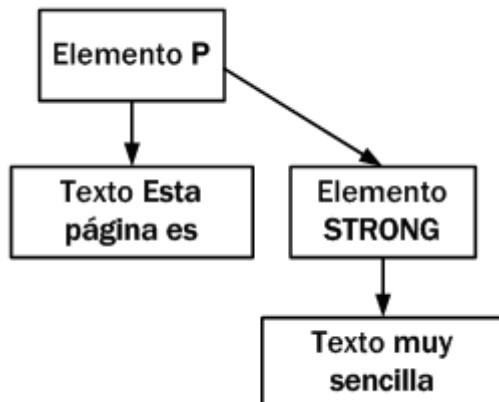


De la misma forma, la siguiente etiqueta XHTML:

`<p>Esta página es <strong>muy sencilla</strong></p>`

Genera los siguientes nodos:

- Nodo de tipo "Elemento" correspondiente a la etiqueta `<p>`.
- Nodo de tipo "Texto" con el contenido textual de la etiqueta `<p>`.
- Como el contenido de `<p>` incluye en su interior otra etiqueta XHTML, la etiqueta interior se transforma en un nodo de tipo "Elemento" que representa la etiqueta `<strong>` y que deriva del nodo anterior.
- El contenido de la etiqueta `<strong>` genera a su vez otro nodo de tipo "Texto" que deriva del nodo generado por `<strong>`.



La transformación automática de la página en un árbol de nodos siempre sigue las mismas reglas:

- Las etiquetas XHTML se transforman en dos nodos: el primero es la propia etiqueta y el segundo nodo es hijo del primero y consiste en el contenido textual de la etiqueta.
- Si una etiqueta XHTML se encuentra dentro de otra, se sigue el mismo procedimiento anterior, pero los nodos generados serán nodos hijo de su etiqueta padre.

Como se puede suponer, las páginas XHTML habituales producen árboles con miles de nodos. Aun así, el proceso de transformación es rápido y automático, siendo las funciones proporcionadas por DOM (que se verán más adelante) las únicas que permiten acceder a cualquier nodo de la página de forma sencilla e inmediata.

## Tipos de nodos

La especificación completa de DOM define 12 tipos de nodos, aunque las páginas XHTML habituales se pueden manipular manejando solamente cuatro o cinco tipos de nodos:

- Document, nodo raíz del que derivan todos los demás nodos del árbol.

- **Element**, representa cada una de las etiquetas XHTML. Se trata del único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- **Attr**, se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas XHTML, es decir, uno por cada par atributo=valor.
- **Text**, nodo que contiene el texto encerrado por una etiqueta XHTML.
- **Comment**, representa los comentarios incluidos en la página XHTML.

Los otros tipos de nodos existentes que no se van a considerar son:

DocumentType, CDataSection, DocumentFragment, Entity, EntityReference, ProcessingInstruction y Notation

## Accediendo a los Nodos

Una vez construido automáticamente el árbol completo de nodos DOM, ya es posible utilizar las funciones DOM para acceder de forma directa a cualquier nodo del árbol. Como acceder a un nodo del árbol es equivalente a acceder a "un trozo" de la página, una vez construido el árbol, ya es posible manipular de forma sencilla la página: acceder al valor de un elemento, establecer el valor de un elemento, mover un elemento de la página, crear y añadir nuevos elementos, etc.

El acceso a los nodos, su modificación y su eliminación solamente es posible cuando el árbol DOM ha sido construido completamente, es decir, después de que la página XHTML se cargue por completo.

El DOM proporciona dos métodos alternativos para acceder a un nodo específico:

- Acceso directo.
- Acceso a través de sus nodos padre.

Las funciones que proporciona DOM para acceder a un nodo a través de sus nodos padre consisten en acceder al nodo raíz de la página y después a sus nodos hijos y a los nodos hijos de esos hijos y así sucesivamente hasta el último nodo de la rama terminada por el nodo buscado. Sin embargo, cuando se quiere acceder a un nodo específico, es mucho más rápido acceder directamente a ese nodo y no llegar hasta él descendiendo a través de todos sus nodos padre.

## Acceso directo a los nodos

### dDocument.getElementById()

Como sucede con todas las funciones que proporciona DOM, la función `getElementsById()` tiene un nombre muy largo, pero que lo hace auto explicativo.

La función `getElementById()` es la más utilizada cuando se desarrollan aplicaciones web dinámicas. Se trata de la función preferida para acceder directamente a un nodo y poder leer o modificar sus propiedades.

La función `getElementById()` devuelve el elemento XHTML cuyo atributo `id` coincide con el parámetro indicado en la función. Como el atributo `id` debe ser único para cada elemento de una misma página, la función devuelve únicamente el nodo deseado.

```
let cabecera = document.getElementById("cabecera");
```

```
<div id="cabecera">
```

```
<a href="/" id="logo">...</a>
</div>
```

### document.getElementsByTagName()

La función `getElementsByTagName(nombreEtiqueta)` obtiene todos los elementos de la página XHTML cuya etiqueta sea igual que el parámetro que se le pasa a la función.

El siguiente ejemplo muestra cómo obtener todos los párrafos de una página XHTML:

```
const parrafos = document.getElementsByTagName("p");
```

El valor que se indica delante del nombre de la función (en este caso, `document`) es el nodo a partir del cual se realiza la búsqueda de los elementos. En este caso, como se quieren obtener todos los párrafos de la página, se utiliza el valor `document` como punto de partida de la búsqueda.

El valor que devuelve la función es un `NodeList` con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. El valor devuelto es una lista o colección de nodos DOM, no un array de cadenas de texto o un array de objetos normales. Por lo tanto, se debe procesar cada valor de la lista de forma correcta.

### Diferencias entre NodeList y Array

Un `NodeList` se puede parecer mucho a un `Array`, pero la realidad es que son dos estructuras completamente distintas.

Por un lado, `NodeList` es una **colección** de nodos del DOM extraídos del HTML.

Y un `Array` es un **tipo** de dato especial en JavaScript, donde podemos almacenar cualquier tipo de dato.

Ambos tienen similitudes, como acceder a la longitud a través de `length`, acceder a los elementos a través de su índice usando `[índice]`.

En un `NodeList` no tenemos disponibles los principales métodos de `Array` que nos facilitan la tarea, como `map()`, `filter()`, `reduce()`, `some()`.

Un dato curioso e interesante del `NodeList` es que es una especie de colección en vivo, lo que quiere decir es que si se agrega o se elimina algún elemento del DOM, los cambios son aplicados inmediatamente y de forma automática al `NodeList`.

Es recomendable transformar los `NodeList` a `Array`, pues la mayoría de los motores de JavaScript están optimizados para trabajar con `Arrays`. Veamos dos formas de transformar un `NodeList` en un `array`.

```
// Forma 1: Spread Operator
const parrafos = document.getElementsByTagName("p");
const parrafosArray = [...parrafos];
```

```
// Usando la clase Array y su método from
const parrafos = document.getElementsByTagName("p");
const parrafosArray = Array.from(parrafos);
```

Y a partir de ahí, utilizar el array en lugar de la lista.

El operador de extensión (Spread Operator), que sirve para insertar una lista dentro de otra, en este caso se usa para convertir una lista en Array.

```
let lista=[1,2,3,4];
let lista2=[...lista,5];
console.log(lista2); [ 1, 2, 3, 4, 5 ]

let otraLista=[1,2,3,4];
let otraLista2=[5,...lista];
console.log(otraLista2); [ 5, 1, 2, 3, 4 ]
```

Siguiendo con el ejemplo que mostraba cómo obtener todos los párrafos de una página XHTML, y sin convertir de NodeList a Array, veamos cómo se puede tratar la lista directamente:

```
const parrafos = document.getElementsByTagName("p");
```

De este modo, se puede obtener el primer párrafo de la página de la siguiente manera:

```
let primerParrafo = parrafos[0];
```

De la misma forma, se podrían recorrer todos los párrafos de la página con el siguiente código:

```
for(let i=0; i<parrafos.length; i++) {
    let parrafo = parrafos[i];
}
```

La función `getElementsByTagName()` se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función. En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
let parrafos = document.getElementsByTagName("p");
let primerParrafo = parrafos[0];
let enlaces = primerParrafo.getElementsByTagName("a");
```

## Document.getElementsByName()

La función `getElementsByName()` es similar a la anterior, pero en este caso se buscan los elementos cuyo atributo name sea igual al parámetro proporcionado. En el siguiente ejemplo, se obtiene directamente el único párrafo con el nombre indicado:

```
let parrafoEspecial = document.getElementsByName("especial");

<p name="prueba">...</p>
<p name="especial">...</p>
<p>...</p>
```

Normalmente el atributo name es único para los elementos HTML que lo definen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML *radiobutton*, el atributo name es común a todos los *radiobutton* que están relacionados, por lo que la función devuelve una colección de elementos.

## Document.getElementsByClassName()

Nos retorna una lista, pero en este caso nos devuelve los nodos que entre sus clases contenga la clase especificada en el argumento del método, es decir, vamos a identificar una clase y traemos todos los elementos que contengan dicha clase, supongamos que queremos obtener todos los nodos que contengan la clase contenedor, lo haríamos de la siguiente manera:

```
const listaContenedor = document.getElementsByClassName("contenedor");
```

## Document.querySelector()

Nos permite obtener cualquier elemento del DOM de acuerdo con el argumento que le indiquemos, podemos pasarle una cadena de caracteres que contiene uno o más selectores CSS, éstos deben ir separado por comas. Este método retorna el primer elemento que coincida con el filtro, es decir, si existen varios elementos que coincidan con la búsqueda, éste nos retornará el primero que encuentre. Si no encuentra ningún elemento, retorna null. Veamos un ejemplo donde buscaremos un elemento que contenga la clase container.

```
let elemento = document.querySelector(".container");
```

Como vemos, debemos especificar el símbolo del selector, similar como ocurre en CSS, si queremos obtener por clase (" .container "), ID (#container), etiqueta directamente (" h1 "), elemento que cumpla con el selector (" p .azul ").

## querySelectorAll()

Este método es casi igual al anterior a diferencia de que este devuelve una lista (**NodeList**) donde se encuentran todos los elementos que coincidan con el o los selectores indicados. Supongamos que queremos hacer la misma búsqueda del ejemplo del ítem anterior, obtener los elementos con la clase container:

```
const listaElementos = document.querySelectorAll(".container");
```

A diferencia del método anterior que nos devuelve la primera coincidencia, aquí vamos a obtener todos esos elementos que coincidan con el selector indicado.

# Creación de nodos

Acceder a los nodos y a sus propiedades (que se verá más adelante) es sólo una parte de las manipulaciones habituales en las páginas. Las otras operaciones habituales son las de crear y eliminar nodos del árbol DOM, es decir, crear y eliminar "trozos" de la página web.

## Nodo appendChild(): Creación de elementos XHTML simples

Como se ha visto, un elemento XHTML sencillo, como por ejemplo un párrafo, genera dos nodos: el primer nodo es de tipo Element y representa la etiqueta <p> y el segundo nodo es de tipo Text y representa el contenido textual de la etiqueta <p>.

Por este motivo, crear y añadir a la página un nuevo elemento XHTML sencillo consta de cuatro pasos diferentes:

1. Creación de un nodo de tipo **Element** que represente al elemento.
2. Creación de un nodo de tipo **Text** que represente el contenido del elemento.
3. Añadir el nodo **Text** como nodo hijo del nodo **Element**.
4. Añadir el nodo **Element** a la página, en forma de nodo hijo del nodo correspondiente al lugar en el que se quiere insertar el elemento.

De este modo, si se quiere añadir un párrafo simple al final de una página XHTML, es necesario incluir el siguiente código JavaScript:

```
// Crear nodo de tipo Element
let parrafo = document.createElement("p");

// Crear nodo de tipo Text
let contenido = document.createTextNode("Hola Mundo!");

// Añadir el nodo Text como hijo del nodo Element
parrafo.appendChild(contenido);

// Añadir el nodo Element como hijo de la pagina
document.body.appendChild(parrafo);
```

El proceso de creación de nuevos nodos puede llegar a ser tedioso, ya que implica la utilización de tres funciones DOM:

- `createElement(etiqueta)`: crea un nodo de tipo Element que representa al elemento XHTML cuya etiqueta se pasa como parámetro.
- `createTextNode(contenido)`: crea un nodo de tipo Text que almacena el contenido textual de los elementos XHTML.
- `nodoPadre.appendChild(nodoHijo)`: añade un nodo como hijo de otro nodo. Se debe utilizar al menos dos veces con los nodos habituales: en primer lugar, se añade el nodo Text como hijo del nodo Element y a continuación se añade el nodo Element como hijo de algún nodo de la página.

### **Element.append()**

Este método es una evolución de `appendChild`, donde podemos agregar más de un nodo, cosa que no podemos hacer con `appendChild`.

Podemos agregar texto, también agrega los nodos al final de los hijos del elemento padre especificado, y no retorna ningún elemento.

Veamos un ejemplo, donde insertaremos un texto en un elemento li, y además agregaremos un nuevo hijo, simulando un menú de varios niveles:

```
const container = document.querySelector("ul.container");
const item = document.createElement("li");
const subItem = document.createElement("ul");

// Agregamos el texto del item, e insertamos la nueva lista
item.append("Categorias", subItem);

// Agregamos el item al menu
container.append(item);
```

## nodoPadre.insertBefore()

Este método permite insertar o agregar un nodo, antes del nodo que pasemos de referencia, y que pertenece al padre del que estamos accediendo a dicho método. Si insertamos un Nodo que ya está en nuestro DOM el método lo cambia de lugar, es decir, lo quita de donde se encuentra actualmente y lo inserta en el sitio al que estamos haciendo referencia. Este método recibe dos argumentos, el primero de ellos es el Nodo que queremos insertar, y el segundo es el Nodo de referencia. Si el segundo argumento es null entonces el nodo se inserta al final de la lista de hijos del padre, es decir, actúa como el `appendChild`. Veamos un ejemplo:

```
const parent = document.querySelector(".container");
const item = document.querySelector(".contact");
const newItem = document.createElement("li");
const text = document.createTextNode("Blog");

newItem.append(text);

/* Insertamos el nuevo elemento al parent, lo insertaremos antes del
elemento con clase contact */
parent.insertBefore(newItem, item);
```

## nodoPadre.insertAdjacentElement()

Este método permite insertar un nodo de acuerdo con una posición especificada, y lo ubica con respecto al elemento sobre el cual es llamado. Es uno de los más complicados de usar, pero el que más opciones de inserción nos proporciona. Este método recibe 2 argumentos, el primero de ellos donde se insertará el nuevo nodo, para esto contamos con cuatro opciones, las cuales son:

- **beforebegin**: Inserta el elemento antes de la referencia. Es decir, antes del nodo que llama al método en cuestión.
- **afterbegin**: Insertar el nodo dentro del elemento referencia, pero además de esto lo inserta antes de su primer hijo.
- **beforeend**: Al igual que la opción anterior lo agrega dentro del elemento referencia, con la particularidad de que ahora lo inserta después de su ultimo hijo.
- **afterend**: Inserta el elemento después del nodo referencia, es decir, después del nodo que llama al método.

# Otras formas de agregar HTML

Existen otras formas de agregar el HTML, estas también funcionan para leer fragmentos de nuestro código, son las siguientes propiedades `innerHTML` y `outerHTML`, veamos qué hace cada una de ellas

## Element.innerHTML()

Esta propiedad devuelve o establece la sintaxis XHTML descrita dentro del elemento al que hace referencia, es decir, esta propiedad devuelve todo el contenido que está dentro del Nodo (element) que la llama. Al acceder a

esta propiedad obtenemos un fragmento de código (DOMString) que contiene todo el extracto serializado de todos los descendientes. Esta propiedad **también es usada para reemplazar todo el XHTML descendiente del elemento que la invoca**. Supongamos que tenemos un div que contiene un párrafo p dentro, ahora, queremos cambiar el contenido de ese div, esto lo hacemos de la siguiente manera:

```
// Elemento base
<div class="container">
    <p>Bienvenido a este sitio</p>
</div>

// Obtenemos el elemento al que queremos cambiar el contenido
let container = document.querySelector(".container");

// Reemplazamos el contenido
container.innerHTML = "<span>Bienvenidos<span>";

// Resultado
<div class="container">
    <span>Bienvenidos<span>
</div>
```

Esta propiedad no es recomendable porque puedes sufrir ataques de “cross-site scripting”, ya que puedes ejecutar código JavaScript maligno, por la forma en que puedes ejecutar JavaScript sin necesidad de tener la etiqueta. Más información en <https://www.a2secure.com/blog/los-peligros-de-los-ataques-de-cross-site-scripting-xss/>

## `Element.outerHTML()`

Es similar a la propiedad anterior, pero la única diferencia es que ésta **toma el elemento padre (element) y reemplaza el contenido completo, incluyendo el elemento contenedor**, veamos como quedaría el ejemplo anterior si usamos `outerHTML`:

```
// Elemento base
<div class="container">
    <p>Bienvenido a este sitio</p>
</div>

// Obtenemos el elemento al que queremos cambiar el contenido
let container = document.querySelector(".container");

// Reemplazamos el contenido
container.outerHTML = "<span>Bienvenidos<span>";

// Resultado
```

```
<span>Bienvenidos</span>
```

## Acceso directo a los atributos XHTML

Una vez que se ha accedido a un nodo, el siguiente paso natural consiste en acceder y/o modificar sus atributos y propiedades. Mediante DOM, es posible acceder de forma sencilla a todos los atributos XHTML y todas las propiedades CSS de cualquier elemento de la página.

Los atributos XHTML de los elementos de la página se transforman automáticamente en propiedades de los nodos. Para acceder a su valor, simplemente se indica el nombre del atributo XHTML detrás del nombre del nodo.

El siguiente ejemplo obtiene de forma directa la dirección a la que enlaza el enlace:

```
let enlace = document.getElementById("enlace");
console.log(enlace.href); // muestra http://www...com

<a id="enlace" href="http://www...com">Enlace</a>
```

En el ejemplo anterior, se obtiene el nodo DOM que representa el enlace mediante la función `document.getElementById()`. A continuación, se obtiene el atributo `href` del enlace mediante `enlace.href`. Para obtener por ejemplo el atributo `id`, se utilizaría `enlace.id`.

Las propiedades CSS no son tan fáciles de obtener como los atributos XHTML. Para obtener el valor de cualquier propiedad CSS del nodo, se debe utilizar el atributo `style`. El siguiente ejemplo obtiene el valor de la propiedad `margin` de la imagen:

```
let imagen = document.getElementById("imagen");
console.log(imagen.style.margin);


```

**Advertencia:** Aunque el funcionamiento es homogéneo entre distintos navegadores, los resultados pueden no ser exactamente iguales.

Si el nombre de una propiedad CSS es compuesto, se accede a su valor modificando ligeramente su nombre:

```
var parrafo = document.getElementById("parrafo");
alert(parrafo.style.fontWeight); // muestra "bold"

<p id="parrafo" style="font-weight: bold;">...</p>
```

La transformación del nombre de las propiedades CSS compuestas consiste en eliminar todos los guiones medios (-) y escribir en mayúscula la letra siguiente a cada guión medio. A continuación se muestran algunos ejemplos:

- `font-weight` se transforma en `fontWeight`
- `line-height` se transforma en `lineHeight`
- `border-top-style` se transforma en `borderTopStyle`
- `list-style-image` se transforma en `listStyleImage`

El único atributo XHTML que no tiene el mismo nombre en XHTML y en las propiedades DOM es el atributo `class`.

Como la palabra `class` está reservada por JavaScript, no es posible utilizarla para acceder al atributo `class` del elemento XHTML. En su lugar, DOM utiliza el nombre `className` para acceder al atributo `class` de XHTML:

```
let parrafo = document.getElementById("parrafo");
console.log(parrafo.class); // muestra "undefined"
console.log(parrafo.className); // muestra "normal"

<p id="parrafo" class="normal">...</p>
```

## Atributos: crear, modificar y eliminar

Este método nos permite crear un nodo de tipo atributo. El DOM no impone que tipo de atributos pueden ser agregados a un particular elemento, solo crea un nodo, que puede ser (o no) en el DOM, veamos un ejemplo:

```
var nodo = document.getElementById("div1");
var a = document.createAttribute("miAtributo");
a.value = "nuevoValor";
nodo.setAttributeNode(a);
console.log(nodo.getAttribute("miAtributo")); // "nuevoValor"
```

Veamos otro ejemplo de cómo podemos hacer esto: supongamos que tenemos un input que recibe el nombre completo de una persona

```
// Nuestro input
<input class="form-control"
       id="persona-nombre"
       placeholder="Nombre completo" />

// Obtenemos el input
const input = document.querySelector("#persona-nombre");

// Obtenemos los atributos
input.className // salida: "form-control"
input.id // salida: "persona-nombre"
input.placeholder // salida: "Nombre completo"
```

Podemos acceder a cada atributo como lo hacemos cuando queremos obtener una propiedad en un objeto de JavaScript, con esto podemos verificar qué valor está tomando cada atributo. Si en el input escribimos texto desde el navegador, podemos acceder a ese valor a través de `input.value`.

JavaScript cuenta además con otras métodos que permiten manipular los atributos de los nodos, estos son `element.setAttribute()`, `element.getAttribute()` y `element.removeAttribute()`. Veamos cómo funciona cada uno:

### Element.setAttribute()

Este método establece el valor de un atributo en el elemento indicado, recibe dos argumentos, el primero de ellos un string donde se indica el nombre del atributo, y el segundo el valor que tomará dicho atributo. Es importante mencionar que, si el atributo ya existe en el elemento, el valor será actualizado. Supongamos que al input del ejemplo anterior queremos agregarle el atributo `name`, lo hacemos de la siguiente manera:

```
input.setAttribute("name", "fullName");

// Ahora el input quedará de la siguiente manera

<input class="form-control"
      id="persona-nombre"
      placeholder="Nombre completo"
      name="fullName" />
```

### Element.getAttribute()

Este método retorna el valor del atributo especificado en el elemento. Si el atributo al que se hace referencia no está definido en el elemento, el valor que retornará dicho método es `null`. Supongamos que queremos obtener el valor del atributo `name` agregado en el ejemplo anterior.

```
input.getAttribute("name"); // Salida: "fullName"
```

### Element.removeAttribute()

Este método elimina el atributo del elemento que lo invoca, con esto podemos quitar atributos que ya no necesitemos, supongamos que queremos eliminar el atributo `name` del input de los ejemplos anteriores, lo hacemos de la siguiente forma

```
input.removeAttribute("name");

// input final
<input class="form-control"
      id="persona-nombre"
      placeholder="Nombre completo" />
```

## Remover (eliminar) Nodos

### Node.removeChild()

Afortunadamente, eliminar un nodo del árbol DOM de la página es mucho más sencillo que añadirlo. En este caso, solamente es necesario utilizar la función `removeChild()`:

```
var parrafo = document.getElementById("provisional");
parrafo.parentNode.removeChild(parrafo);

<p id="provisional">...</p>
```

La función `removeChild()` requiere como parámetro el nodo que se va a eliminar. Además, esta función debe ser invocada desde el elemento padre de ese nodo que se quiere eliminar. Retorna el nodo eliminado. La forma más segura y rápida de acceder al nodo padre de un elemento es mediante la propiedad `nodoHijo.parentNode`.

Así, para eliminar un nodo de una página XHTML se invoca a la función `removeChild()` desde el valor `parentNode` del nodo que se quiere eliminar. Cuando se elimina un nodo, también se eliminan automáticamente todos los nodos hijos que tenga, por lo que no es necesario borrar manualmente cada nodo hijo.

## **Element.remove()**

Es una mejora del método anterior, este nos permite eliminar un nodo del DOM con solo invocarlos desde el elemento que queremos eliminar, es decir, para este método no hace falta conocer el parent del elemento, solo lo invitamos y elimina el nodo. Veamos cómo funciona con el ejemplo anterior:

```
// Nuestro HTML
<div class="parent">
  <p class="p-child">Titulo</p>
  <span class="span-child">Subtitulo</span>
</div>

// Obtenemos el nodo a eliminar
const nodoAEliminar = document.querySelector(".p-child");

// Eliminamos el nodo
nodoAEliminar.remove();
```

## **Node.replaceChild()**

Este método reemplaza un nodo hijo del elemento especificado por otro. Recibe dos argumentos, el primero de ellos es el nuevo elemento, y como segundo argumento el nodo que será reemplazado por el nuevo. Veamos cómo funciona usando el ejemplo anterior: supongamos que queremos reemplazar el span por un párrafo.

```
// Nuestro HTML
<div class="parent">
  <p class="p-child">Titulo</p>
  <span class="span-child">Subtitulo</span>
</div>

// Obtenemos el nodo padre
```

```
const parent = document.querySelector(".parent");

// Obtenemos el nodo a reemplazar
const nodoAReemplazar = document.querySelector(".p-child");

// El nuevo elemento
const nodoNuevo = document.createElement("p");
nodoNuevo.textContent = "Soy un nuevo nodo";

// Reemplazamos el nodo
parent.replaceChild(nodoNuevo, nodoAReemplazar);
```

Estudia las propiedades y métodos de los **Nodos** en  
<https://developer.mozilla.org/en-US/docs/Web/API/Node>

# Índice:

JavaScript – El DOM .....	1
Introducción.....	1
Árbol de nodos.....	1
Tipos de nodos.....	3
Accediendo a los Nodos.....	4
Acceso directo a los nodos .....	4
getElementById() .....	4
getElementsByName() .....	5
Diferencias entre NodeList y Array .....	5
getElementsByName() .....	6
getElementsByClassName().....	7
querySelector() .....	7
querySelectorAll() .....	7
Creación de nodos .....	7
appendChild(): Creación de elementos XHTML simples.....	7
nodoPadre.append().....	8
nodoPadre.insertBefore() .....	9
nodoPadre.insertAdjacentElement() .....	9
Otras formas de agregar HTML.....	9
element.innerHTML:.....	9
element.outerHTML: .....	10
Acceso directo a los atributos XHTML .....	11
Atributos: crear, modificar y eliminar.....	12
createAttribute().....	12
element.setAttribute().....	13
element.getAttribute() .....	13
element.removeAttribute() .....	13
Remover (eliminar) Nodos.....	13
Node.removeChild().....	13
Element.remove() .....	14
Node.replaceChild() .....	14
Índice: .....	16



## DWEC - Javascript Web Cliente.

JavaScript 05 – Browser Object Model (BOM).....	1
Introducción.....	1
Timers .....	1
Objetos del BOM.....	3
Objeto window.....	3
Diálogos.....	4
Objeto location .....	4
Objeto history .....	5
Otros objetos .....	5

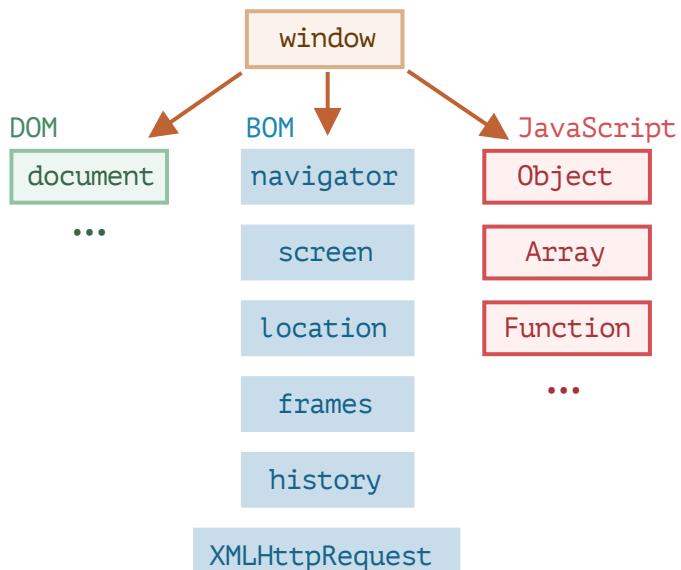
# JavaScript 05 – Browser Object Model (BOM)

## Introducción

Hemos visto cómo interactuar con la página (DOM), ahora veremos cómo acceder a objetos que nos permitan interactuar con el navegador utilizando el BOM (Browser Object Model).

Usando los objetos BOM podemos:

- Abrir, cambiar y cerrar ventanas.
- Ejecutar código transcurrido cierto tiempo (*timers*).
- Obtener información del navegador.
- Ver y modificar propiedades de la pantalla.
- Gestionar cookies, ...



## Timers

Permiten ejecutar código en el futuro (cuando transcurran los milisegundos indicados). Hay 2 tipos:

- `setTimeout(función, milisegundos)`: ejecuta la función especificada **una sola vez**, cuando transcurran los milisegundos indicados.
- `setInterval(función, milisegundos)`: ejecuta la función especificada **cada vez que transcurran** los milisegundos indicados, hasta que sea cancelado el *timer*.

Se pueden pasar más parámetros a las funciones tras los milisegundos y serán los parámetros que recibirá la función a ejecutar.

Ambas funciones devuelven un identificador que nos permitirá cancelar la ejecución del código, con:

- `clearTimeout(identificador)`
- `clearInterval(identificador)`

Ejemplo:

```
const idTimeout = setTimeout(() => console.log('Timeout que se ejecuta al cabo de 1 seg.'), 1000);

let i = 1;
const idInterval = setInterval(() => {
    console.log('Interval cada 3 seg. Ejecución nº: ' + i++);
    if (i === 5) {
        clearInterval(idInterval);
        console.log('Fin de la ejecución del Interval');
    }
}, 3000);
```

EJERCICIO: Prueba a ejecutar cada una de esas funciones.

En lugar de definir la función a ejecutar podemos llamar a una función que ya exista:

```
function showMessage() {
    console.log('Timeout que se ejecuta al cabo de 1 seg.')
}

const idTimeout=setTimeout(showMessage, 1000);
```

Pero en ese caso hay que poner sólo el nombre de la función, sin (), ya que si los ponemos se ejecutaría la función en ese momento y no transcurrido el tiempo indicado.

Si necesitamos pasar algún parámetro a la función, los añadiremos como parámetros de `setTimeout` o `setInterval` después del intervalo.

Ejemplo:

```
function showMessage(msg) {
    alert(msg)
}

const idTimeout = setTimeout(showMessage, 1000, 'Timeout que se ejecuta al cabo de 1 seg.');
```

Otro ejemplo:

```
function myCallback(a, b) {
    // Tu código debe ir aquí
    // Los parámetros son totalmente opcionales
    console.log(a);
    console.log(b);
}

const intervalID = setInterval(myCallback, 500, 'parámetro 1', 'parámetro 2');
```

## Objetos del BOM

Al contrario que para el DOM, no existe un estándar de BOM pero sus objetos (window, location, history, etc) son bastante parecidos en los diferentes navegadores.

### Objeto window

Representa la ventana del navegador y es el objeto principal. De hecho puede omitirse al llamar a sus propiedades y métodos, por ejemplo, el método setTimeout() es en realidad window.setTimeout().

Sus principales propiedades y métodos son:

- `.name`: nombre de la ventana actual
- `.statusbar`: valor de la barra de estado
- `.screenX/.screenY`: distancia de la ventana a la esquina izquierda/superior de la pantalla
- `.outerWidth/.outerHeight`: ancho/alto total de la ventana, incluyendo la toolbar y la scrollbar
- `.innerWidth/.innerHeight`: ancho/alto útil del documento, sin la toolbar y la scrollbar
- `.open(url, nombre, opciones)`: abre una nueva ventana. Devuelve el nuevo objeto ventana.

Las principales **opciones** (lista de ítems separados por comas sin espacios) son:

- `.toolbar`: si tendrá barra de herramientas
- `.location`: si tendrá barra de dirección
- `.directories`: si tendrá botones Adelante/Atrás
- `.status`: si tendrá barra de estado
- `.menubar`: si tendrá barra de menú
- `.scrollbar`: si tendrá barras de desplazamiento
- `.resizable`: si se puede cambiar su tamaño
- `.width=px/.height=px`: ancho/alto
- `.left=px/.top=px`: posición izq/sup de la ventana
- `.opener`: referencia a la ventana desde la que se abrió esta ventana (para ventanas abiertas con `open`)
- `.close()`: la cierra (pide confirmación, a menos que la hayamos abierto con `open`)
- `.moveTo(x,y)`: la mueve a las coord indicadas
- `.moveBy(x,y)`: la desplaza los px indicados
- `.resizeTo(x,y)`: la da el ancho y alto indicados
- `.resizeBy(x,y)`: le añade ese ancho/alto
- `.pageXOffset / pageYOffset`: scroll actual de la ventana horizontal / vertical
- **Otros métodos:** `.back()`, `.forward()`, `.home()`, `.stop()`, `.focus()`, `.blur()`, `.find()`, `.print()`, ...

Más información en [https://www.w3schools.com/jsref/obj\\_window.asp](https://www.w3schools.com/jsref/obj_window.asp)

**NOTA:** por seguridad no se puede mover una ventana fuera de la pantalla, ni darle un tamaño menor de 100x100 px. Tampoco se puede mover una ventana no abierta con `.open()`, o si tiene varias pestañas.

### EJEMPLO:

- a) Abrir una nueva ventana de dimensiones 500x200px en la posición (200,100)
- b) Escribir en la nueva ventana (con `document.write`) un título H1 que diga ‘Hola javascritos del Claudio’.
- c) Al hacer clic sobre un botón de la ventana inicial que la ventana se desplace 40 px a la izquierda y 50 hacia abajo
- d) Al hacer clic sobre otro botón de la ventana incicial, que se cierre la nueva ventana.

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

```

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action="">
    <input type="button" value="Deslazar" onclick="desplazar()">
    <input type="button" value="Cerrar" onclick="cerrar()">
  </form>
  <script src="js/bom1.js"></script>
</body>
</html>

```

```

function desplazar(){
  nuevaVentana.moveBy(40,50);
}

const cerrar = ()=> {
  nuevaVentana.close();
}

const nuevaVentana=window.open("", "", "width= 500, height=200, left=200, top=100");
nuevaVentana.document.write("<h1>Hola javascritos del Claudio</h1>");

```

EJERCICIO: Haz que a los 2 segundos de abrir la página se abra un *popup* con un mensaje de bienvenida. Esta ventana tendrá en su interior un botón Cerrar que permitirá que el usuario la cierre haciendo clic en él. Tendrá el tamaño justo para visualizar el mensaje y no tendrá barras de scroll, ni de herramientas, ni de dirección... únicamente el mensaje.

## Diálogos

Hay 3 métodos del objeto *window* que ya conocemos y que nos permiten abrir ventanas de diálogo con el usuario:

- *window.alert(mensaje)*: muestra un diálogo con el mensaje indicado y un botón de 'Aceptar'
- *window.confirm(mensaje)*: muestra un diálogo con el mensaje indicado y botones de 'Aceptar' y 'Cancelar'. Devuelve *true* si se ha pulsado el botón de aceptar del diálogo y *false* si no.
- *window.prompt(mensaje [, valor predeterminado])*: muestra un diálogo con el mensaje indicado, un cuadro de texto (vacío o con el valor predeterminado indicado) y botones de 'Aceptar' y 'Cancelar'. Si se pulsa 'Aceptar' devolverá un *string* con el valor que haya en el cuadro de texto y si se pulsa 'Cancelar' o se cierra devolverá *null*.

## Objeto location

Contiene información sobre la URL actual del navegador y podemos modificarla. Sus principales propiedades y métodos son:

- *.href*: devuelve la URL actual completa
- *.protocol*, *.host*, *.port*: devuelve el protocolo, host y puerto respectivamente de la URL actual

- `.pathname`: devuelve la ruta al recurso actual
- `.reload()`: recarga la página actual
- `.assign(url)`: carga la página pasada como parámetro
- `.replace(url)`: ídem pero sin guardar la actual en el historial

**EJERCICIO:**

- a) muestra la ruta completa de la página actual
- b) muestra el servidor de esta página
- c) carga la página de Google usando el objeto *location*

```
<script>
    console.log( window.location.href);
    console.log(location.host);
    console.log(location.assign('https://www.google.es/'));
</script>
```

**Objeto history**

Permite acceder al historial de páginas visitadas y navegar por él:

- `.length`: muestra el número de páginas almacenadas en el historial
- `.back()`: vuelve a la página anterior
- `.forward()`: va a la siguiente página
- `.go(num)`: se mueve *num* páginas hacia adelante en el historial (si *num* es positivo), o hacia atrás (si *num* es negativo).

**EJERCICIO:** Vuelve a la página anterior**Otros objetos**

Los otros objetos que incluye BOM son:

- `document`: el objeto *document* que hemos visto en el DOM
- `navigator`: nos informa sobre el navegador y el sistema en que se ejecuta
  - `.userAgent`: muestra información sobre el navegador que usamos
  - `.plataform`: muestra información sobre la plataforma sobre la que se ejecuta
  - ...
- `screen`: nos da información sobre la pantalla
  - `.width/.height`: ancho/alto total de la pantalla (resolución)
  - `.availWidth/.availHeight`: igual pero excluyendo la barra del S.O.
  - ...

**EJERCICIO:** obtén todas las propiedades `width/height` y `availWidth/availHeight` del objeto *screen*. Compáralas con las propiedades `innerWidth/innerHeight` y `outerWidth/outerHeight` de *window*.



## DWEC – Javascript Web Cliente.

JavaScript 06 – Eventos.....	1
Introducción.....	1
Escuchar un evento utilizando un escuchador o <i>listener</i> .....	1
Event listeners.....	2
Tipos de eventos .....	3
Eventos de página .....	3
Eventos de ratón .....	3
Eventos de teclado.....	4
Eventos de toque .....	4
Eventos de formulario.....	4
Los objetos event y this .....	4
event .....	4
this .....	6
Bindeo del objeto this .....	7
Propagación de eventos (bubbling) .....	8
innerHTML y escuchadores de eventos .....	10
Eventos personalizados .....	11

# JavaScript 06 – Eventos

## Introducción

Los eventos permiten detectar acciones que realiza el usuario, o cambios que suceden en la página, y reaccionar como respuesta.

Existen muchos eventos diferentes, se puede ver la lista en [W3schools](#). 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.

## Escuchar un evento utilizando un escuchador o *listener*.

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 sería:

```
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, 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. (*También podemos evitarlo si el script es llamado al final del BODY*).

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');
    }
}
```

## Event listeners

La forma recomendada de escuchar un evento 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 el evento:

```
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 pasar algún parámetro a la función escuchadora, que a veces es conveniente, debemos usar funciones anónimas como escuchadores de eventos:

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

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

NOTA: igual que antes, debemos estar seguros de que se ha creado el árbol DOM antes de poner un escuchador, por lo que se recomienda ponerlos siempre dentro de la función asociada al evento `window.onload`. O mejor: `window.addEventListener('load', ...)` como se ve en el ejemplo anterior.

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 produzca 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 (tb. podemos usar *mouseover/mouseout*)
- **mousemove**: se produce continuamente mientras el puntero se *movea* dentro del elemento

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

### EJERCICIO:

- a) Pon un escuchador al botón 1 de la [página de ejemplo de DOM](#) para que al hacer click se muestre el un alert con ‘Click sobre botón 1’
- b) Pon otro escuchador al mismo botón para que se abra otra ventana nueva (de 200 px de ancho y 100 de alto) con un texto dentro que reze “Nueva ventana emergente”. **Nota:** Comprueba si hay diferencias si se abre la página desde “Live Server” o directamente como archivo local.
- c) Pon otro *listener* al mismo botón para que al pasar el ratón sobre él se muestre debajo de los botones un párrafo en rojo con la frase “Se va a abrir una ventana nueva”.
- d) Pon otro escuchador al mismo botón que al salir el cursor del ratón, desaparezca el párrafo del apartado anterior.
- e) Pon un escuchador al botón 2 que desactive el escuchador del primer apartado.

## 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 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 event y this

Al producirse un evento, se generan automáticamente en su función manejadora 2 objetos: **this** y **event**.

### event

**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 se ve en el ejemplo que hay un poco más abajo.
- **.currentTarget:** Identifica el target (objetivo) actual del evento, ya que el evento atraviesa el DOM. Siempre hace referencia al elemento al cual el controlador del evento fue asociado, a diferencia de *event.target*, que identifica el elemento en el que se produjo el evento.

**Ejemplo:** Tenemos un elemento *P* al que le ponemos un escuchador de ‘click’ que dentro tiene un elemento *STRONG*.

Si hacemos *\_click* sobre el elemento *STRONG*: **event.target** valdrá el *STRONG* que es donde hemos hecho click (está dentro de *<p>*), pero tanto para *P* como para *STRONG* **.event.currentTarget** valdrá el elemento *<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 hiperenlace hará que se ejecute su código y después saltará a la página indicada en el *href* del hiperenlace. Este método cancela la acción por defecto del navegador para el evento.

Otro ejemplo de uso de este método: si el evento era el *submit* de un formulario éste no se enviará, o si era un *click* sobre un hiperenlace no se irá a la página indicada en él.

- **.stopPropagation():** Como por defecto un evento se produce sobre un elemento y todos sus padres, al usar este método se evita esta propagación.

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 algún escuchador llama a este método, el evento no se propagará a los demás elementos padre.

*Un evento, dependiendo del tipo, puede tener 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 es 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`.
- Hay que capturar sólo lo que sea necesario, se producen muchos eventos de teclado.
- Para obtener el carácter a partir del código, se aconseja utilizar: `String.fromCharCode(n1, n2, ..., )`
- Lo mejor para familiarizarse con los diferentes eventos es consultar los [ejemplos de w3schools](#).

**EJERCICIO A:** Pon 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 los distintos DIV la posición del puntero respecto del navegador y respecto de la página.

**EJERCICIO B:** Pon un listener al BODY de la [página de ejemplo](#) para que al pulsar cualquier tecla nos muestre en un párrafo el `key` y el `keyCode` de la tecla pulsada. Pruébalo con diferentes teclas.

**this**

**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.

**Ojo!**: No usar funciones flecha. Más información en el documento JavaScript - Anexo - “Uso de this en contexto”.

Veamos un ejemplo:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Ejemplo 1 DOM</title>
  </head>
  <body>
    <h1 class="important">Ejemplo de eventos con this</h1>

    <form action="#">
      <label for="input1">
```

```

    >Dato 1 <input id="input1" type="text" size="20" /><br />
</label>
<label for="input2">
    >Dato 2 <input id="input2" type="text" size="20" /><br />
</label>
<label for="input3">Dato 3
    <input id="input3" type="text" size="20" /><br />
</label>
</form>
<script src="js/this.js"></script>
</body>
</html>

```

```

const arrayInputs=Array.from(document.getElementsByTagName('input'));

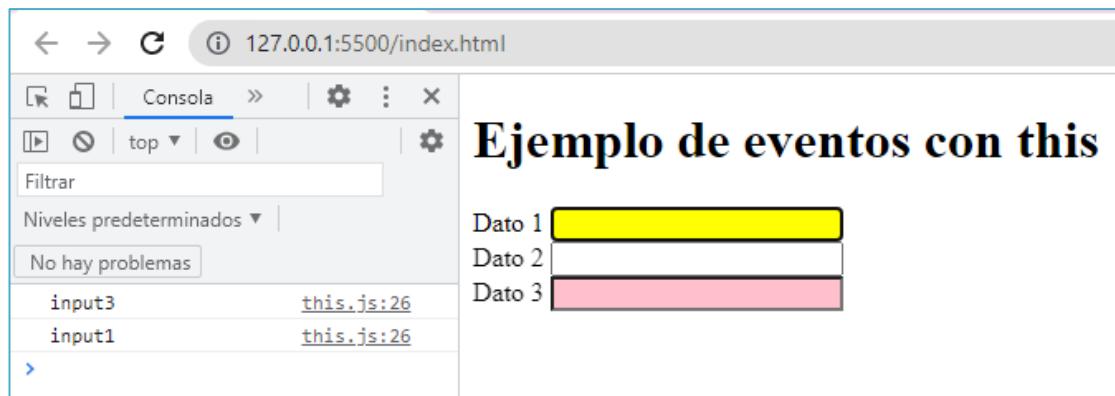
function pintar(even){
    console.log(even.target.id);
    this.style.backgroundColor='yellow';
}

function pintarRosa(even){
    this.style.backgroundColor='pink';
}

arrayInputs.forEach(element => {
    element.addEventListener('focus', pintar); //Al hacer clic los pone en amarillo
    element.addEventListener('blur', pintarRosa); // Al salir los deja rosa (para siempre)
});

```

Ejemplo de ejecución:



## 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 *this* antes de entrar a la función escuchadora.

Por ejemplo: Si la función escuchadora es un método de una clase, en *this* 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
}
```

Ejercicio: tarea 0xx : Una tabla de varia filas y columnas que vaya metiendo los valores de un array según se haga clic en la tabla.

## 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 `<spam>`) 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

Ejemplo:

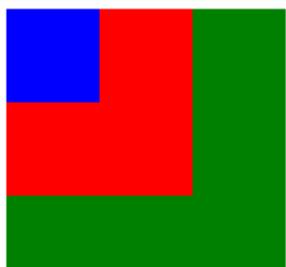
```
<div id="divVerde" style="background-color: green; width: 150px; height: 150px;">
```

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

```
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);
```

Veremos algo similar a :



### CASO A - Tercer parámetro por defecto (false)

Haciendo clic en el div Azul: veremos un resultado así:

```
Has pulsado: divAzul. Fase: 2, this: divAzul, event.target: divAzul, event.currentTarget: divAzul
Has pulsado: divRojo. Fase: 3, this: divRojo, event.target: divAzul, event.currentTarget: divRojo
Has pulsado: divVerde. Fase: 3, this: divVerde, event.target: divAzul, event.currentTarget: divVerde
```

Primero responde Azul en fase 2 (objetivo) y luego responden Rojo y Verde en fase 3 (burbujeo).

Observa que **event.target** siempre es Azul, mientras que **event.currentTarget** va cambiando.

### CASO B - Tercer parámetro por defecto (false)

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>`

Probando a añadir ese parámetro a los escuchadores del ejemplo anterior con los divs de colores:

```
divVerde.addEventListener('click', divClick, true );
divRojo.addEventListener('click', divClick, true);
```

```
divAzul.addEventListener('click', divClick, true );
```

Volviendo a hacer clic en el div Azul: veremos un resultado así:

```
Has pulsado: divVerde. Fase: 1, this: divVerde, event.target: divAzul, event.currentTarget: divVerde
Has pulsado: divRojo. Fase: 1, this: divRojo, event.target: divAzul, event.currentTarget: divRojo
Has pulsado: divAzul. Fase: 2, this: divAzul, event.target: divAzul, event.currentTarget: divAzul
```

Observamos que primero responden Verde y Rojo en fase 1 (captura), y por último Azul en fase 2 (objetivo).

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

Se pueden ver las distintas fases de un evento en la página [domevents.dev](http://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 click 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 (con 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()`.

Más información en la [página de MDN](#).



## DWEC – Javascript Web Cliente.

JavaScript 07 – Objetos nativos .....	1
Introducción.....	1
Funciones globales.....	1
Objetos nativos del lenguaje.....	2
Objeto Math .....	3
Objeto Date.....	4

# JavaScript 07 – Objetos nativos

## Introducción

En este tema vamos a ver:

- Las **funciones globales de Javascript**, muchas de las cuales ya hemos visto como `Number()` o `String()`.
- Los **objetos nativos** que incorpora Javascript y que nos facilitarán el trabajo proporcionándonos métodos y propiedades útiles para no tener que “reinventar la rueda” en nuestras aplicaciones.
- Uno de ellos está el objeto `RegExp` que nos permite trabajar con **expresiones regulares** (son iguales que en otros lenguajes) que nos serán de gran ayuda, sobre todo a la hora de validar formularios y que por eso veremos en la siguiente unidad.

## Funciones globales

- `parseInt(valor)`: devuelve el valor pasado como parámetro convertido a entero o `NaN` si no es posible la conversión. Este método es mucho más permisivo que `Number` y convierte cualquier cosa que comience por un número (si encuentra un carácter no numérico detiene la conversión y devuelve lo convertido hasta el momento). Ejemplos:

```
console.log( parseInt(3.84) )           // imprime 3 (ignora los decimales)
console.log( parseInt('3.84') )          // imprime 3
console.log( parseInt('28manzanas') )    // imprime 28
console.log( parseInt('manzanas28') )    // imprime NaN
```

- `parseFloat(valor)`: igual, pero devuelve un número decimal. Ejemplos:

```
console.log( parseFloat(3.84) )           // imprime 3.84
console.log( parseFloat('3.84') )          // imprime 3.84
console.log( parseFloat('3,84') )          // imprime 3 (La coma no es un carácter numérico)
console.log( parseFloat('28manzanas') )    // imprime 28
console.log( parseFloat('manzanas28') )    // imprime NaN
```

- `Number(valor)`: convierte el valor a un número. Es como `parseFloat` pero más estricto. Si no puede convertir todo el valor, devuelve `NaN`. Ejemplos:

```
console.log( Number(3.84) )           // imprime 3.84
console.log( Number('3.84') )          // imprime 3.84
console.log( Number('3,84') )          // imprime NaN (La coma no es un carácter numérico)
console.log( Number('28manzanas') )    // imprime NaN
console.log( Number('manzanas28') )    // imprime NaN
```

- `String(valor)`: convierte el valor pasado en una cadena de texto. Si le pasamos un objeto llama al método `toString()` del objeto. Ejemplos:

```
console.log( String(3.84) )           // imprime '3.84'
console.log( String([24, 3, 12]) )      // imprime '24,3,12'
console.log( {nombre: 'Marta', edad: 26} ) // imprime "[object Object]"
```

- `Boolean(valor)`: convierte el valor pasado a un booleano. Sería el resultado de tenerlo como condición en un `if`. Muchas veces en vez de usar esto usamos la doble negación `!!` que da el mismo resultado. Ejemplos:

```
console.log( Boolean('Hola') )         // Equivaldría a !!'Hola'. Imprime true
console.log( Boolean(0) )               // Equivaldría a !!0. Imprime false
```

- `isNaN(valor)`: devuelve `true` si lo pasado NO es un número o no puede convertirse en un número. Ejemplos:

```
console.log( isNaN(3.84) )            // imprime false
console.log( isNaN('3.84') )          // imprime false
console.log( isNaN('3,84') )          // imprime true (La coma no es un carácter numérico)
console.log( isNaN('28manzanas') )     // imprime true
console.log( isNaN('manzanas28') )     // imprime true
```

- `isFinite(valor)`: devuelve `false` si es número pasado es infinito (o demasiado grande)

```
console.log( isFinite(3.84) )          // imprime true
console.log( isFinite(3.84 / 0) )        // imprime false
```

- `encodeURI(string) / decodeURI(string)`: transforma la cadena pasada a una URL codificada válida, transformando los caracteres especiales que contenga, excepto `, / ? : @ & = + $ #`. Debemos usarla siempre que vayamos a pasar una URL. Ejemplo:
  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: "http://domain.com?val=1%202%203&val2=r+y%256"
- `encodeURIComponent(string) / decodeURIComponent(string)`: transforma también los caracteres que no transforma la anterior. Debemos usarla para codificar parámetros, pero no una URL entera. Ejemplo:
  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: "http%3A%2F%2Fdomain.com%3Fval%3D1%202%203%26val2%3Dr%2By%256"

## Objetos nativos del lenguaje

En Javascript casi todo son objetos. Ya hemos visto diferentes objetos:

- `window`

- `screen`
- `navigator`
- `location`
- `history`
- `document`

Los 5 primeros se corresponden al modelo de objetos del navegador y `document` se corresponde al modelo de objetos del documento. Todos nos permiten interactuar con el navegador para realizar distintas acciones.

También tenemos los tipos de objetos nativos, que no dependen del navegador. Son:

- `Number`
- `String`
- `Boolean`
- `Array`
- `Function`
- `Object`
- `Math`
- `Date`
- `RegExp`

Además de los tipos primitivos de **número**, **cadena**, **booleano**, **undefined** y **null**, Javascript tiene todos los objetos indicados. Como ya hemos visto, se puede crear un número usando su tipo primitivo (`let num = 5`) o su objeto (`let num = new Number(5)`) pero es mucho más eficiente usar los tipos primitivos. Pero aunque lo creemos usando el tipo de dato primitivo se considera un objeto y tenemos acceso a todas sus propiedades y métodos. Ejemplo: `num.toFixed(2)`

Ya hemos visto las principales propiedades y métodos de `Number`, `String`, `Boolean` y `Array` y aquí vamos a ver los de `Math` y `Date` y en el apartado de validar formularios, las de `RegExp`.

## Objeto Math

Proporciona constantes y métodos para trabajar con valores numéricos:

- **constantes:** `.PI` (número pi), `.E` (número de Euler), `.LN2` (algoritmo natural en base 2), `.LN10` (logaritmo natural en base 10), `.LOG2E` (logaritmo de E en base 2), `.LOG10E` (logaritmo de E en base 10), `.SQRT2` (raíz cuadrada de 2), `.SQRT1_2` (raíz cuadrada de 1/2). Ejemplos:

```
console.log( Math.PI )           // imprime 3.141592653589793
console.log( Math.SQRT2 )        // imprime 1.4142135623730951
```

- `Math.round(x)`: redondea x al entero más cercano
- `Math.floor(x)`: redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- `Math.ceil(x)`: redondea x hacia arriba (5.01 → 6)
- `Math.min(x1,x2,...)`: devuelve el número más bajo de los argumentos que se le pasan.
- `Math.max(x1,x2,...)`: devuelve el número más alto de los argumentos que se le pasan.
- `Math.pow(x, y)`: devuelve  $x^y$  (x elevado a y).
- `Math.abs(x)`: devuelve el valor absoluto de x.
- `Math.random()`: devuelve un número decimal aleatorio entre 0 (incluido) y 1 (no incluido).

Si queremos un número entre otros rangos haremos: `Math.random() * (max - min) + min`  
O si lo queremos sin decimales, haremos: `Math.round(Math.random() * (max - min) + min)`

- `Math.cos(x)`: devuelve el coseno de x (en radianes).
- `Math.sin(x)`: devuelve el seno de x.
- `Math.tan(x)`: devuelve la tangente de x.
- `Math.sqrt(x)`: devuelve la raíz cuadrada de x

Ejemplos:

```
console.log( Math.round(3.14) )      // imprime 3
console.log( Math.round(3.84) )      // imprime 4
console.log( Math.floor(3.84) )      // imprime 3
console.log( Math.ceil(3.14) )       // imprime 4
console.log( Math.sqrt(2) )          // imprime 1.4142135623730951
```

## Objeto Date

Es la clase que usaremos siempre que vayamos a trabajar con fechas. Al crear una instancia de la clase le pasamos la fecha que queremos crear o lo dejamos en blanco para que nos cree la fecha actual. Si le pasamos la fecha podemos pasarle:

- milisegundos, desde la fecha EPOCH
- cadena de fecha
- valor para año, mes (entre 0 y 11), día, hora, minutos, segundos, milisegundos

Ejemplos:

```
let date1=new Date()      // Mon Jul 30 2018 12:44:07 GMT+0200 (CEST) (es cuando he ejecutado la instrucción)
let date7=new Date(1532908000000)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (miliseg. desde 1/1/1070)
let date2=new Date('2018-07-30')     // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST) (La fecha pasada a las 0h. GMT)
let date3=new Date('2018-07-30 05:30') // Mon Jul 30 2018 05:30:00 GMT+0200 (CEST) (La fecha pasada a las 05:30h.
                                         Local)
let date6=new Date('07-30-2018')     // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (OJO: formato MM-DD-AAAA)
let date7=new Date('30-Jul-2018')     // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (tb. podemos poner 'Julio')
let date4=new Date(2018,7,30)        // Thu Ago 30 2018 00:00:00 GMT+0200 (CEST) (OJO: 0->Ene, 1->Feb... y a las 0h.
                                         Local)
let date5=new Date(2018,7,30,5,30)   // Thu Ago 30 2018 05:30:00 GMT+0200 (CEST) (OJO: 0->Ene, 1->Feb,...)
```

EJERCICIO: Crea en la consola dos variables fecNac1 y fecNac2 que contengan tu fecha de nacimiento. La primera la creas pasando una cadena y la segunda pasando año, mes y día.

Cuando ponemos la fecha como texto, como separador de las fechas podemos usar `,` / o `espacio`.

Como separador de las horas debemos usar `:`

Cuando ponemos la fecha como parámetros numéricos (separados por `,`) podemos poner valores fuera de rango que se sumarán al valor anterior. Por ejemplo:

```
let date=new Date(2018,7,41)      // Mon Sep 10 2018 00:00:00 GMT+0200 (CEST) -> 41=31Ago+10
let date=new Date(2018,7,0)        // Tue Jul 31 2018 00:00:00 GMT+0200 (CEST) -> 0=0Ago=31Jul (el
                                 anterior)
let date=new Date(2018,7,-1)       // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) -> -1=0Ago-1=31Jul-1=30Jul
```

OJO con el rango de los meses que empieza en 0->Ene, 1->Feb,...,11->Dic

Tenemos métodos **getter** y **setter** para obtener o cambiar los valores de una fecha:

- **fullYear**: permite ver (*get*) y cambiar (*set*) el año de la fecha:

```
let fecha=new Date('2018-07-30') // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getFullYear() ) // imprime 2018
fecha.setFullYear(2019) // Tue Jul 30 2019 02:00:00 GMT+0200 (CEST)
```

- **month**: devuelve/cambia el número de mes, pero recuerda que 0->Ene,...,11->Dic

```
let fecha=new Date('2018-07-30') // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getMonth() ) // imprime 6
fecha.setMonth(8) // Mon Sep 30 2019 02:00:00 GMT+0200 (CEST)
```

- **date**: devuelve/cambia el número de día:

```
let fecha=new Date('2018-07-30') // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() ) // imprime 30
fecha.setDate(-2) // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
```

- **day**: devuelve el número de día de la semana (0->Dom, 1->Lun, ..., 6->Sáb). Este método **NO tiene setter**:

```
let fecha=new Date('2018-07-30') // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDay() ) // imprime 1
```

- **hours, minutes, seconds, milliseconds, :** devuelve/cambia el número de hora, minuto, segundo o milisegundo, respectivamente.
- **time**: devuelve/cambia el número de milisegundos desde Epoch (1/1/1970 00:00:00 GMT):

```
let fecha=new Date('2018-07-30') // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getTime() ) // imprime 1532908800000
fecha.setTime(1000*60*60*24*25) // Fri Jan 02 1970 01:00:00 GMT+0100 (CET) (Le hemos añadido 25 días a Epoch)
```

**EJERCICIO:** Realiza en la consola los siguientes ejercicios (usa las variables que creaste antes)

- muestra el día de la semana en que naciste
- modifica fecNac1 para que contenga la fecha de tu cumpleaños de este año (cambia sólo el año)
- muestra el día de la semana de tu cumpleaños de este año
- calcula el nº de días que han pasado desde que naciste hasta hoy

Para mostrar la fecha tenemos varios métodos diferentes:

- **.toString()**: "Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)"
- **.toUTCString()**: "Mon, 30 Jul 2018 00:00:00 GMT"
- **.toDateString()**: "Mon, 30 Jul 2018"
- **.toTimeString()**: "02:00:00 GMT+0200 (hora de verano de Europa central)"
- **.toISOString()**: "2018-07-30T00:00:00.000Z"
- **.toLocaleString()**: "30/7/2018 2:00:00"
- **.toLocaleDateString()**: "30/7/2018"
- **.toLocaleTimeString()**: "2:00:00"

**EJERCICIO:** muestra en distintos formatos la fecha y la hora de hoy

**NOTA:** recuerda que las fechas son objetos y que se copian y se pasan como parámetro por referencia:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=fecha
otraFecha.setDate(28)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )         // imprime 28 porque fecha y otraFecha son el mismo
objeto
```

Una forma sencilla de copiar una fecha es crear una nueva pasándole la que queremos copiar:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=new Date(fecha)
otraFecha.setDate(28)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )         // imprime 30
```

En realidad lo que le estamos pasando es el tiempo Epoch de la fecha (es como hacer otraFecha=new Date(fecha.getTime()))

**NOTA:** la comparación entre fechas funciona correctamente con los operadores `>`, `>=`, `<` y `<=`

Pero **NO funciona** con `==`, `====`, `!=` y `!==` ya que compara los objetos y ve que son objetos diferentes.

Si queremos saber si 2 fechas son iguales (siendo diferentes objetos):

- el código que pondremos NO es `fecha1 === fecha2`
- **pondremos** `fecha1.getTime() === fecha2.getTime()`.

EJERCICIO: comprueba si es mayor tu fecha de nacimiento o el 1 de enero de este año.

Podemos probar los distintos métodos de las fechas en la página de [w3schools](#).



## DWEC – Javascript Web Cliente.

JavaScript 08 – Validación de formularios .....	1
Introducción.....	1
Validación del navegador incorporada en HTML5 .....	2
Validación mediante la API de validación de formularios.....	3
yup .....	8
Expresiones regulares .....	8
Patrones.....	8
Métodos.....	9

# JavaScript 08 – Validación de formularios

## Introducción

En este tema vamos a ver cómo realizar una de las acciones principales de Javascript que es la validación de formularios en el lado cliente.

Se trata de una verificación útil porque evita enviar datos al servidor que sabemos que no son válidos, pero NUNCA puede sustituir a la validación en el lado servidor, ya que en el lado cliente otro usuario puede manipular el código desde la consola y hacer que se salten las validaciones que le pongamos.

Antes de enviar datos al servidor, es importante asegurarse de que se completan todos los controles de formulario requeridos, y en el formato correcto. Esto se denomina **validación de formulario en el lado del cliente** y ayuda a garantizar que los datos que se envían coinciden con los requisitos establecidos en los diversos controles de formulario.

La validación en el lado del cliente es una verificación inicial y una característica importante para garantizar una buena experiencia de usuario mediante la detección de datos no válidos en el lado del cliente que el usuario puede corregir de inmediato. Si el servidor recibe datos y, a continuación, los rechaza, se produce un retraso considerable en la comunicación entre el servidor y el cliente que insta al usuario a corregir sus datos.

Si observamos una web bien hecha que incluya un formulario de registro, vereos que proporciona mensajes o comentarios cuando no se introducen datos en el formato que se espera. Se verán mensajes del tipo:

- «Este campo es obligatorio» (No se puede dejar este campo en blanco).
- «Introduzca su número de teléfono en el formato xxx-xxxx» (Se requiere un formato de datos específico para que se considere válido).
- «Introduzca una dirección de correo electrónico válida» (los datos que introdujiste no están en el formato correcto).
- «Su contraseña debe tener entre 8 y 30 caracteres y contener una letra mayúscula, un símbolo y un número». (Se requiere un formato de datos muy específico para tus datos).

Esto se llama **validación de formulario**: Cuando al introducir datos, el navegador y/o el servidor web verifican que estén en el formato correcto y dentro de las restricciones establecidas por la aplicación. La validación realizada

en el navegador se denomina **validación en el lado del cliente**, mientras que la validación realizada en el servidor se denomina **validación en el lado del servidor**. Aquí nos centraremos en la validación en el lado del cliente.

Si la información está en el formato correcto, la aplicación permite que los datos se envíen al servidor y (en general) que se guarden en una base de datos. Si la información no está en el formato correcto, da al usuario un mensaje de error que explica lo que debe corregir y le permite volver a intentarlo.

Razones principales para aplicar validación:

- **Queremos obtener los datos correctos en el formato correcto.** Nuestras aplicaciones no funcionarán correctamente si los datos de nuestros usuarios se almacenan en el formato incorrecto, son incorrectos o se omiten por completo.
- **Queremos proteger los datos de nuestros usuarios.** Obligar a nuestros usuarios a introducir contraseñas seguras facilita proteger la información de su cuenta.
- **Queremos protegernos a nosotros mismo.** Hay muchas formas en que los usuarios maliciosos puedan usar mal los formularios desprotegidos y dañar la aplicación (consulta el anexo [Seguridad de sitios web](#)).

Básicamente tenemos **2 maneras de validar un formulario en el lado cliente:**

1. Usar la validación incorporada en HTML5 y dejar que sea el navegador quien se encargue de todo.
2. Realizar nosotros la validación mediante Javascript.

La **ventaja** de la **primera opción** es que **no tenemos que escribir código**, sino simplemente poner unos atributos a los INPUT que indiquen qué se ha de validar. La **principal desventaja** es que **no tenemos ningún control sobre el proceso**, lo que **provocará cosas como:**

- El **navegador valida campo a campo**: cuando encuentra un error en un campo lo muestra y hasta que no se soluciona no valida el siguiente lo que hace que el proceso sea molesto para el usuario que no ve todo lo que hay mal de una vez.
- Los **mensajes son los predeterminados del navegador** y en ocasiones pueden no ser muy claros para el usuario.
- Los **mensajes se muestran en el idioma en que está configurado el navegador**, no en el de nuestra página.

## Validación del navegador incorporada en HTML5

Funciona añadiendo atributos a los campos del formulario que queremos validar. Los más usados son:

- **required**: indica que **el campo es obligatorio**. La validación fallará si no hay nada escrito en el input. En el caso de un grupo de *radiobuttons* se pone sobre cualquiera de ellos (o sobre todos) y obliga a que haya seleccionada una opción cualquiera del grupo.
- **pattern**: obliga a que **el contenido del campo cumpla la expresión regular** indicada. Por ejemplo, para un código postal sería pattern="`^[\d]{5}$`". Al final de este tema hay una pequeña introducción a las expresiones regulares en Javascript.
- **minlength / maxlength**: indica la longitud mínima/máxima del contenido del campo.
- **min / max**: indica el valor mínimo/máximo del contenido de un campo numérico.

También se producen errores de validación si el contenido de un campo no se adapta al **type** indicado (email, number, ...) o si el valor de un campo numérico no cumple con el **step** indicado.

Cuando el contenido de un campo **es válido**, dicho campo obtiene automáticamente la pseudoclase **:valid**

Si el contenido del campo **no es válido** tendrá la pseudoclase **:invalid**, lo que nos permite poner reglas en nuestro CSS para destacar dichos campo. Por ejemplo:

```
input:invalid {
    border: 2px dashed red;
}
```

La validación se realiza al enviar el formulario, y al encontrar un error se muestra dicho error, se detiene la validación del resto de campos y no se envía el formulario.

**Importante:** Tienes un pequeño resumen, y un [ejemplo de validación](#) de formularios desde HTML5, en el documento Anexo: "JavaScript – Anexo - Validar Formularios con HTML5.pdf"

## Validación mediante la API de validación de formularios

Mediante JavaScript tenemos acceso a todos los campos del formulario, por lo que podemos hacer la validación como queramos, pero es una tarea pesada, repetitiva y que provoca código spaghetti difícil de leer y mantener en el futuro.

Para hacerla más simple podemos usar la [API de validación de formularios](#) de HTML5 que permite que sea el navegador quien se encargue de comprobar la validez de cada campo, pero las acciones (mostrar mensajes de error, no enviar el formulario, ...) las realizamos desde Javascript.

Esto nos da la ventaja de:

- Los requisitos de validación de cada campo están como atributos HTML de dicho campo por lo que son fáciles de ver.
- Nos evitamos la mayor parte del código dedicada a comprobar si el contenido del campo es válido. Nosotros mediante la API sólo preguntamos si se cumplen o no, y tomamos las medidas adecuadas.
- Aprovechamos las pseudo-clases `:valid` o `:invalid` que el navegador pone automáticamente a los campos, por lo que no tenemos que añadir clases en CSS para destacarlos.

Las principales propiedades y métodos que nos proporciona esta API son:

- **checkValidity()**: método que nos dice si el campo al que se aplica es o no válido. También se puede aplicar al formulario para saber si es válido o no.
- **validationMessage**: en caso de que un campo no sea válido esta propiedad contiene el texto del error de validación proporcionado por el navegador. Si es válido esta propiedad es una cadena vacía
- **validity**: es un objeto que tiene propiedades booleanas para saber qué requisito del campo es el que falla:
  - **valueMissing**: indica si no se cumple el atributo **required** (es decir, valdrá *true* si el campo tiene el atributo *required* pero no se ha introducido nada en él)
  - **typeMismatch**: indica si el contenido del campo no cumple con su atributo **type** (ej. `type="email"`)
  - **patternMismatch**: indica si no se cumple con el **pattern** indicado en su atributo
  - **tooShort / tooLong**: indican si no se cumple el atributo **minlength** o **maxlength** respectivamente.
  - **rangeUnderflow / rangeOverflow**: indica si no se cumple el atributo **min / max**
  - **stepMismatch**: indica si no se cumple el atributo **step** del campo
  - **customError**: indica al campo que se le ha puesto un error personalizado con **setCustomValidity**
  - **valid**: indica si ese campo cumple con todas sus restricciones de validación y, por tanto, se considera válido.

- **setCustomValidity(mensaje)**: añade un error personalizado al campo (que ahora ya NO será válido) con el mensaje pasado como parámetro. Para quitar este error se hace `setCustomValidity('')`.

### Ejemplo de validación con JavaScript

```
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Favorite fruit start</title>
    <style>
      input:invalid {
        border: 2px dashed red;
      }

      input:valid {
        border: 2px solid greenyellow;
      }
    </style>
  </head>

  <body>
    <form>
      <label for="mail">Me gustaría que me proporcionara una dirección
          de correo electrónico:<label>
      <input type="email" id="mail" name="mail">
      <button>Enviar</button>
    </form>

    <script>
      const email = document.getElementById("mail");
      email.addEventListener("input", function (event) {
        if (email.validity.typeMismatch) {
          email.setCustomValidity("¡Se esperaba una dirección de correo electrónico!");
        } else {
          email.setCustomValidity("");
        }
      });
    </script>
  </body>
</html>
```

En este ejemplo la constante `email` guarda una referencia para el `input` de la dirección de correo electrónico, luego se le añade un detector de eventos que ejecuta el código de la función cada vez que el valor de la entrada cambia.

Dentro del código que contiene, verificamos si la propiedad `validity.typeMismatch` del `input` de la dirección de correo electrónico devuelve `true`, lo que significa que el valor que contiene no coincide con el patrón para una dirección de correo electrónico bien formada. Si es así, llamamos al método `setCustomValidity()` con un mensaje personalizado. Esto hace que la entrada no sea válida, de modo que cuando se intenta enviar el formulario, el envío falla y se muestra el mensaje de error personalizado.

Si la propiedad validity.typeMismatch devuelve false, se llama al método setCustomValidity() con una cadena vacía. Esto hace que la entrada sea válida y el formulario sea enviado.

#### Ejemplo para ver propiedades y métodos de la API

Veamos un ejemplo simple del valor de las diferentes propiedades involucradas en la validación de un campo de texto obligatorio y cuyo tamaño debe estar entre 5 y 50 caracteres. Prueba este código introduciendo en el input cadenas de texto de diferentes longitudes.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
        .error {
            color: red;
        }
    </style>
</head>
<body>
    <form action="">
        <label>Nombre:</label>
        <input type="text" required minlength="5" maxlength="50" />
        <span id="error" class="error"></span>
        <br />
        <button id="comprueba">Comprueba</button>
        <p>checkValidity: <span id="checkValidity"></span></p>
        <p>validationMessage: <span id="validationMessage"></span></p>
        <p>validity.valueMissing: <span id="valueMissing"></span></p>
        <p>validity.tooShort: <span id="tooShort"></span></p>
        <p>validity.tooLong: <span id="tooLong"></span></p>
    </form>
    <script>
        document
            .getElementById("comprueba")
            .addEventListener("click", (event) => {
                const inputName = document.getElementsByName("input")[0];

                document.getElementById("error").innerHTML = inputName.validationMessage;
                document.getElementById("checkValidity").innerHTML = inputName.checkValidity();
                document.getElementById("validationMessage").innerHTML = inputName.validationMessage;
                document.getElementById("valueMissing").innerHTML = inputName.validity.valueMissing;
                document.getElementById("tooShort").innerHTML = inputName.validity.tooShort;
                document.getElementById("tooLong").innerHTML = inputName.validity.tooLong;
            });
    </script>
</body>
</html>
```

**Ejemplo de validación usando “novalidate”**

Para validar un formulario nosotros, pero usando esta API, debemos añadir al <FORM> el atributo **novalidate** que hace que el navegador no se encargue de mostrar los mensajes de error, ni de decidir si se envía o no el formulario (aunque sí valida los campos), sino que lo haremos nosotros.

*index.html*

```
<form novalidate>
  <label for="nombre">Por favor, introduzca su nombre (entre 5 y 50 caracteres): </span>
  <input type="text" id="nombre" name="nombre" required minlength="5" maxlength="50">
  <span class="error"></span>
  <br />
  <label for="mail">Por favor, introduzca una dirección de correo electrónico: </label>
  <input type="email" id="mail" name="mail" required minlength="8">
  <span class="error"></span>
  <button type="submit">Enviar</button>
</form>
```

*main.js*

```
const form = document.getElementsByName('form')[0];

const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');
const email = document.getElementById('mail');
const emailError = document.querySelector('#mail + span.error');

form.addEventListener('submit', (event) => {
  if(!form.checkValidity()) {
    event.preventDefault();
  }
  nombreError.textContent = nombre.validationMessage;
  emailError.textContent = email.validationMessage;
});
```

*style.css*

```
.error {
  color: red;
}

input:invalid {
  border: 2px dashed red;
}
```

En el navegador se vería algo así:

Por favor, introduzca su nombre (entre 5 y 50 caracteres):

Por favor, introduzca una dirección de correo electrónico:

Y si se produce error en los inputs, al enviar se obtendrá algo como:

Por favor, introduzca su nombre (entre 5 y 50 caracteres):  Aumenta la longitud del texto a 5 caracteres como mínimo (actualmente, el texto tiene 3 caracteres).

Por favor, introduzca una dirección de correo electrónico:  Incluye un signo "@" en la dirección de correo electrónico. La dirección "sblanco" no incluye el signo "@".

Estamos usando:

- validationMessage para mostrar el posible error de cada campo, o quitar el error cuando el campo sea válido
- checkValidity() para no enviar/procesar el formulario si contiene errores

Si no nos gusta el mensaje del navegador y queremos personalizarlo, podemos hacer una función que reciba un <input> y, usando su propiedad validity, que devuelva un mensaje en función del error detectado:

```
function customErrorValidationMessage(input) {
  if (input.checkValidity()) {
    return ''
  }
  if (input.validity.valueMissing) {
    return 'Este campo es obligatorio'
  }
  if (input.validity tooShort) {
    return `Debe tener al menos ${input.minLength} caracteres`
  }
  // Y seguiremos comprobando cada atributo que hayamos usado en el HTML
  return 'Error en el campo' // por si se nos ha olvidado comprobar algo
}
```

Y ahora en vez de nombreError.textContent = nombre.validationMessage haremos nombreError.textContent = customErrorValidationMessage(nombre).

Si tenemos que validar algo que no puede hacerse mediante atributos HTML (por ejemplo si el nombre de usuario ya está en uso) deberemos hacer la validación “a mano” y, en caso de no ser válido, ponerle un error con .setCustomValidation().

Pero debemos recordar quitar el error si todo es correcto o el formulario siempre será inválido. Modificando el ejemplo quedaría:

```
const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');

if (nombreEnUso(nombre)) {
  nombre.setCustomValidation('Ese nombre de usuario ya está en uso')
} else {
  nombre.setCustomValidation('') // Se quita el error personalizado
}
form.addEventListener('submit', (event) => {
  if(!form.checkValidity()) {
    ...
  }
})
```

## yup

Existen múltiples librerías que facilitan enormemente el tedioso trabajo de validar un formulario. Un ejemplo es [yup](#).

# Expresiones regulares

Las expresiones regulares permiten buscar un patrón dado en una cadena de texto. Se usan mucho a la hora de validar formularios o para buscar y reemplazar texto.

En JavaScript pueden crearse expresiones regulares de dos formas:

1. Poniéndolas entre caracteres barra / (forma recomendada)
2. Instanciándolas desde la clase `RegExp`

```
let cadena='Hola mundo';
let expr=/mundo/;
expr.test(cadena);      // devuelve true porque en La cadena se encuentra La expresión 'mundo'
```

## Patrones

La potencia de las expresiones regulares es que podemos usar patrones para construir la expresión. Los más comunes son:

- **[..] (corchetes):** dentro se ponen varios caracteres o un rango y permiten comprobar si el carácter de esa posición de la cadena coincide con alguno de ellos. Ejemplos:
  - `[abc]`: cualquier carácter de los indicados ('a' o 'b' o 'c')
  - `[^abc]`: cualquiera excepto los indicados
  - `[a-z]`: cualquier minúscula (el carácter '-' indica el rango entre 'a' y 'z', incluidas)
  - `[a-zA-Z]`: cualquier letra
- **( | ) (pipe):** debe coincidir con una de las opciones indicadas:
  - `(x|y)`: la letra x o la y (sería equivalente a `[xy]`)
  - `(http|https)`: cualquiera de las 2 palabras
- **Metacaracteres:**
  - `.` (punto): un único carácter, sea el que sea
  - `\d`: un dígito (`\D`: cualquier cosa menos dígito)
  - `\s`: espacio en blanco (`\S`: lo opuesto)
  - `\w`: una palabra o carácter alfanumérico (`\W` lo contrario)
  - `\b`: delimitador de palabra (espacio, pppio, fin)
  - `\n`: nueva línea
- **Cuantificadores:**
  - `+`: al menos 1 vez (ej. `[0-9]+` al menos un dígito)
  - `*`: 0 o más veces
  - `?`: 0 o 1 vez
  - `{n}`: n caracteres (ej. `[0-9]{5}` = 5 dígitos)
  - `{n,}`: n o más caracteres
  - `{n,m}`: entre n y m caracteres
  - `^`: al ppio de la cadena (ej.: `^[a-zA-Z]` = empieza por letra)
  - `$`: al final de la cadena (ej.: `[0-9]$` = que acabe en dígito)
- **Modificadores:**

- `/i`: que no distinga entre Maysc y minsc (Ej. `/html/i` = buscará html, Html, HTML, ...)
- `/g`: búsqueda global, busca todas las coincidencias y no sólo la primera
- `/m`: busca en más de 1 línea (para cadenas con saltos de línea)

EJERCICIO: contruye una expresión regular para lo que se pide a continuación y pruébala con distintas cadenas:

- un código postal
- un NIF formado por 8 números, un guión y una letra mayúscula o minúscula
- un número de teléfono y aceptamos 2 formatos: XXX XX XX XX o XXX XXX XXX. El primer número debe ser un 6, un 7, un 8 o un 9

## Métodos

Los usaremos para saber si la cadena coincide con determinada expresión o para buscar y reemplazar texto:

- `expr.test(cadena)`: devuelve `true` si la cadena coincide con la expresión. Con el modificador `/g` hará que cada vez que se llama busque desde la posición de la última coincidencia. Ejemplo:

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hay solo dos coincidencias

let reg2 = /am/gi;          // ahora no distinguirá mayúsculas y minúsculas
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true. Ahora tenemos 3 coincidencias con este nuevo
patrón
```

- `expr.exec(cadena)`: igual pero en vez de `true` o `false` devuelve un objeto con la coincidencia encontrada, su posición y la cadena completa:

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null
```

- `cadena.match(expr)`: igual que `exec` pero se aplica a la cadena y se le pasa la expresión. Si ésta tiene el modificador `/g` devolverá un array con todas las coincidencias:

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(str.match(reg)); // Imprime ["am", "am", "Am"]
```

- `cadena.search(expr)`: devuelve la posición donde se encuentra la coincidencia buscada, o -1 si no aparece.
- `cadena.replace(expr, cadena2)`: devuelve una nueva cadena con las coincidencias de la cadena reemplazadas por la cadena pasada como 2º parámetro:

```
let str = "I am amazed in America";
console.log(str.replace(/am/gi, "xx")); // Imprime "I xx xxazed in xxerica"

console.log(str.replace(/am/gi, function(match) {
    return "-" + match.toUpperCase() + "-";
})); // Imprime "I -AM- -AM-azed in -AM-erica"
```

No vamos a profundizar más sobre las expresiones regulares. Es muy fácil encontrar por internet la que necesitemos en cada caso (para validar un e-mail, un NIF, un CP, ...). Podemos aprender más en:

- [w3schools](#)
- El anexo de Expresiones Regulares
- Y muchas otras páginas

También, hay páginas que nos permiten probar expresiones regulares con cualquier texto, como [regexr](#).



## DWEC - Javascript Web Cliente.

JavaScript – Formato JSON .....	1
Introducción.....	1
Características del formato .....	1
Ejemplos.....	1
Estructura de los datos .....	3
Métodos del objeto JSON .....	3

# JavaScript – Formato JSON

## Introducción

El formato JSON (acrónimo de JavaScript Object Notation, 'notación de objeto de JavaScript') es una forma de convertir objetos Javascript en una cadena de texto para poderlos enviar.

### Características del formato

- La estructura de datos JSON se compone de un conjunto de objetos o arrays que contendrán números, cadenas booleanos y nulos.
- Un objeto JSON comienza y termina con llaves, y contiene una colección desordenada de pares nombre-valor.
- Cada nombre y valor están separados por dos puntos, y los pares están separados por comas.
- La coma final está prohibida.
- El nombre es una cadena entre comillas dobles. Los caracteres de comillas no deben ser inclinadas o "inteligentes".
- En los números, los ceros a la izquierda están prohibidos; un punto decimal debe estar seguido al menos por un dígito.
- En las cadenas deben estar entre comillas dobles. No se permiten todos los caracteres de escape; sí se permiten los caracteres de separador de línea Unicode (U+2028) y el separador de párrafo (U+2029).
- Un array JSON comienza y termina con corchetes y contiene una colección ordenada de valores separados por comas. Un valor puede ser una cadena entre comillas dobles, un número, un booleano true o false, nulo, un objeto JSON o un array.
- Los objetos y los arrays JSON se pueden anidar, lo que posibilita una estructura jerárquica de datos.

### Ejemplos

En el siguiente ejemplo, se muestra una estructura de datos JSON con dos objetos válidos.

```
{  
  "id": 1006410,  
  "title": "Amazon Redshift Database Developer Guide"  
}
```

```
{
  "id": 100540,
  "name": "Amazon Simple Storage Service User Guide"
}
```

En el siguiente se muestran los **mismos datos como dos arrays JSON**:

```
[
  1006410,
  "Amazon Redshift Database Developer Guide"
]
[
  100540,
  "Amazon Simple Storage Service User Guide"
]
```

### Conversiones de objetos

El **objeto alumno**:

```
let alumno = {
  id: 5,
  nombre: 'Ana',
  apellidos: 'Zubiri Peláez'
}
```

se transformaría en la cadena de texto:

```
{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }
```

y el array:

```
let alumnos = [
  {
    id: 5,
    nombre: "Ana",
    apellidos: "Zubiri Peláez"
  },
  {
    id: 7,
    nombre: "Carlos",
    apellidos: "Pérez Ortíz"
  },
]
```

Se transformaría en la cadena:

```
[{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }, { "id": 7, "nombre": "Carlos", "apellidos": "Pérez Ortíz" }]
```

Nótese que tanto las claves como los valores van entrecomillados (con comillas dobles). No sirven comillas simples.

## Estructura de los datos

Los mismos datos pueden tener distinta estructura.

Archivo colores1.json	Archivo colores2.json	Archivo colores3.json
<pre>{   "arrayColores": [     {       "nombreColor": "rojo",       "valorHexadec": "#f00"     },     {       "nombreColor": "verde",       "valorHexadec": "#0f0"     },     {       "nombreColor": "azul",       "valorHexadec": "#00f"     },     {       "nombreColor": "cyan",       "valorHexadec": "#0ff"     },     {       "nombreColor": "magenta",       "valorHexadec": "#f0f"     },     {       "nombreColor": "amarillo",       "valorHexadec": "#ff0"     },     {       "nombreColor": "negro",       "valorHexadec": "#000"     }   ] }</pre>	<pre>{   "arrayColores": [     {       "rojo": "#f00",       "verde": "#0f0",       "azul": "#00f",       "cyan": "#0ff",       "magenta": "#f0f",       "amarillo": "#ff0",       "negro": "#000"     }   ] }</pre>	<pre>{   "rojo": "#f00",   "verde": "#0f0",   "azul": "#00f",   "cyan": "#0ff",   "magenta": "#f0f",   "amarillo": "#ff0",   "negro": "#000" }</pre>

Los ejemplos anteriores representan lo que podrían ser archivos JSON conteniendo datos en formato JSON.

Se trata de 3 archivos que contienen aproximadamente la misma información. Sin embargo, hay algunas diferencias:

- En el **archivo colores1.json** existe un único objeto de datos donde el nombre es *arrayColores* y su valor es **un array de objetos JSON**. Cada objeto del array está formado por los pares (*nombreColor* y su valor), y (*valorHexadec* y su valor). En este ejemplo en concreto el array consta de 7 elementos con información correspondiente a 7 colores.
- En el **archivo colores2.json** existe un único objeto de datos donde el nombre es *arrayColores*, cuyo valor es **un array que contiene un único objeto JSON formado por siete pares** (nombre – valor) que representa información sobre siete colores.
- En el **archivo colores3.json** existe un único objeto de datos que está formado por siete pares (nombre – valor) que representa información sobre siete colores.

Siendo las 3 formas válidas, se deberá utilizar aquella que se indique en las instrucciones o, de no existir pautas precisas, utilizar aquel diseño que favorezca el desarrollo y mantenimiento de la aplicación.

## Métodos del objeto JSON

Para convertir objetos en cadenas de texto JSON y viceversa, Javascript proporciona 2 métodos:

- **JSON.stringify(objeto)**: recibe un objeto JS y devuelve la cadena de texto correspondiente.

```
// convierte de formato JSON a objeto
const cadenaAlumnos = JSON.stringify(alumnos)
```

- **JSON.parse(cadena)**: realiza el proceso inverso, convirtiendo una cadena de texto en un objeto.

```
// convierte un objeto a formato JSON
```

```
const alumnos = JSON.parse(cadenaAlumnos)
```



## DWEC - Javascript Web Cliente.

JavaScript – document.cookie .....	1
Introducción:.....	1
Persistencia de la información .....	1
Concepto de cookie.....	2
Tiempo de vida de las cookies.....	3
Contenidos de las cookies.....	4
Cookies con Javascript: document.cookie .....	4
Crear una cookie .....	4
Modificar una cookie .....	6
Eliminar cookies .....	6
Recuperar el contenido de cookies .....	7
Saber si las cookies están habilitadas o no.....	7
Ejercicio HacerAlgoUnaSolaVez .....	8

# JavaScript – cookies y document.cookie

## Introducción:

Una cookie es un archivo que una página web guarda en el ordenador del cliente, y proporciona un sistema para que desde el servidor se pueda identificar al cliente sin necesidad de registrarse en el servidor.

En esta web hay buena información sobre los tipos de cookies: <https://www.xataka.com/basics/que-cookies-que-tipos-hay-que-pasa-desactivas>

## Persistencia de la información

Durante la navegación por una aplicación web, el usuario podrá visitar numerosos documentos HTML que se corresponderán con diferentes urls. Como las variables en JavaScript tienen una vida limitada al propio documento HTML en el que van insertas o enlazadas y su información desaparece cuando se carga un nuevo documento HTML, a menudo se necesita un sistema para disponer del valor establecido en algunas variables desde otro documento.

Cargar un archivo común donde definamos unas variables nos puede servir para disponer de dichas variables con su valor inicial en diferentes urls, pero si el valor de esa variable se modifica y continúa la navegación, no dispondremos de información sobre dicha modificación en el resto de urls, ya que “se perderán”.

Lo que vamos buscando es la persistencia de la información, es decir, que no desaparezca la información cuando cambiamos de url.

Hay diferentes maneras de abordar este problema que trataremos en éste y otros capítulos:

- Desde el lado cliente:
  - Utilizando cookies.

- Utilizando almacenamiento en el navegador (localStorage y sessionStorage).
- Desde el lado servidor:
  - Transmitiendo y almacenando información en una base de datos para recuperarla posteriormente.

## Concepto de cookie

Las cookies fueron creadas por trabajadores de la empresa Netscape como forma de dar una respuesta sencilla a la necesidad de almacenar información relacionada con la identificación de usuarios y acciones que un usuario desarrolla durante la navegación. Por ejemplo, se planteaba la siguiente cuestión: si un usuario accede a una tienda web y queremos que pueda ir agregando productos a un carrito de compras, ¿cómo saber qué productos ha almacenado cuando cambia de url? Sin el uso de alguna forma de dotar de persistencia a la información, la información se perdía. Y cierta información sería muy costoso manejarla con herramientas como bases de datos.

La forma de solucionarlo fue inventar las cookies. Las cookies son información que se almacena en el navegador de forma persistente, es decir, que no desaparece cuando el usuario navega a través de diferentes urls. Además, las cookies son enviadas al servidor cuando el usuario hace una petición, de modo que, el servidor puede conocer esa información y actuar en consecuencia.

Las cookies pueden ser creadas de diferentes maneras, por ejemplo:

a) Ser creadas por el servidor, y enviarlas al navegador para que las almacene. Supongamos que entramos en una página web de compras por internet. En ese momento somos usuarios anónimos, pero una vez introduzcamos nuestro nombre de usuario y password (por ejemplo, supongamos que somos el usuario Albert Einstein), el servidor envía una cookie al navegador que podría ser:

`session_id_ = 6n4465736gf9863b52e641757fa0b7db`, donde `session_id` es el nombre la cookie y la cadena de letras y números el valor que tiene la cookie, lo que permitirá saber al servidor que quien hace una petición es la misma persona (el mismo computador cliente) que había hecho una petición anteriormente. La comprobación de que sea el mismo computador cliente (realmente se refiere al mismo navegador cliente) se basa en comparar el valor de esa cadena larga: si coincide, es el mismo usuario, si es diferente, es otro usuario.

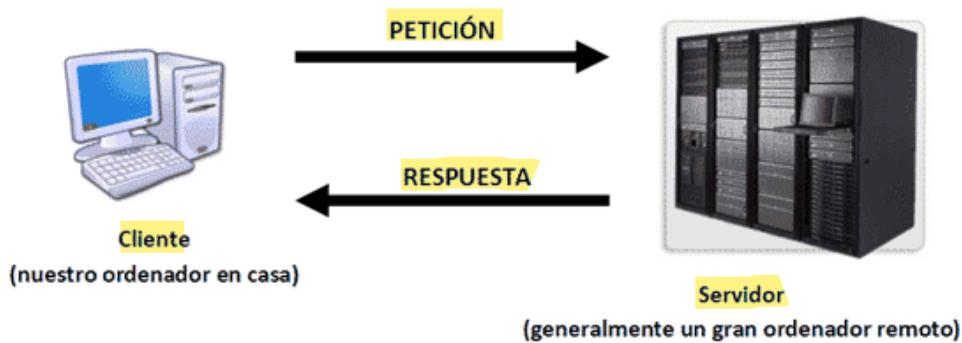
Un usuario no puede conocer el valor de la cookie que tiene otro debido a que existen prácticamente infinitas posibilidades de combinación de letras y números. Si se usan cadenas arbitrariamente largas, es prácticamente imposible averiguar por casualidad o mediante pruebas el contenido de una cookie.

Ahora bien, cada vez que cambiemos de url esta cookie es enviada al servidor. El servidor lee la cookie y comprueba: ¡esta cookie pertenece al usuario Albert Einstein!

Aunque el usuario haya cambiado de url, la cookie informa en cada momento que él es Albert Einstein, con lo cual no hace falta estar introduciendo en cada url que se visite el nombre de usuario y password.

b) Ser creadas mediante JavaScript en el navegador, almacenarse en el navegador y enviarse posteriormente al servidor en cada comunicación que tenga lugar.

Simplificadamente podríamos decir que navegar sin cookies sería algo similar a lo que se muestra en este esquema, donde cada petición al servidor equivale a una “petición nueva”:



Mientras que navegar con cookies sería algo similar a lo que se muestra en este esquema, donde el navegador está almacenando información en un almacén de cookies y con cada petición que hace esas cookies son transmitidas (y además pueden irse añadiendo más cookies a las ya existentes previamente):



**Resumiendo:** las cookies son datos que se almacenan en el navegador del usuario.

Las cookies pueden presentar algunos problemas de seguridad, pero en general se considera que si se utilizan de forma adecuada son un mecanismo que resulta práctico para realizar muchas de las tareas que normalmente se requieren en la navegación web.

## Tiempo de vida de las cookies

Las cookies son datos temporales, es decir, su intención no es almacenarse “para siempre”, sino almacenarse por un tiempo para facilitar la navegación. Una cookie puede tener asociada una fecha de borrado o expiración, en cuyo caso permanecerá en el navegador del usuario hasta que llegue dicha fecha (a no ser que el usuario decida hacer un borrado de cookies). En este caso, puede ocurrir que el usuario cierre el navegador y lo abra al cabo de unas horas o días, y la información en forma de cookies siga estando ahí. Las cookies con fecha de borrado se suelen llamar cookies persistentes, porque no se destruyen excepto cuando llega la fecha de expiración.

Otras cookies no tienen fecha de borrado o expiración, o si la tienen es muy corta (pongamos que una hora de duración). Si la cookie no tiene fecha de borrado, se destruye cuando se cierra el navegador.

Hay que tener en cuenta que los navegadores pueden almacenar información de otras maneras además de como cookies (por ejemplo, como opciones de configuración, perfiles de usuario, contraseñas, etc.).

Además, las cookies (al igual que JavaScript) pueden desactivarse en los navegadores. La mayoría de los usuarios navegan con cookies activadas (al igual que con JavaScript activado), pero teóricamente un usuario puede deshabilitarlas.

## Contenidos de las cookies

Las cookies podemos verlas como pequeños ficheros de texto que se almacenan en el navegador, cuyo contenido es el siguiente:

1. Un par nombre – valor que define la cookie. Por ejemplo, el nombre puede ser `user_name` y el valor `pelaez`.
2. Una fecha de caducidad (en algunos casos, estará indefinida, con lo cual la cookie será borrada cuando se cierre el navegador). La fecha se expresa en tiempo UTC, o como un número de segundos desde el momento actual.
3. El dominio y ruta del servidor donde la cookie fue definida, lo que permite que si existieran dos cookies con el mismo nombre se pudiera saber qué cookie corresponde a cada url que se visita. No está permitido falsear dominios, es decir, si el dominio desde el que se establece una cookie es `daw2cliente.com`, no se podría poner como información asociada a la cookie que el dominio es `microsoft.com`, sino que la cookie únicamente puede ir asociada a `daw2cliente.com`. Esto permite que, si se está navegando por un sitio, no haya necesidad de enviar todas las cookies almacenadas en el navegador al servidor de ese sitio, sino únicamente las cookies relacionadas con ese sitio.

La ruta permite especificar un directorio específico al cual se debe considerar asociada la cookie. De este modo, el navegador no tendría que enviar la cookie a todas las páginas de un dominio, sino solo a las páginas concretas cuya ruta esté definida. Normalmente la ruta definida es simplemente `</>` lo que significa que la cookie es válida en todo el dominio.

## Cookies con JavaScript: document.cookie

Las cookies podemos verlas como pequeños ficheros de texto que se almacenan en el navegador pero que desde el punto de vista de Javascript es una cadena de texto que contiene parejas `clave=valor` separadas por ; (punto y coma).

```
<nombre>=<valor>; expires=<fecha>; max-age=<segundos>; path=<ruta>; domain=<dominio>; secure;
```

Para obtener todas las cookies accesibles desde una localización se utiliza la propiedad:

```
let todasLasCookies = document.cookie;
```

### Crear una cookie

Para crear una cookie con JavaScript usaremos la siguiente sintaxis:

```
document.cookie=nuevaCookie;
```

donde `nuevaCookie` es una cadena de texto con pares clave-valor separadas por punto y coma.

*Los atributos son:*

`<nombre>=<valor>`

**Requerido.** `<nombre>` es el nombre (key) que identifica la cookie y `<valor>` es su valor. A diferencia de las cookies en PHP, en JavaScript se puede crear una cookie con un valor vacío (`<nombre>=`).

`expires=<fecha> y max-age=<segundos>`

**Opcional.** Ambos parámetros especifican el tiempo de validez de la cookie. `expires` establece una fecha (ha de estar en formato UTC) mientras que `max-age` establece una duración máxima en segundos. `max-`

`age` toma preferencia sobre `expires`. Si no se especifica ninguno de los dos se creará una **session cookie**. Si es `max-age=0` o `expires=fechaPasada` la cookie se elimina.

#### `path=<ruta>`

Opcional. Establece la ruta para la cual la cookie es válida. Si no se especifica ningún valor, la cookie será válida para la ruta la página actual.

#### `domain=<dominio>`

Opcional. Dentro del dominio actual, se puede indicar el subdominio para el que la cookie es válida. El valor predeterminado es el subdominio actual. Establecer `domain=.miweb.com` para una cookie que sea válida para cualquier subdominio (nota el punto delante del nombre del dominio). Por motivos de seguridad, los navegadores no permiten crear cookies para dominios diferentes al que crea la cookie (*same-origin policy*).

#### `secure`

Opcional. Atributo sin valor. Si está presente la cookie sólo es válida para conexiones encriptadas (por ejemplo, mediante protocolo HTTPS).

#### Ejemplos de creación de cookie:

```
document.cookie= "nombreCookie=valorCookie; expires=fechaDeExpiración; path=rutaParaLaCookie";
```

Se pueden añadir a la cadena de texto otros parámetros clave-valor.

También se puede dejar sin especificar `expires` (la cookie se borrará al cerrar el navegador) y `path` (la cookie quedará asociada al dominio), con lo que la definición quedaría:

```
document.cookie = 'nombreCookie=valorCookie; ';
```

Las cookies se envían en las cabeceras HTTP y, por tanto, deben estar correctamente codificadas. Conviene utilizar `encodeURIComponent()` para evitar sorpresas. Ejemplo:

```
let testvalue = "Hola mundo!";
document.cookie = "nombreCookie=" + encodeURIComponent( testvalue );
```

Si se va a utilizar el atributo `expires`, ha de ser con una fecha en formato UTC. Puede ser de ayuda el método `Date.toUTCString()`. Por ejemplo, una cookie con caducidad para el 1 de Febrero del año 2068 a las 11:20:

```
let expiresdate = new Date(2068, 1, 02, 11, 20);
let cookievalue = "Hola Mundo!";
document.cookie = "testcookie=" + encodeURIComponent( cookievalue ) + "; expires=" +
expiresdate.toUTCString();
```

En lugar de `expires` se puede usar `max-age`. En este caso, en lugar de especificar una fecha concreta se especifica el número de segundos desde la creación de la cookie hasta su caducidad. Por ejemplo:

```
document.cookie = "color=blue; max-age=" + 60*60*24*30 + "; path=/; domain=daw2cliente.com; secure"
```

El ejemplo anterior serviría para indicar que el nombre de la cookie es *color*, su valor *blue*, su fecha de expiración 30 días (expresado en segundos resulta 30 días \* 24 horas/día \* 60 minutos/hora \* 60 segundos/minuto").

Si no se especifica `max-age` ni `expires`, la cookie expirará al terminar la sesión actual. Si se especifican ambos atributos, tiene prioridad `max-age`.

### La propiedad document.cookie

Cuando se van creando cookies, éstas se van añadiendo a document.cookie (es decir, document.cookie no funciona como una propiedad que se vaya sobrescribiendo, sino que cada definición de document.cookie añade una cookie a la colección de cookies del documento). Este comportamiento se debe a que document.cookie no es un dato con un valor, sino una propiedad de acceso con métodos set y get nativos. Cada vez que se le asigna una nueva cookie, no se sobrescriben las cookies anteriores, sino que la nueva se añade a la colección de cookies del documento.

### Modificar una cookie

Una cookie se puede redefinir: para ello simplemente hemos de usar document.cookie indicando el mismo nombre de cookie que existiera previamente. Los datos de esa cookie serán sobrescritos, quedando reemplazados los anteriormente existentes por los nuevos.

Es importante tener en cuenta que, si una cookie se crea para un dominio o para un path determinado y se quiere modificar, el dominio y el path han de coincidir. De lo contrario se crearán dos cookies diferentes válidas para cada path y dominio. Por ejemplo, imaginemos que estamos en «miweb.com/blog» (el valor predeterminado del path es en este caso /blog):

```
// Supongamos que estamos en "miweb.com/blog"
// y creamos las siguientes cookies

// Creamos la cookie para el path "/"
document.cookie = "nombre=Miguel; path=/";

// Con la siguiente linea se crea una nueva cookie para el path "/blog" (valor por defecto)
// pero no se modifica la cookie "nombre" anterior porque era para un path diferente
document.cookie = "nombre=Juan";

// Con la siguiente línea SÍ se modifica la cookie "nombre" del path "/" correctamente
document.cookie = "nombre=Juan; path=/";
```

### Eliminar cookies

Una cookie se puede eliminar: para ello hemos de usar document.cookie indicando el nombre de cookie que queremos eliminar y establecer una fecha de expiración ya pasada (por ejemplo del día de ayer, o simplemente indicar **expires=Thu, 01 Jan 1970 00:00:00 UTC**). El navegador al comprobar que la fecha de caducidad de la cookie ha pasado, la eliminará.

Por ejemplo, creamos la cookie con el identificador nombre y valor Miguel igual que antes:

```
document.cookie = "nombre=Miguel";
```

Tenemos dos formas de eliminarla:

```
document.cookie = "nombre=; expires=Thu, 01 Jan 1970 00:00:00 UTC";

// O con max-age
document.cookie = "nombre=; max-age=0";
```

## Recuperar el contenido de cookies

Para recuperar el contenido de cookies hemos de trabajar con `document.cookie` como si fuera una cadena de texto

Para recuperar el valor de la cookie, hemos de buscar dentro de esa cadena de texto (string). Para ello usaremos las herramientas que nos proporciona JavaScript.

```
let todasLasCookies = document.cookie;
```

En la variable `todasLasCookies` las cookies se organizan de la siguiente manera:

```
nombreCookieA=valorCookieA; nombreCookieB=valorCookieB; ... ; nombreCookieN = valorCookieN;
```

A tener en cuenta:

- El string sólo contiene pares de nombre de la cookie y su valor. No se puede acceder a otros parámetros a través de `document.cookie`.
- Sólo se obtienen las cookies válidas para el documento actual. Esto implica que cookies para otros paths, dominios o cookies caducadas no se pueden leer. Aunque en una página puedan crearse cookies para otros subdominios y paths, sólo se pueden leer las que sean válidas para el subdominio y path actual.

Por ejemplo, imagina que estamos en el subdominio `noticias.miweb.com`. Aquí podemos crear una cookie para el subdominio `tienda.miweb.com`, pero **esta cookie no es válida para el documento en el que estamos** (`noticias.miweb.com`), por lo que no podemos leer su valor desde aquí, aunque sí hemos podido crearla:

```
// Suponiendo que estamos en noticias.miweb.com

// Se crean dos cookies para dos subdominios diferentes
document.cookie = "cookienoticias=valorcn; domain=noticias.miweb.com";
document.cookie = "cookietienda=valorct; domain=tienda.miweb.com";

let lasCookies = document.cookie;
alert( lasCookies );
// Obtenemos cookienoticias=valorcn
// No podemos acceder a la cookie cookietienda
// porque es válida solo para tienda.miweb.com y estamos en noticias.miweb.com
```

Para leer el contenido de cookies individuales se hace manipulando el string con todas las cookies y dividirlo por cada `;` para separar cada par `nombrecookie=valor`. Luego se divide cada uno de estos pares por `=` para separar el nombre de la cookie y su valor. Se puede conseguir utilizando varios métodos.

Ejercicio: Realiza y prueba una función que obtenga el valor de una cookie pasando su nombre como parámetro.

Ejercicio: utilizando expresiones regulares, realiza y prueba una función que obtenga el valor de una cookie pasando su nombre como parámetro.

## Saber si las cookies están habilitadas o no.

Un usuario puede eliminar las cookies de su navegador. También puede desactivar las cookies. Por ello, puede ser interesante saber si están activadas, se sabe leyendo el objeto `navigator` y la propiedad booleana `cookieEnabled`.

Ejercicio: Realiza una página web que muestre un mensaje en el documento indicando si las cookies están o no activadas. Prueba la función activando y desactivando las cookies de tu navegador.

## Ejercicio: HacerAlgoUnaSolaVez

```
/*
Para usar el siguiente código: reemplaza todas las veces la
palabra hacerAlgoUnaSolaVez (el nombre de la cookie) con un nombre personalizado.
*/
function hazUnaVez() {
    if (document.cookie.replace(/(?:^(?:\s*)hacerAlgoUnaSolaVez\s*\=\s*([^;]*).*$|^\.*$/g, "$1") != "true") {
        alert("Hacer algo aquí!");
        document.cookie = "hacerAlgoUnaSolaVez=true; expires=Fri, 31 Dec 9999 23:59:59 GMT";
    }
}

<button onclick="hazUnaVez()">Solo hacer algo una vez</button>
```



## DWEC – Javascript Web Cliente.

JavaScript – Almacenamiento web HTML.....	1
Introducción.....	1
Objetos de almacenamiento web HTML.....	1
Métodos y propiedades disponibles: .....	2
El objeto de almacenamiento local (localStorage).....	2
Almacenar y recuperar información .....	2
Eliminar información.....	3
Eliminar todos los datos .....	3
Ejemplo: .....	3
El objeto de almacenamiento (sessionStorage).....	3
Ejemplo: .....	4
Persistencia de la información almacenada.....	4
El evento storage .....	4
Diferencias entre cookies y storage .....	4
Ejemplo: Almacenar un array de objetos con JSON y recuperarlo.....	5
Usando Object.create() .....	8
Obteniendo el array de objetos de la clase Persona: .....	8

# JavaScript – Almacenamiento web HTML

## Introducción

Con el almacenamiento web HTML las aplicaciones web pueden almacenar datos localmente dentro del navegador del usuario.

Antes de HTML5, los datos de la aplicación tenían que almacenarse en cookies.

El almacenamiento web es más seguro y se pueden almacenar grandes cantidades de datos localmente, sin afectar el rendimiento del sitio web.

A diferencia de las cookies, el límite de almacenamiento es mucho mayor (al menos 5 MB), y la información nunca se transfiere al servidor.

El almacenamiento web queda definido por su origen (por su dominio y el protocolo utilizado). Si el usuario cambia de página, el almacén de datos es distinto.

## Objetos de almacenamiento web HTML

El almacenamiento web HTML proporciona dos objetos para almacenar datos en el cliente:

- `window.localStorage`: almacena datos sin fecha de caducidad.

- `window.sessionStorage`: almacena datos para una sesión (los datos se pierden cuando se cierra la pestaña del navegador).

Antes de usar el almacenamiento web es aconsejable comprobar la compatibilidad del navegador con `localStorage` y `sessionStorage`:

```
if (typeof(Storage) !== "undefined") {
    // Code for LocalStorage/sessionStorage.
} else {
    // Sorry! No Web Storage support..
}
```

## Métodos y propiedades disponibles:

En la siguiente tabla se describen los métodos y propiedades para el objeto `window.sessionStorage`. El objeto `window.localStorage` utiliza los mismos.

Método o propiedad de sessionStorage	Descripción
<code>sessionStorage.setItem('clave', 'valor');</code>	Guarda la información <b>valor</b> a la que se podrá acceder invocando a <b>clave</b> . Por ejemplo, <i>clave</i> puede ser nombre y <i>valor</i> puede ser Carlos.
<code>sessionStorage.getItem('clave')</code>	Recupera el <b>value</b> de la clave especificada. Por ejemplo, si <i>clave</i> es nombre puede recuperar “Carlos”.
<code>sessionStorage[clave]=valor</code>	Igual que <code>setItem</code>
<code>sessionStorage.length</code>	Devuelve el número de items guardados por el objeto <code>sessionStorage</code> actual.
<code>sessionStorage.key(i)</code>	Cada item se almacena con un índice que comienza por cero y se incrementa unitariamente por cada item añadido. Con esta sintaxis rescatamos la clave correspondiente al item con índice i.
<code>sessionStorage.removeItem(clave)</code>	Elimina un item almacenado en <code>sessionStorage</code>
<code>sessionStorage.clear()</code>	Elimina todos los items almacenados en <code>sessionStorage</code> , quedando vacío el espacio de almacenamiento.

## El objeto de almacenamiento local (`localStorage`)

El objeto `localStorage` almacena los datos sin fecha de caducidad. Los datos no se eliminarán cuando se cierre el navegador y estarán disponibles al día, semana o año siguiente.

### Almacenar y recuperar información

Se utilizan los métodos `localStorage.setItem()` y `localStorage.getItem()`

Los pares de nombre/valor siempre se almacenan como cadenas. Hay que convertirlos a otro formato cuando sea necesario.

Ejemplo para:

- Crear un par de nombre/valor de almacenamiento local con nombre="apellido" y valor="Smith"
- Recuperar el valor de "apellido" e insértelo en el elemento con id="resultado"

```
// Store
// Crear un par de nombre/valor de almacenamiento local con nombre="apellido" y valor="Peláez"
localStorage.setItem("apellido", "Peláez");

// Retrieve
// Recuperar el valor de "apellido" e insertarlo en el elemento con id="resultado"
document.getElementById("resultado").innerHTML = localStorage.getItem("apellido");
```

El ejemplo anterior también podría escribirse así:

```
// Store
localStorage.apellido = "Peláez";
// Retrieve
document.getElementById("resultado").innerHTML = localStorage.apellido;
```

## Eliminar información

Se utiliza el método `localStorage.removeItem()`. La sintaxis para eliminar el elemento `localStorage "apellido"` es la siguiente:

```
localStorage.removeItem("apellido");
```

## Eliminar todos los datos

Para eliminar todos los datos y dejar limpio el almacenamiento local de nuestro dominio y protocolo (origen):

```
localStorage.clear();
```

## Ejemplo:

Ejemplo que cuenta el número de veces que un usuario ha hecho clic en un botón. En este código, la cadena de valor se convierte a Number para poder incrementar el contador:

```
if (localStorage.clickcount) {
  localStorage.clickcount = Number(localStorage.clickcount) + 1;
} else {
  localStorage.clickcount = 1;
}

document.getElementById("resultado").innerHTML = "You have clicked the button " +
localStorage.clickcount + " time(s).";
```

## El objeto de almacenamiento (sessionStorage)

El objeto `sessionStorage` es igual al objeto `localStorage`, excepto que almacena los datos para una sola sesión. Los datos se eliminan cuando el usuario cierra la pestaña específica del navegador.

Se utilizan los mismos métodos que en el objeto localStorage.

### Ejemplo:

El siguiente ejemplo cuenta la cantidad de veces que un usuario ha hecho clic en un botón en la sesión actual:

```
if (sessionStorage.clickcount) {
    sessionStorage.clickcount = Number(sessionStorage.clickcount) + 1;
} else {
    sessionStorage.clickcount = 1;
}
document.getElementById("resultado").innerHTML = "You have clicked the button " +
sessionStorage.clickcount + " time(s) in this session.";
```

Ejercicio: escribe el código para guardar automáticamente el contenido de un campo de texto y, si se actualiza el navegador, restaurar el contenido del campo de texto para que no se pierda lo que ya tiene escrito.

## Persistencia de la información almacenada.

Hay que tener en cuenta que, si un usuario realiza una limpieza de la caché del navegador, hará que se borren los datos almacenados con localStorage.

Si el usuario no limpia la caché, los datos se mantendrán durante mucho tiempo. En cambio, hay usuarios que tienen configurado el navegador para que la caché se limpie en cada ocasión en que cierran el navegador. En este caso la persistencia que ofrece localStorage es similar a la que ofrece sessionStorage.

No podemos confiar el funcionamiento de una aplicación web a que el usuario limpie o no limpie la caché, por tanto, deberemos seguir trabajando con datos del lado del servidor siempre que deseemos obtener una persistencia de duración indefinida.

### El evento storage

localStorage permite que se reconozcan datos desde distintas ventanas.

Para detectar que en una ventana que se ha producido un cambio en los datos se definió el evento **storage**: este evento se dispara cuando tiene lugar un cambio en el espacio de almacenamiento y puede ser detectado por las distintas ventanas que estén abiertas.

Para crear una respuesta a este evento podemos escribir:

```
window.addEventListener("storage", nombreFuncionRespuesta, false);
```

Donde nombreFuncionRespuesta es el nombre de la función que se invocará cuando se produzca el evento.

## Diferencias entre cookies y storage

Los objetos storage juegan un papel similar a las cookies, pero por otro lado hay diferencias importantes:

- Las **cookies** están disponibles tanto en el servidor como en el navegador del usuario. Los **objetos storage** sólo están disponibles en el navegador del usuario.

- Las **cookies** se concibieron como **pequeños paquetes de identificación**, con una **capacidad limitada** (unos 4 Kb). Los **objetos storage** se han concebido para **almacenar datos a mayor escala** (pudiendo comprender cientos o miles de datos con un espacio de almacenamiento de varios Mb).

Hay que tener en cuenta que, de una forma u otra, **ni las cookies ni los objetos storage** están pensados para el almacenamiento de grandes volúmenes de información, sino para la gestión de los flujos de datos propios de la navegación web.

## Ejemplo: Almacenar un array de objetos con JSON y recuperarlo

*Archivos index.html y clasePersona.js completos:*

Tenemos un archivo index.html y otro con la clase Persona

<pre>&lt;!DOCTYPE html&gt; &lt;html lang="es"&gt; &lt;head&gt;   &lt;meta charset="UTF-8"&gt;   &lt;title&gt;Document&lt;/title&gt; &lt;/head&gt; &lt;body&gt;   &lt;script type ="module" src="js/app.js"&gt;&lt;/script&gt; &lt;/body&gt; &lt;/html&gt;</pre>	<pre>export class Persona{   constructor(nombre, apellido){     this._nombre = nombre;     this._apellido = apellido;   }   get nombre(){     return this._nombre;   }   set nombre(nombre){     this._nombre = nombre;   }   get apellido(){     return this._apellido;   }   set apellido(apellido){     return this._apellido = apellido;   } }</pre>

Se trata de crear objetos de la clase Persona, guardarlos como cadena y recuperarlos como array.

Vamos a utilizar los métodos

- JSON.stringify()
- JSON.parse()

```
//Guardar el objeto personas en localStorage
localStorage.setItem('personasCadena', JSON.stringify(personas));
```

```
// recuperar de localStorage en crudo (texto sin más)
let personasAlmacenadas = localStorage.getItem('personasCadena');
```

```
// recuperar de localStorage parseando (obteniendo el array de objetos)
// let personas2 = JSON.parse(localStorage.getItem('personasCadena'));
```

```
let personas2=JSON.parse(personasAlmacenadas);
```

Archivo app.js completo:

```
import {Persona} from './clases/clasePersona.js';

// (1) personas: es un array de objetos de la clase persona
// (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
// (3) personas2: es un array de objetos que conseguimos al parsear personasAlmacenadas

let personas=[
    new Persona('Juan', 'Pérez'),
    new Persona('Ana', 'González'),
    new Persona('Faustino', 'Sigüenza')
];

// Tenemos 3 personas y añadimos una cuarta
personas.push(new Persona('Andrés', 'Hernández'));
console.log('personas: ', personas);

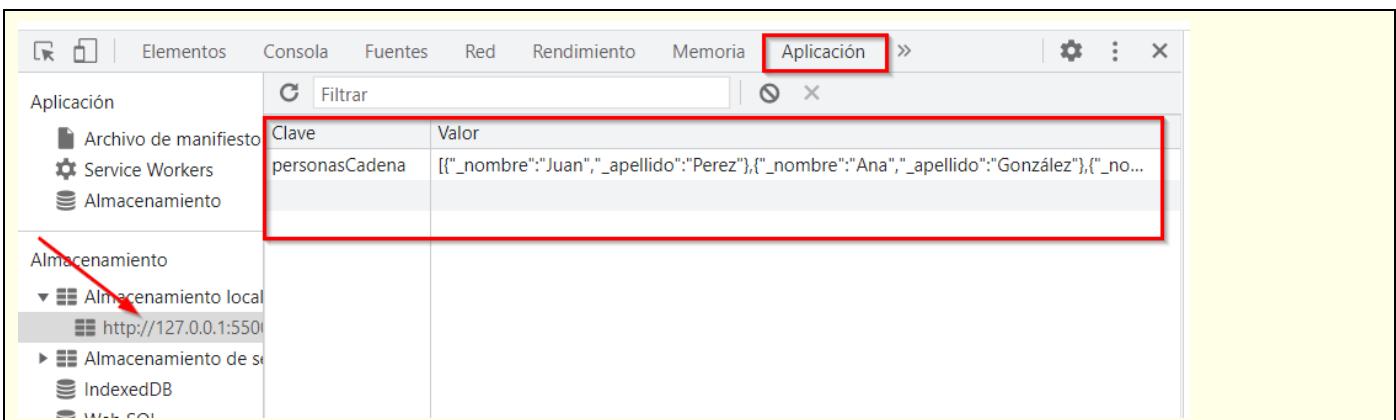
// Guardamos el objeto personas en localStorage
localStorage.setItem('personasCadena', JSON.stringify(personas));

// Recuperamos de localStorage en crudo, como texto
let personasAlmacenadas = localStorage.getItem('personasCadena');
console.log("personasAlmacenadas: ", personasAlmacenadas);

// Recuperamos de localStorage parseando
//let personas2 = JSON.parse(localStorage.getItem('personasCadena'))
let personas2 = JSON.parse(personasAlmacenadas);
console.log('personas2: ', personas2);
```

Inspeccionando el almacenamiento en localStorage:

Se puede visualizar, editar y eliminar el contenido de localStorage inspeccionando el documento:



The screenshot shows the Chrome DevTools interface with the 'Aplicación' (Application) tab selected. In the left sidebar, under 'Almacenamiento', the 'LocalStorage' section is expanded, showing an entry for the key 'personasCadena'. The value is a JSON string representing an array of objects. A red box highlights this entry.

Clave	Valor
personasCadena	[{"_nombre": "Juan", "_apellido": "Perez"}, {"_nombre": "Ana", "_apellido": "González"}, {"_n..."]

### Mirando en la consola:

```

js > JS prueba.js > ...
1 // (1) personas: es un Array de objetos de la clase Persona
2 // (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
3 // (3) personas2: es un Array de objetos que conseguimos al parsear personasAlmacenadas
4 let personas=[
5     new Persona('Juan', 'Perez'),
6     new Persona('Ana', 'González'),
7     new Persona('Faustino', 'Sigüenza')
8 ];
9 // tenemos 3 personas y añadimos una cuarta persona
10 personas.push(new Persona('Andrés', 'Hernández'));
11 console.log('personas: ', personas);
12
13 //Guardar el objeto personas en localStorage
14 localStorage.setItem('personasCadena',JSON.stringify(personas));
15
16 // recuperar de localStorage en crudo
17 let personasAlmacenadas = localStorage.getItem('personasCadena');
18 console.log("personasAlmacenadas:", personasAlmacenadas);
19
20 // recuperar de localStorage parseando
21 //let personas2 = JSON.parse(localStorage.getItem('personasCadena'));
22 let personas2=JSON.parse(personasAlmacenadas);
23 console.log('personas2:', personas2);

```

The screenshot shows the browser's developer tools with the "Consola" tab selected. The code is run in the console, and the output is displayed in three sections. Section 1 shows the original array of Person objects. Section 2 shows the string representation of the array stored in localStorage. Section 3 shows the array of Person objects reconstructed from the stored string.

### Problemita:

Desplegando los arrays en la consola observamos:

```

js > JS prueba.js > ...
1 // (1) personas: es un Array de objetos de la clase Persona
2 // (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
3 // (3) personas2: es un Array de objetos que conseguimos al parsear personasAlmacenadas
4 let personas=[
5     new Persona('Juan', 'Perez'),
6     new Persona('Ana', 'González'),
7     new Persona('Faustino', 'Sigüenza')
8 ];
9 // tenemos 3 personas y añadimos una cuarta persona
10 personas.push(new Persona('Andrés', 'Hernández'));
11 console.log('personas: ', personas);
12
13 //Guardar el objeto personas en localStorage
14 localStorage.setItem('personasCadena',JSON.stringify(personas));
15
16 // recuperar de localStorage en crudo
17 let personasAlmacenadas = localStorage.getItem('personasCadena');
18 console.log("personasAlmacenadas:", personasAlmacenadas);
19
20 // recuperar de localStorage parseando
21 //let personas2 = JSON.parse(localStorage.getItem('personasCadena'));
22 let personas2=JSON.parse(personasAlmacenadas);
23 console.log('personas2:', personas2);

```

The screenshot shows the browser's developer tools with the "Consola" tab selected. The code is run in the console, and the output is displayed in three sections. Section 1 shows the original array of Person objects. Section 2 shows the string representation of the array stored in localStorage. Section 3 shows the array of Person objects reconstructed from the stored string. Red arrows point from the array numbers in the code to their corresponding expanded forms in the console, highlighting the difference between the original objects and the parsed objects.

Tenemos un pequeño problema:

- Guardamos un array de objetos de la clase Persona (el llamado **personas**)
- Recuperamos un array de objetos sin clase (al que hemos llamado **personas2**)

Los datos son los mismos, pero el segundo array (**personas2**) no puede utilizar los métodos de la clase.

```

24 //mostrando el atributo _nombre de personas con getNombre()
25 personas.forEach(element => {
26   console.log(element.getNombre());
27 });
28 );
29
30 //mostrando el atributo _nombre de personas2 con etNombre()
31 personas2.forEach(element => {
32   console.log(element.getNombre());
33 });
34
35 //mostrando el atributo _nombre de personas directamente
36 personas.forEach(element => {
37   console.log(element._nombre);
38 });
39
40 //mostrando el atributo _nombre de personas2 directamente
41 personas2.forEach(element => {
42   console.log(element._nombre);
43 });
44

```

En el ejemplo anterior no se pueden utilizar los métodos `get()` para obtener los valores de los atributos de los elementos en `personas2`, debemos obtener los atributos directamente.

## Usando `Object.create()`

Para resolver el caso anterior, podemos utilizar `Object.create(objetoModelo)` para añadir a un array `personas3` los objetos de la clase Persona que hemos recuperado de `localStorage`.

A continuación, ponemos un ejemplo:

```

js > prueba2.js > ...
17 const personaModelo= new Persona(); 1
18
19 const nuevaPersona = Object.create(personaModelo); 2
20 nuevaPersona._nombre='Pedro'; 3
21 nuevaPersona._apellido='Gómez'; 4
22
23 console.log(nuevaPersona.getNombre()); 5
24
25 const otraPersona = Object.create(personaModelo); 6
26 otraPersona._nombre='Carmen'; 7
27 otraPersona._apellido='Sevilla'; 8
28
29 personas.push(nuevaPersona);
30 personas.push(otraPersona);
31 console.log(personas);

```

- (1) Creamos un objeto modelo de la clase que queramos tener. No hace falta que tenga datos.
- (2) Creamos un nuevo objeto utilizando el objeto modelo existente.
- (3) (6) Asignamos contenido a los atributos.
- (4) Ya podemos utilizar los métodos de la clase.
- (5) Podemos crear más objetos de la clase.
- (7) Metemos los objetos de la clase Persona en el array personas
- (8) Comprobamos el contenido del array.

## Obteniendo el array de objetos de la clase Persona:

Se va a `resolver el problemilla`:

*Archivo app.js resultante:*

```
import {Persona} from './clases/clasePersona.js';
```

```

// (1) personas: es un array de objetos de la clase persona
// (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
// (3) personas2: es un array de objetos que conseguimos al parsear personasAlmacenadas

let personas=[  

    new Persona('Juan', 'Pérez'),  

    new Persona('Ana', 'González'),  

    new Persona('Faustino', 'Sigüenza')  

];  
  

// Tenemos 3 personas y añadimos una cuarta  

personas.push(new Persona('Andrés', 'Hernández'));  

console.log('personas: ', personas);  
  

// Guardamos el objeto personas en localStorage  

localStorage.setItem('personasCadena', JSON.stringify(personas));  
  

// Recuperamos de localStorage en crudo, como texto  

let personasAlmacenadas = localStorage.getItem('personasCadena');  

console.log("personasAlmacenadas: ", personasAlmacenadas);  
  

// Recuperamos de localStorage parseando  

//let personas2 = JSON.parse(localStorage.getItem('personasCadena'))  

let personas2 = JSON.parse(personasAlmacenadas);  

console.log('personas2: ', personas2);  
  

// Recuperar el array de objetos de la clase Persona:  

// const personaModelo = new Persona(); // (**)
const personas3=[];  
  

personas2.forEach((p) =>{  

    // let nuevaPersona= Object.create(personaModelo); // (**)  

    let nuevaPersona=new Persona() // (**)  

    nuevaPersona.nombre = p._nombre;  

    nuevaPersona.apellido = p.apellido;  

    console.log(nuevaPersona);  

    personas3.push(nuevaPersona);  

})  
  

console.log('personas3: ',personas3);

```

(\*\*) Se puede optar por:

- a) Utilizar personaModelo como modelo de nuevaPersona, o
- b) Crear nuevaPersona como una instancia de la clase Persona.



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 1 .....	1
Introducción.....	1
Métodos HTTP.....	3
Formato JSON .....	4
Json Server .....	7
REST client.....	10
Thundert Client para VSC y json-server.....	13

# JavaScript - Ajax 1

## Introducción

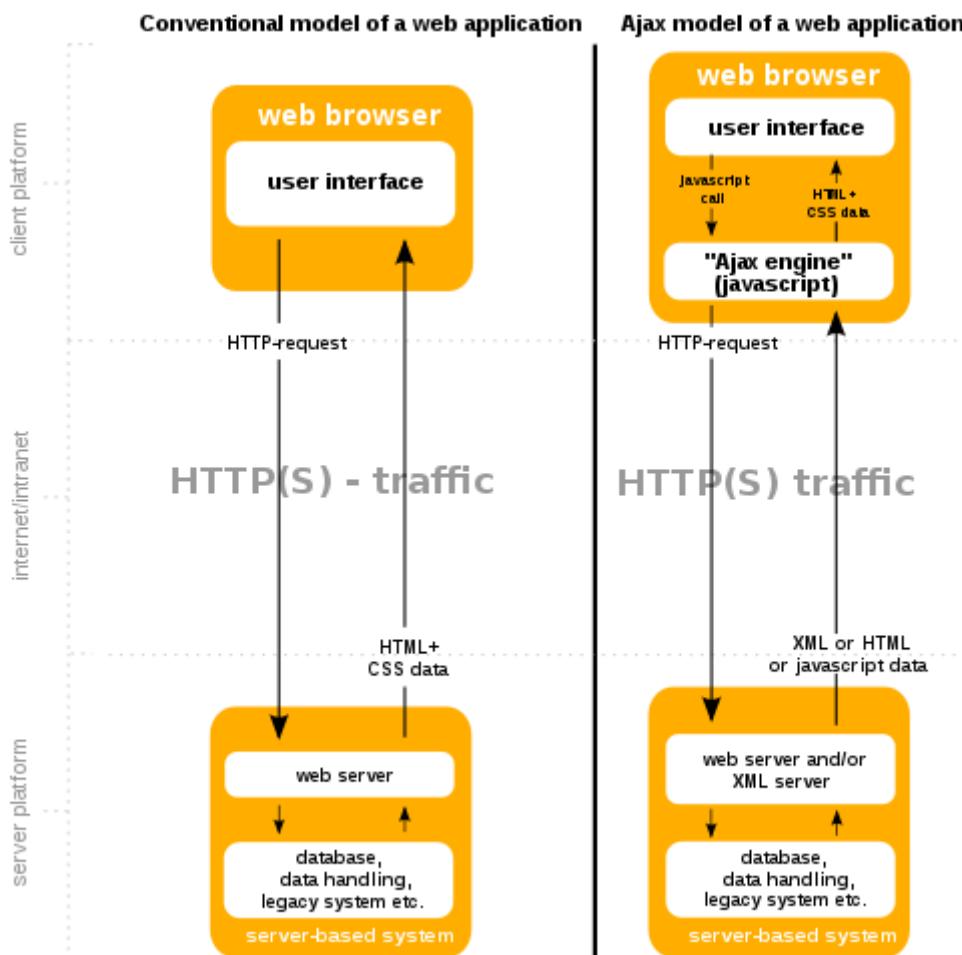
AJAX es el acrónimo de **Asynchronous Javascript And XML** (Javascript asíncrono y XML) y es lo que usamos para hacer peticiones asíncronas al servidor desde Javascript.

Cuando hacemos una petición al servidor no nos responde inmediatamente (la petición tiene que llegar al servidor, procesarse allí y enviarse la respuesta que llegará al cliente).

Lo que significa **asíncrono** es que la página (cliente) no permanecerá bloqueada esperando esa respuesta, sino que continuará ejecutando su código e interactuando con el usuario y, en el momento en que llegue la respuesta del servidor se ejecutará la función que habíamos indicado al hacer la llamada Ajax.

Respecto a **XML**: es el formato en que se intercambia la información entre el servidor y el cliente, aunque actualmente el formato más usado es **JSON** que es más simple y legible. Podríamos decir que hoy no se usa Ajax y sí Ajaj, pero suena a que te pica la garganta, así que seguiremos diciendo Ajax aunque usemos JSON.

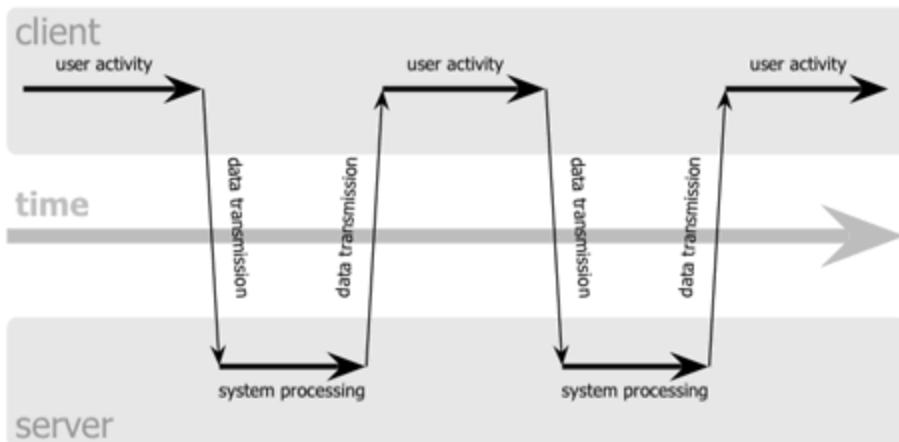
Básicamente Ajax nos permite poder mostrar nuevos datos enviados por el servidor sin tener que recargar la página, que continuará disponible mientras se reciben y procesan los datos enviados por el servidor en segundo plano.



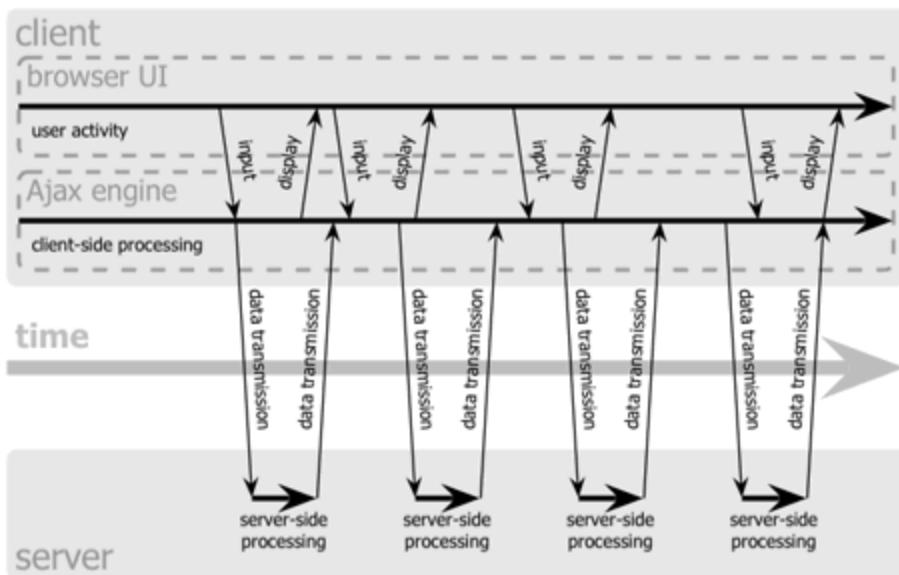
Sin Ajax, cada vez que necesitamos nuevos datos del servidor la página deja de estar disponible para el usuario hasta que se recarga con lo que envía el servidor.

Con Ajax la página está siempre disponible para el usuario y simplemente se modifica (cambiando el DOM) cuando llegan los datos del servidor:

## classic web application model (synchronous)



## Ajax web application model (asynchronous)



## Métodos HTTP

Las peticiones Ajax usan el protocolo **HTTP** (el mismo que utiliza el navegador para cargar una página).

El protocolo **HTTP** envía al servidor:

- Unas cabeceras **HTTP** con información como el *userAgent* del navegador, el idioma, etc.
- El tipo de petición.**
- Opcionalmente**, **datos o parámetros**. Por ejemplo, en la petición que procesa un formulario se envían los datos del mismo.

Hay diferentes tipos de petición que podemos hacer:

- GET**: suele usarse para **obtener datos sin modificar nada** (equivale a un **SELECT** en SQL). **Si enviamos datos** (ej. el ID del registro a obtener) **suelen ir en la url de la petición** (formato **URIEncoded**). Ej.: [localhost/users/3](http://localhost/users/3), <https://jsonplaceholder.typicode.com/users> o [www.google.es?search=js](http://www.google.es?search=js)
- POST**: suele usarse para **añadir un dato en el servidor** (equivalente a un **INSERT**). **Los datos enviados van en el cuerpo de la petición HTTP** (igual que sucede al enviar desde el navegador un formulario por POST)
- PUT**: es similar al **POST**, pero suele usarse para **actualizar datos del servidor** (como un **UPDATE** de SQL). Los datos se envían en el **cuerpo de la petición** (como en el **POST**). **La información para identificar el objeto**

a modificar va en la url (como en el GET). El servidor hará un UPDATE sustituyendo el objeto actual por el que se le pasa como parámetro.

- **PATCH**: es similar al PUT, pero la diferencia es que en el PUT hay que pasar todos los campos del objeto a modificar (los campos no pasados se eliminan del objeto), mientras que en el PATCH sólo se pasan los campos que se quieren cambiar y el resto permanecen como están.
- **DELETE**: se usa para eliminar un dato del servidor (como un DELETE de SQL). La información para identificar el objeto a eliminar se envía en la url (como en el GET).
- Existen otros tipos de peticiones que no trataremos en esta documentación, que son:
  - **HEAD**: pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
  - **CONNECT**: establece un túnel hacia el servidor identificado por el recurso.
  - **OPTIONS**: es utilizado para describir las opciones de comunicación para el recurso de destino.
  - **TRACE**: realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.

El servidor acepta la petición, la procesa y envía al cliente una respuesta que consiste en:

- a) El recurso solicitado.
- b) Adjunta unas cabeceras de respuesta que incluyen: el tipo de contenido enviado, el idioma, etc.
- c) El código de estado.

Los códigos de estado más comunes son:

- 2xx: son peticiones procesadas correctamente. Las más usuales son 200 (ok) o 201 (created, como respuesta a una petición POST satisfactoria).
- 3xx: son códigos de redirección que indican que la petición se redirecciona a otro recurso del servidor, como 301 (el recurso se ha movido permanentemente a otra URL) o 304 (el recurso no ha cambiado desde la última petición por lo que se puede recuperar desde la caché).
- 4xx: indican un error por parte del cliente, como 404 (Not found, no existe el recurso solicitado) o 401 (Not authorized, el cliente no está autorizado a acceder al recurso solicitado).
- 5xx: indican un error por parte del servidor. Ejemplos: 500 (error interno del servidor) o 504 (timeout, el servidor no responde).

En cuanto a la información enviada por el servidor al cliente, normalmente serán datos en formato **JSON** o **XML** (cada vez menos usado) que el cliente procesará y mostrará en la página al usuario. También podría ser HTML, texto plano, ...

## Formato JSON

El formato JSON (acrónimo de JavaScript Object Notation, 'notación de objeto de JavaScript') es una forma de convertir objetos Javascript en una cadena de texto para poderlos enviar.

La estructura de datos JSON se compone de un conjunto de objetos o arrays que contendrán números, cadenas booleanos y nulos.

Un **objeto** JSON comienza y termina con llaves, y contiene una colección desordenada de pares **nombre**-**valor**.

Cada **nombre** y **valor** están separados por dos puntos, y los pares están separados por comas.

La coma final está prohibida.

El **nombre** es una cadena entre comillas dobles. Los caracteres de comillas no deben ser inclinadas o "inteligentes".

En los **números**, los ceros a la izquierda están prohibidos; un punto decimal debe estar seguido al menos por un dígito.

En las cadenas deben estar entre comillas dobles. No se permiten todos los caracteres de escape; sí se permiten los caracteres de separador de línea Unicode (U+2028) y el separador de párrafo (U+2029).

Un array JSON comienza y termina con corchetes y contiene una colección ordenada de valores separados por comas. Un valor puede ser una cadena entre comillas dobles, un número, un booleano true o false, nulo, un objeto JSON o un array.

Los objetos y los arrays JSON se pueden anidar, lo que posibilita una estructura jerárquica de datos.

En el siguiente ejemplo, se muestra una estructura de datos JSON con dos objetos válidos.

```
{
  "id": 1006410,
  "title": "Amazon Redshift Database Developer Guide"
}
{
  "id": 100540,
  "name": "Amazon Simple Storage Service User Guide"
}
```

En el siguiente se muestran los mismos datos como dos arrays JSON:

```
[
  1006410,
  "Amazon Redshift Database Developer Guide"
]
[
  100540,
  "Amazon Simple Storage Service User Guide"
]
```

### Conversiones de objetos

El objeto alumno:

```
let alumno = {
  id: 5,
  nombre: 'Ana',
  apellidos: 'Zubiri Peláez'
}
```

se transformaría en la cadena de texto:

```
{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }
```

y el array:

```
let alumnos = [
  {
    id: 5,
    nombre: "Ana",
    apellidos: "Zubiri Peláez"
  },
  {
    id: 6,
    nombre: "Juan",
    apellidos: "Gómez"
  }
]
```

```
{
  id: 7,
  nombre: "Carlos",
  apellidos: "Pérez Ortíz"
},
]
```

en la cadena:

```
[{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }, { "id": 7, "nombre": "Carlos", "apellidos": "Pérez Ortíz" }]
```

Nótese que tanto las claves como los valores van entrecomillados (con comillas dobles). No sirven comillas simples.

### Estructura de los datos

Los mismos datos pueden tener distinta estructura. Ejemplo:

Archivo colores1.json	Archivo colores2.json	Archivo colores3.json
{ "arrayColores": [ { "nombreColor": "rojo", "valorHexadec": "#f00" }, { "nombreColor": "verde", "valorHexadec": "#0f0" }, { "nombreColor": "azul", "valorHexadec": "#00f" }, { "nombreColor": "cyan", "valorHexadec": "#0ff" }, { "nombreColor": "magenta", "valorHexadec": "#f0f" }, { "nombreColor": "amarillo", "valorHexadec": "#ff0" }, { "nombreColor": "negro", "valorHexadec": "#000" } ]	{ "arrayColores": [ { "rojo": "#f00", "verde": "#0f0", "azul": "#00f", "cyan": "#0ff", "magenta": "#f0f", "amarillo": "#ff0", "negro": "#000" }] }	{ "rojo": "#f00", "verde": "#0f0", "azul": "#00f", "cyan": "#0ff", "magenta": "#f0f", "amarillo": "#ff0", "negro": "#000" }

Los ejemplos anteriores representan lo que podrían ser archivos JSON conteniendo datos en formato JSON.

Se trata de 3 archivos que contienen aproximadamente la misma información. Sin embargo, hay algunas diferencias:

- En el archivo **colores1.json** existe un único objeto de datos donde el nombre es *arrayColores* y su valor es un array de objetos JSON. Cada objeto del array está formado por los pares (*nombreColor* y su valor), y (*valorHexadec* y su valor). En este ejemplo en concreto el array consta de 7 elementos con información correspondiente a 7 colores.
- En el archivo **colores2.json** existe un único objeto de datos donde el nombre es *arrayColores*, cuyo valor es un array que contiene un único objeto JSON formado por siete pares (nombre – valor) que representa información sobre siete colores.
- En el archivo **colores3.json** existe un único objeto de datos que está formado por siete pares (nombre – valor) que representa información sobre siete colores.

Siendo las 3 formas válidas, se deberá utilizar aquella que se indique en las instrucciones o, de no existir pautas precisas, utilizar aquel diseño que favorezca el desarrollo y mantenimiento de la aplicación.

### Métodos del objeto JSON

Para convertir objetos en cadenas de texto JSON y viceversa, Javascript proporciona 2 métodos:

- **JSON.stringify(objeto)**: recibe un objeto JS y devuelve la cadena de texto correspondiente.

```
//convierte de formato JSON a objeto
const cadenaAlumnos = JSON.stringify(alumnos)
```

- **JSON.parse(cadena)**: realiza el proceso inverso, convirtiendo una cadena de texto en un objeto.

```
//convierte un objeto a formato JSON
const alumnos = JSON.parse(cadenaAlumnos)
```

## Json Server

Las peticiones Ajax se hacen a un servidor que proporcione una API.

Se puede utilizar **Json Server** que es un servidor API-REST que funciona bajo Node.js

(nota para mí: me ha fallado Node ver.18, instalé versión 16 y bien)

Si aún no está instalado Node.js, se puede hacer desde <https://nodejs.org/es/>

Node.js utiliza un fichero JSON como contenedor de los datos en lugar de una base de datos.

Para instalar json-server en nuestra máquina, lo instalaremos de forma global para poderlo utilizar en otros ejercicios, desde cualquier terminal hay que ejecutar **npm** (Node Package Manager) de la siguiente forma:

```
npm install -g json-server
```

Para que sirva (provea de este servicio los datos de) un fichero datos.json, se ejecuta la sentencia:

```
json-server datos.json
```

Se puede poner la opción **--watch** ( o **-w**) para que actualice los datos si se modifica el fichero **.json** externamente (si lo editamos).

El fichero **datos.json** será un fichero que contenga un objeto JSON con una propiedad para cada “*tabla*” de nuestra BBDD.

Por ejemplo, si queremos simular una BBDD con las tablas *users* y *posts* vacías, el contenido del fichero será:

```
{
  "users": [],
  "posts": []
}
```

Otro ejemplo de un fichero json con 2 tablas: películas y clasificaciones

Películas y clasificaciones son dos entidades distintas.

```
{
```

```
"peliculas": [
  {
    "id": 1,
    "nombre": "El sexto sentido",
    "director": "M. Night Shyamalan",
    "clasificacion": "Drama"
  },
  {
    "id": 2,
    "nombre": "Pulp Fiction",
    "director": "Tarantino",
    "clasificacion": "Acción"
  },
  {
    "id": 3,
    "nombre": "Todo Sobre Mi Madre",
    "director": "Almodobar",
    "clasificacion": "Drama"
  },
  {
    "id": 4,
    "nombre": "300",
    "director": "Zack Snyder",
    "clasificacion": "Acción"
  },
  {
    "id": 5,
    "nombre": "El silencio de los corderos",
    "director": "Jonathan Demme",
    "clasificacion": "Drama"
  },
  {
    "id": 6,
    "nombre": "Forrest Gump",
    "director": "Robert Zemeckis",
    "clasificacion": "Comedia"
  },
  {
    "id": 7,
    "nombre": "Las Hurdes",
    "director": "Luis Buñuel",
    "clasificacion": "Documental"
  }
],
"clasificaciones": [
  {
    "nombre": "Drama",
    "id": 1
  },
  {
    "nombre": "Comedia",
    "id": 2
  }
]
```

```

},
{
  "nombre": "Acción",
  "id": 4
},
{
  "nombre": "Terrorífica",
  "id": 15
}
]
}

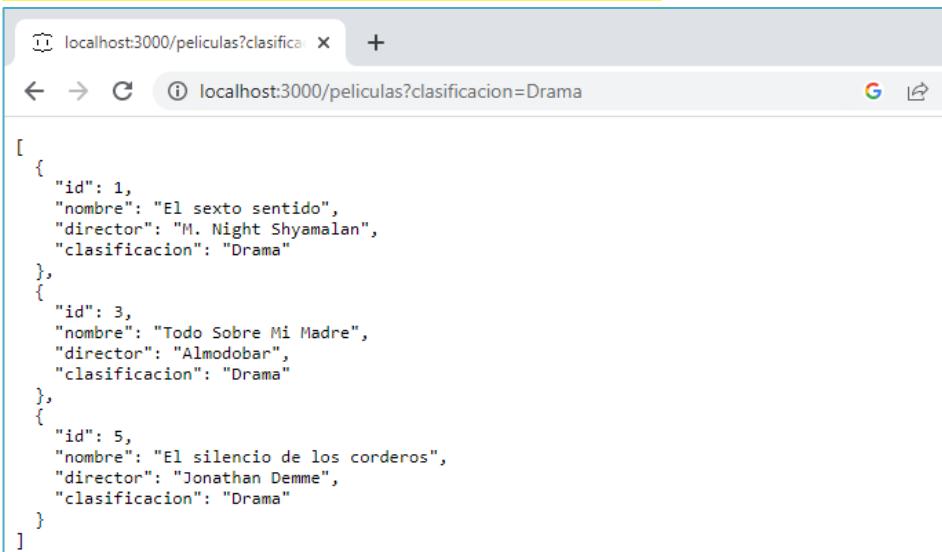
```

Al iniciar el servicio, por defecto la API escucha en el puerto 3000 y servirá los diferentes objetos definidos en el fichero `json`. Por ejemplo:

- `http://localhost:3000/peliculas/`: devuelve un array con todos los elementos de la tabla `peliculas` del fichero `json`
- `http://localhost:3000/peliculas/5`: devuelve un objeto con el elemento de la tabla `peliculas` cuya propiedad `id` valga 5

También pueden hacerse peticiones más complejas como:

- `http://localhost:3000/peliculas?clasificacion=Drama`: devuelve un array con todos los elementos de `peliculas` cuya propiedad `clasificacion` valga `Drama`. La siguiente imagen muestra el resultado:



The screenshot shows a browser window with the URL `localhost:3000/peliculas?clasificacion=Drama`. The page displays a JSON array of three movie objects, each with properties: id, nombre, director, and clasificacion. All three movies in the array have the value "Drama" for the "clasificacion" property.

```

[
  {
    "id": 1,
    "nombre": "El sexto sentido",
    "director": "M. Night Shyamalan",
    "clasificacion": "Drama"
  },
  {
    "id": 3,
    "nombre": "Todo Sobre Mi Madre",
    "director": "Almodobar",
    "clasificacion": "Drama"
  },
  {
    "id": 5,
    "nombre": "El silencio de los corderos",
    "director": "Jonathan Demme",
    "clasificacion": "Drama"
  }
]

```

Para más información: <https://github.com/typicode/json-server>.

Si queremos acceder a la API desde otro equipo (no desde `localhost`) tenemos que indicar la IP de la máquina que ejecuta `json-server` y que se usará para acceder. Por ejemplo, si vamos a ejecutar el servidor en la máquina 192.168.0.10 pondremos:

```
json-server --host 192.168.0.10 datos.json
```

Si se desea cambiar el puerto por defecto (3000) donde escucha el servidor, a otro (por ejemplo, el 4200):

```
json-server --host 192.168.0.10 -p 4200 datos.json
```

Y la ruta para acceder a la API será `http://192.168.0.10:4200`.

**EJERCICIO:** instalar json-server en tu máquina. Ejecútalo indicando un nombre de fichero que no existe: como verás crea un fichero json de prueba con 3 tablas: *posts*, *comments* y *profiles*. Ábrelo en tu navegador para ver los datos

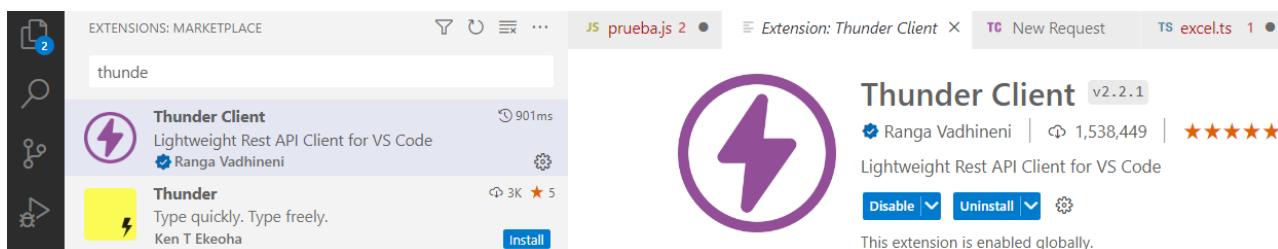
## REST client

Para probar las peticiones GET podemos poner la URL en la barra de direcciones del navegador, pero para probar el resto de peticiones debemos instalar en nuestro navegador una extensión que nos permita realizar las peticiones indicando el método a usar, las cabeceras a enviar y los datos que enviaremos a servidor, además de la URL.

Existen multitud de aplicaciones para realizar peticiones HTTP, como [Advanced REST client](#), [Postman](#), etc.

Además, cada navegador tiene sus propias extensiones para hacer esto, como [Advanced Rest Client](#) para Chrome o [RestClient](#) para Firefox.

También se puede utilizar una extensión para Visual Studio Code llamada *Thunder Client*.

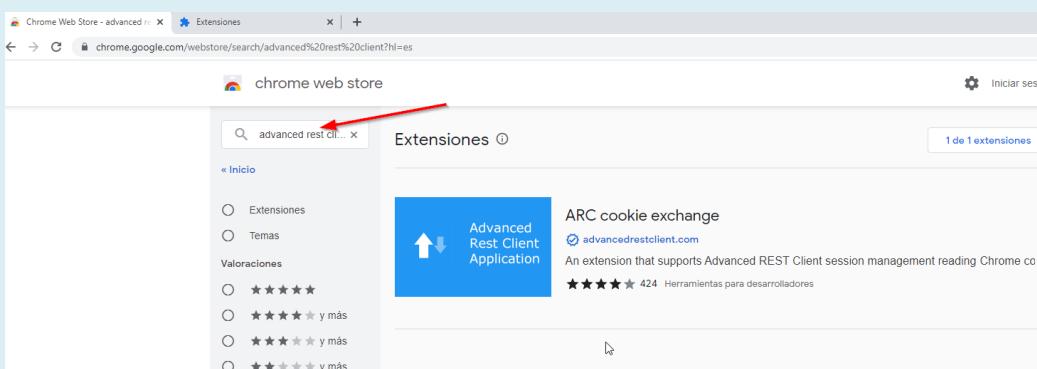


Para realizar el siguiente ejercicio hay que instalar alguna de las extensiones mencionadas y hacer todas las peticiones desde allí (incluyendo los GET), lo que permitirá ver los códigos de estado devueltos, las cabeceras, etc.

**EJERCICIO:** Vamos a realizar diferentes peticiones HTTP a la API <https://jsonplaceholder.typicode.com>. En concreto trabajaremos contra la tabla *todos* (que contiene **tareas** para hacer).

Esta API está disponible para hacer pruebas sin necesidad de montar un servidor.

Para la resolución de este ejercicio se instaló la extensión Advanced Rest Client en Google Chrome:



1º Obtener todas las tareas de la tabla todos. Devolverá un array con todas las tareas y el código devuelto será 200  
– Ok

**Request**

Method: GET Request URL: https://jsonplaceholder.typicode.com/todos

Parameters: **200 OK** 198.66 ms

```
[Array[200]
-0: {
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
},
-1: {
  "userId": 1,
  "id": 2,
  "title": "quis ut nam facilis et officia qui",
  "completed": false
},
-2: {
  "userId": 1,
  "id": 3,
  "title": "fugiat veniam minus",
  "completed": false
},
-3: {
  "userId": 1,
  "id": 4,
  "title": "et porro tempora",
  "completed": true
},
-4: {
  "userId": 1,
  "id": 5,
  "title": "laboriosam ut enim et",
  "completed": false
}, ...]
```

2º Obtener la tarea cuyo id sea 55. Devolverá el objeto de la tarea 55 y el código devuelto será 200 – Ok

**Request**

Method: GET Request URL: https://jsonplaceholder.typicode.com/todos/55

Parameters: **200 OK** 185.43 ms

```
{
  "userId": 3,
  "id": 55,
  "title": "voluptatum omnis minima qui occaecati provident nulla voluptatem ratione",
  "completed": true
}
```

3º Obtener la tarea con id 201. Como no existe, devolverá un objeto vacío y el código de error 404 - Not found.

**Request**

Method: GET Request URL: https://jsonplaceholder.typicode.com/todos/201

Parameters: **404 Not Found** 415.52 ms

```
{}
```

4º Crear una nueva tarea. En el cuerpo de la petición le pasaremos sus datos: *userID*: 1, *title*: Prueba de POST y *completed*: false. No se le pasa la id (de eso se encarga la BBDD, que la creará automática). La respuesta debe ser un código 201 (created) y adjuntará el nuevo registro creado con todos sus datos incluyendo la id. Como es una API de prueba en realidad no lo está añadiendo a la BBDD, por lo que si luego hacemos una petición buscando esa id, nos dirá que no existe.

Advanced REST client

Method: POST Request URL: https://jsonplaceholder.typicode.com/todos

Parameters ^

Headers Body Variables

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "userId": 1,
  "title": "Prueba de POST",
  "completed": false
}
```

201 Created 583.57 ms DETAILS ▾

□ ☰ <> ■■■

```
{
  "userId": 1,
  "title": "Prueba de POST",
  "completed": false,
  "id": 201
}
```

5º Modificar con un PATCH la tarea con id 55 para que su title sea ‘Prueba de POST’. Devolverá el nuevo registro con un código 200. Se observa que al hacer un PATCH los campos que no se pasan se mantienen como estaban.

Advanced REST client

Method: PATCH Request URL: https://jsonplaceholder.typicode.com/todos/55

Parameters ^

Headers Body Variables

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "title": "Prueba de POST"
}
```

200 OK 381.48 ms DETAILS ▾

□ ☰ <> ■■■

```
{
  "userId": 3,
  "id": 55,
  "title": "Prueba de POST",
  "completed": true
}
```

6º Modificar con un PUT la tarea con id 55 para que su title sea ‘Prueba de POST’. Devolverá el nuevo registro con un código 200. Como se aprecia en la imagen, en esta API los campos que no se pasan se eliminan; en otras los campos no pasados se mantienen como estaban.

The screenshot shows the Advanced REST client interface. In the 'Request' tab, a PUT method is selected with the URL <https://jsonplaceholder.typicode.com/todos/55>. The 'Body' tab contains the following JSON:

```
{
  "title": "Prueba de POST"
}
```

The response status is 200 OK with a duration of 417.70 ms. The response body is:

```
{
  "title": "Prueba de POST",
  "id": 55
}
```

7º Eliminar con DELETE la tarea con id 55. Como se ve en la siguiente imagen, esta API devuelve un objeto vacío al eliminar; otras devuelven el objeto eliminado.

The screenshot shows the Advanced REST client interface. In the 'Request' tab, a DELETE method is selected with the URL <https://jsonplaceholder.typicode.com/todos/55>. The 'Body' tab is empty, indicated by a red asterisk (\*).

The response status is 200 OK with a duration of 499.48 ms. The response body is:

```
{}
```

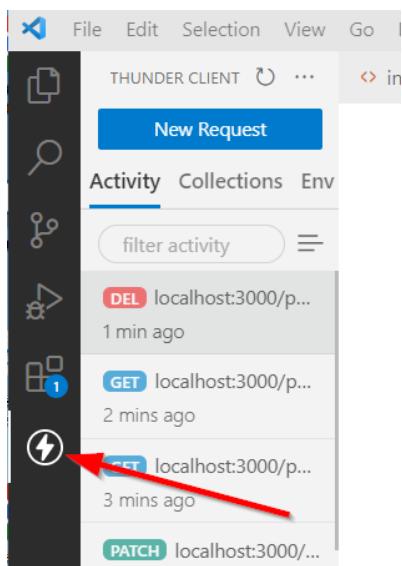
## Thundert Client para VSC y json-server

Se instala como una versión de Visual Studio Code



**Thunder Client** v2.3.2  
 Ranga Vadhineni | ⚡ 1,624,594 | ★★★★★ (154) |  
 Lightweight Rest API Client for VS Code  
 Reload Required | Disable | Uninstall | ⚡  
 This extension is enabled globally.

Una vez instalada nos proporciona en VSC una interfaz cómoda para realizar peticiones.



Para trabajar con json-server, podemos utilizar un terminal de VSC, tanto para instalar json-server:

```
PROBLEMS OUTPUT TERMINAL powershell
PS C:\Users\Santi\DAW2cliente\CURSO_22_23\AJAX\thunder1> npm install json-server
```

como para servir los datos de un fichero json:

```
PS C:\Users\Santi\DAW2cliente\CURSO_22_23\AJAX\POST01> json-server.cmd .\peliculas.json
^__^/ hi!
Loading .\peliculas.json
Done

Resources
http://localhost:3000/peliculas
http://localhost:3000/clasificaciones
```

Si en algún momento en el terminal nos apareciera un mensaje advirtiendo que no se pueden ejecutar scripts, podemos desde el mismo terminal o desde un terminal PowerShell activar la política que deseemos sobre la ejecución de scripts. Se recomienda el modo RemoteSigned.

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\WINDOWS\system32> Get-ExecutionPolicy
Restricted
PS C:\WINDOWS\system32> Set-ExecutionPolicy RemoteSigned
```



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 2 .....	1
Realizar peticiones Ajax .....	1
Eventos de XMLHttpRequest .....	2
Ejemplos de envío de datos .....	4
Enviar datos al servidor en formato JSON.....	5
Enviar datos al servidor en formato URLEncoder .....	5
Enviar ficheros al servidor con FormData .....	6

# JavaScript - Ajax 2

## Realizar peticiones Ajax

Hemos visto lo que es el protocolo HTTP. Ahora que tenemos instalado un servidor que nos proporciona una API (json-server), vamos a realizar peticiones HTTP en nuestro código javascript usando Ajax.

Para hacer una petición debemos crear una instancia del objeto **XMLHttpRequest** que es el que controlará todo el proceso. Los pasos a seguir son:

1. Creamos la instancia del objeto: `const peticion=new XMLHttpRequest()`
2. Para establecer la comunicación con el servidor ejecutamos el método **.open()** al que se le pasa como parámetro el tipo de petición (GET, POST, ...) y la URL del servidor: `peticion.open('GET', 'https://jsonplaceholder.typicode.com/todos')`
3. OPCIONAL: Si queremos añadir cabeceras a la petición HTTP llamaremos al método **.setRequestHeader()**. Por ejemplo, si enviamos datos con POST hay que añadir la cabecera `Content-type` que indica al servidor en qué formato van los datos: `peticion.setRequestHeader('Content-type', 'application/x-www-form-urlencoded')`
4. Enviamos la petición al servidor con el método **.send()**. A este método se le pasa como parámetro los datos a enviar al servidor en el cuerpo de la petición (si es un POST, PUT o PATCH le pasaremos una cadena de texto con los datos a enviar:

```
peticion.send('dato1='+encodeURIComponent(dato1)+'&dato2='+encodeURIComponent(dato2))
```

Si es una petición GET o DELETE no le pasaremos datos:

```
peticion.send()
```

1. Ponemos un escuchador (*listener*) al objeto `peticion` para saber cuándo está disponible la respuesta del servidor.

## Eventos de XMLHttpRequest

Tenemos diferentes eventos que el servidor envía para informarnos del estado de nuestra petición y que nosotros podemos capturar:

El evento **readystatechange** se produce cada vez que el servidor cambia el estado de la petición.

Cuando hay un cambio en el estado cambia el valor de la propiedad **readyState** de la petición. Sus valores posibles son:

- 0: petición no iniciada (se ha creado el objeto XMLHttpRequest)
- 1: establecida conexión con el servidor (se ha hecho el *open*)
- 2: petición recibida por el servidor (se ha hecho el *send*)
- 3: se está procesando la petición
- 4: petición finalizada y respuesta lista (este es el evento que nos interesa porque ahora tenemos la respuesta disponible). A nosotros sólo nos interesa cuando su valor sea 4 que significa que ya están los datos. En ese momento la propiedad **status** contiene el estado de la petición HTTP (200: Ok, 404: Not found, 500: Server error, ...) que ha devuelto el servidor.

Cuando **readyState** vale 4 y **status** vale 200: tenemos los datos en la propiedad **responseText** (o **responseXML** si el servidor los envía en formato XML).

Ejemplo:

```
const peticion = new XMLHttpRequest();
console.log("Estado inicial de la petición: " + peticion.readyState);
peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
console.log("Estado de la petición tras el 'open': " + peticion.readyState);
peticion.send();
console.log("Petición hecha");
peticion.addEventListener('readystatechange', function() {
  console.log("Estado de la petición: " + peticion.readyState);
  if (peticion.readyState === 4) {
    if (peticion.status === 200) {
      console.log("Datos recibidos:");
      let usuarios = JSON.parse(peticion.responseText); // Pasamos los datos JSON a un objeto
      console.log(usuarios);
    } else {
      console.log("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
    }
  }
})
console.log("Petición acabada");
```

El resultado de ejecutar ese código es el siguiente:

```

1 const peticion = new XMLHttpRequest();
2 console.log("Estado inicial de la petición: " + peticion.readyState);
3 peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
4 console.log("Estado de la petición tras el 'open': " + peticion.readyState);
5 peticion.send();
6 console.log("Petición hecha");
7 peticion.addEventListener('readystatechange', function() {
8   console.log("Estado de la petición: " + peticion.readyState);
9   if (peticion.readyState === 4) {
10     if (peticion.status === 200) {
11       console.log("Datos recibidos:");
12       let usuarios = JSON.parse(peticion.responseText); // Convertimos los datos JSON a un objeto
13       console.log(usuarios);
14     } else {
15       console.log("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
16     }
17   }
18 })
19 console.log("Petición acabada");
20

```

Nótese que cuando cambia de estado la petición, `readyState` cambia de valor.

- `readyState` vale 0 al crear el objeto XMLHttpRequest
- `readyState` vale 1 cuando abrimos la conexión con el servidor
- Luego se envía al servidor y es éste el que va informando al cliente de cuándo cambia el estado

**MUY IMPORTANTE:** La última línea ('Petición acabada') se ejecuta antes que las de 'Estado de la petición'. Ocurre así porque es una **petición asíncrona** y la ejecución del programa continúa sin esperar a que responda el servidor.

Como normalmente no nos interesa saber cada cambio en el estado de la petición, sino que sólo queremos saber cuándo ha terminado de procesarse, tenemos otros **eventos** que nos pueden ser de utilidad:

- **load**: se produce cuando se recibe la respuesta del servidor. Equivale a `readyState==4`. En `status` tendremos el estado de la respuesta
- **error**: se produce si sucede algún error al procesar la petición (de red, de servidor, ...)
- **timeout**: si ha transcurrido el tiempo indicado y no se ha recibido respuesta del servidor. Se puede cambiar el tiempo por defecto modificando la propiedad `timeout` antes de enviar la petición.
- **abort**: si se cancela la petición (se hace llamando al método `.abort()` de la petición).
- **loadend**: se produce siempre que termina la petición, independientemente de si se recibe respuesta o sucede algún error (incluyendo un `timeout` o un `abort`).

Ejemplo de código que sí usaremos:

```

const peticion=new XMLHttpRequest();
peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
peticion.send();
peticion.addEventListener('load', function() {
  if (peticion.status==200) {
    let usuarios=JSON.parse(peticion.responseText);
    // procesamos los datos que tenemos en usuarios
    console.log('correcto:');
    console.log(usuarios);
  } else {
    muestraError(peticion);
  }
})
peticion.addEventListener('error', muestraError);
peticion.addEventListener('abort', muestraError);
peticion.addEventListener('timeout', muestraError);

function muestraError(peticion) {

```

```

if (peticion.status) {
    console.log("Error "+peticion.status+" ("+peticion.statusText+) en la petición");
} else {
    console.log("Ocurrió un error o se abortó la conexión");
}
}

```

Conviene recordar que tratamos con peticiones asíncronas por lo que tras la línea:

```
peticion.addEventListener('load', function() {
```

no se ejecuta la línea siguiente:

```
if (peticion.status==200) {
```

sino la de:

```
peticion.addEventListener('error', muestraError);
```

Una petición asíncrona es como pedir una pizza: tras encargarla por teléfono, lo siguiente no es ir a la puerta a recogerla, sino que seguimos haciendo cosas por casa y cuando suena el timbre de casa entonces vamos a la puerta a por ella.

Prueba el ejemplo anterior con la petición correcta (tal como está) y cambiando petición.open() para que se produzca algún error.

Prueba a hacer peticiones válidas y erróneas con el fichero *peliculas.json* en json-server. Prueba con el servidor json-server apagado.

### Ejemplo:

Realiza peticiones petición GET al archivo de datos “*peliculas.json*” utilizando el servior **json-server** que estará a la escucha de peticiones en el puerto 4000 de tu equipo.

- Una petición debe devolver las películas cuyo director sea Tarantino. Visualiza el resultado en consola.
- Realizar otra petición GET errónea para visualizar el error.
- Realiza una petición válida con el servidor apagado.

## Ejemplos de envío de datos

Vamos a ver algunos ejemplos de **envío de datos al servidor con POST**. Supondremos que tenemos una **página con un formulario para dar de alta nuevos productos**:

El **fichero *productos.json*** de partida puede ser el siguiente:

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    }
  ]
}
```

Index.html sería:

```
<form id="addProduct">
    <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
    <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>

    <button type="submit">Añadir</button>
</form>
```

### Enviar datos al servidor en formato JSON

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
    const newProduct = {
        name: document.getElementById("name").value,
        descrip: document.getElementById("descrip").value,
    }
    const peticion = new XMLHttpRequest();
    peticion.open('POST', 'http://localhost:4000/productos');
    peticion.setRequestHeader('Content-type', 'application/json'); // Siempre tiene
                                                                // que estar esta línea si se envían datos
    peticion.send(JSON.stringify(newProduct)); // Hay que convertir el objeto
                                                // a una cadena de texto JSON para enviarlo
    peticion.addEventListener('load', function() {
        // procesamos los datos
    })
})
```

Para enviar el objeto hay que convertirlo a una cadena JSON con la función **JSON.stringify()** (es la opuesta a **JSON.parse()**). Y siempre que se envían datos al servidor hay que indicar el formato que tienen en la cabecera de **Content-type**:

```
peticion.setRequestHeader('Content-type', 'application/json');
```

**Para evitar problemas:** Ejecutar la página en el navegador después de iniciar el servicio json-server en el servidor.

### Enviar datos al servidor en formato URLEncoded

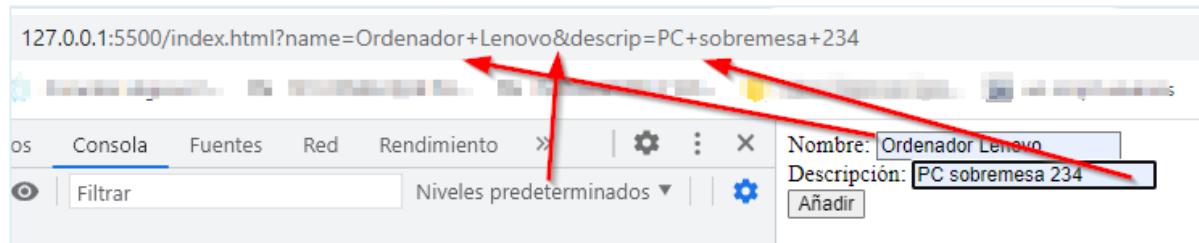
Con el mismo archivo productos.json y el mismo index.html, cambiamos el .js

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
    // Aquí va el código para comprobar que los datos son correctos
    const name = document.getElementById("name").value;
    const descrip = document.getElementById("descrip").value;

    const peticion = new XMLHttpRequest();
    peticion.open('POST', 'http://localhost:4000/productos');
    peticion.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');// formato
    // la siguiente línea lleva & para separar los parámetros
    peticion.send('name=' + encodeURIComponent(name) + '&descrip=' + encodeURIComponent(descrip));
    peticion.addEventListener('load', function() {
        // procesamos los datos
    })
})
```

```
    })
})
```

En este caso los datos se envían como hace el navegador por defecto en un formulario. Recordad **siempre codificar lo que introduce el usuario para evitar problemas con caracteres no estándar y ataques SQL Injection.**



### Enviar ficheros al servidor con FormData

[FormData](#) es una interfaz de XMLHttpRequest que permite construir fácilmente pares de clave=valor para enviar los datos de un formulario. Se envían en el mismo formato en que se enviarían directamente desde un formulario ("multipart/form-data") por lo que no hay que poner encabezado de 'Content-type'.

Vamos a añadir al formulario un campo donde el usuario pueda subir la foto del producto:

```
<form id="addProduct">
  <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
  <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>
  <label for="photo">Fotografía: </label><input type="file" name="photo" id="photo" required><br>

  <button type="submit">Añadir</button>
</form>
```

Podemos enviar al servidor todo el contenido del formulario:

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
  const peticion=new XMLHttpRequest();
  const datosForm = new FormData(document.getElementById('addProduct'));
  // Automáticamente ha añadido todos los inputs, incluyendo tipo 'file', blob, ...
  // Si quisiéramos añadir algún dato más haríamos:
  formData.append('otrodato', 12345);
  // Y lo enviamos
  peticion.open('POST', 'https://localhost/products');
  peticion.send(datosForm);
  peticion.addEventListener('load', function() {
    // procesamos los datos aquí
  })
})
```

También podemos enviar sólo los campos que queramos:

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
```

```
const formData=new FormData(); // creamos un formData vacío
formData.append('name', document.getElementById('name').value);
formData.append('descrip', document.getElementById('descrip').value);
formData.append('photo', document.getElementById('photo').files[0]);

const peticion=new XMLHttpRequest();
peticion.open('POST', 'https://localhost/products');
peticion.send(formData);
peticion.addEventListener('load', function() {
    // procesamos los datos aquí
})
})
```

Más información de cómo usar formData en [MDN web docs](#).



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 3 .....	1
Problema con las peticiones Ajax y cómo resolverlo:.....	1
Formulario para obtener (GET) un producto dado su id:.....	3
Funciones callback .....	4
Promesas.....	5
Fetch .....	8
Propiedades y métodos de la respuesta .....	9
Gestión de errores con fetch.....	10
Otros métodos de petición POST, PUT.....	11

## JavaScript - Ajax 3

### Problema con las peticiones Ajax y cómo resolverlo:

Vamos a ver un ejemplo de una llamada a Ajax. Vamos a hacer una página que muestre en un párrafo el nombre y la descripción del producto del cual indiquemos su id en un input. En resumen, lo que hacemos es:

1. El usuario de nuestra aplicación introduce el código (id) del producto del que queremos ver sus datos.
2. Tenemos un escuchador para que al introducir un código de un usuario llamamos a una función `getProd()` que:
  - Se encarga de hacer la petición Ajax al servidor
  - Si se produce un error se encarga de informar al usuario de nuestra aplicación
3. Cuando se reciben los datos deben pintarse en la tabla

El archivo `productos.json` que contiene los datos:

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    }
  ]
}
```

La página html sería algo así:

```
<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
```

```

<button type="submit">Buscar</button>
<p id="p1">Aquí vendrán los datos</p>
</form>

```

Si Ajax no fuera una petición asíncrona el código Javascript de todo esto sería algo como el siguiente (ATENCIÓN, este código **NO FUNCIONA**):

```

1  const SERVER = 'http://localhost:4000';
2
3 window.addEventListener('load', function() {
4     document.getElementById('addProduct').addEventListener('submit', (event) => {
5         event.preventDefault();
6         let idProd = document.getElementById('id-prod').value;
7         if (isNaN(idProd) || idProd == '') {
8             alert('Debes introducir un número');
9         } else {
10            const datos = getProd(idProd);
11            console.log(datos);
12            // pintamos los datos en la página
13            document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
14        }
15    })
16})
17
18 function getProd(idProd) {
19     const peticion = new XMLHttpRequest();
20     peticion.open('GET', SERVER + '/productos?id=' + idProd);
21     peticion.send();
22     peticion.addEventListener('load', function() {
23         if (peticion.status === 200) {
24             const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
25             console.log(datos);
26             return datos
27         } else {
28             console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
29         }
30     })
31     peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
32 }

```

En la página se observa que aunque hagamos click varias veces, el contenido párrafo no se actualiza:

Id Producto:

nada

Por otro lado, viendo el terminal, observamos que la primera vez que hacemos click con el id=1, devuelve un código 200 (ha sido correcto el envío) y las siguientes veces devuelve código 304 (la información está en la caché porque es la misma petición y no hubo cambios en esos datos desde que devolvió código 200).

```

Loading .\productos.json
Done

Resources
http://localhost:4000/productos

Home
http://localhost:4000

Type s + enter at any time to create a snapshot of the database
GET /productos?id=2 200 11.331 ms - 65
GET /productos?id=2 304 24.557 ms - -
GET /productos?id=2 304 37.382 ms - -

```

Si miramos en la consola, se aprecia que en la línea 25 los datos obtenidos son los mismos, al *parsear* se obtiene un array de longitud 1, cuyo único elemento es el objeto que buscamos.

```
undefined                                         get200bad.js:11
✖ ► Uncaught TypeError: Cannot read properties of undefined (reading '0')
  at HTMLElement.<anonymous> (get200bad.js:13:54)
▶ [{}]
undefined                                         get200bad.js:11
✖ ► Uncaught TypeError: Cannot read properties of undefined (reading '0')
  at HTMLElement.<anonymous> (get200bad.js:13:54)
▶ [{}]
undefined                                         get200bad.js:11
✖ ► Uncaught TypeError: Cannot read properties of undefined (reading '0')
  at HTMLElement.<anonymous> (get200bad.js:13:54)
                                         get200bad.js:25
▼ [{}]
▶ 0: {name: 'Teclado', descrip: 'Teclado mecánico Cherry ps/2', id: 2}
  length: 1
▶ [[Prototype]]: Array(0)
```

Pero también se observa que la línea 11 recibe *undefined*, y más tarde, aparece el array que vemos en la línea 25. Lo que explica el error de la línea 13: intenta pintar los datos que aún no ha recibido porque la respuesta tarda en llegar. Este error debemos corregirlo.

Visto de otro modo: El array *datos* es *undefined* en la línea 11 porque cuando se llama a *getProd(idProd)*, esta función no devuelve nada de inmediato, sino que devuelve tiempo después, cuando el servidor contesta, pero entonces nadie está escuchando.

La solución es que todo el código, no sólo de la petición Ajax sino también el de qué hacer con los datos cuando llegan, se encuentre en la función que pide los datos al servidor.

Está resuelto en el apartado siguiente:

## Formulario para obtener (GET) un producto dado su id:

Este ejemplo resuelve el problema del apartado anterior.

```
<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>
```

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault(); // Cancela la acción predeterminada del evento 'submit'
    const idProd = document.getElementById('id-prod').value;
    if (isNaN(idProd) || idProd == '') {
      alert('Debes introducir un número');
    } else {
      getProducto(idProd);
    }
  });
});
```

```

        }
    })
}

function getProducto(idProd) {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', SERVER + '/productos?id=' + idProd);
    peticion.send();
    peticion.addEventListener('load', function() {
        if (peticion.status === 200) {
            const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
            document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
            console.log(datos);
        } else {
            console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
        }
    })
    peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
}

```

**Nota:** Al poner `event.preventDefault()` en la función asociada a un evento, se cancela la acción predeterminada que tuviera ese evento.

Para pintar los datos, hay que tener en cuenta que GET devuelve un array (en este caso con un único elemento).

Este código funciona correctamente, pero tiene una pega: tenemos que tratar los datos (en este caso pintarlos en el párrafo) en la función que gestiona la petición, porque es la que sabe cuándo están disponibles esos datos. Por tanto, nuestro código es poco claro.

Esto se podría mejorar usando una función **callback**.

## Funciones callback

La idea es que creamos una función que procese los datos (`renderProd`) y se la pasamos a `getProd` como una función **callback** para que la llame cuando tenga los datos:

```

const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
    document.getElementById('addProduct').addEventListener('submit', (event) => {
        event.preventDefault();
        let idProd = document.getElementById('id-prod').value;
        if (isNaN(idProd) || idProd.trim() == '') {
            alert('Debes introducir un número');
        } else {
            getProd(idProd, renderProd)
        }
    })
})

function renderProd(datos) {
    // aquí pintamos los datos. Habrá casos que será muy extenso.
    document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
}

```

```

}

function getProd(idProd, callback) {
  const peticion = new XMLHttpRequest()
  peticion.open('GET', SERVER + '/productos?id=' + idProd);
  peticion.send()
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
      callback(JSON.parse(peticion.responseText));
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición")
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'))
}

```

Hemos creado una función que se ocupa de renderizar (pintar) los datos y se la pasamos a la función que gestiona la petición para que la llame cuando los datos están disponibles (cuando petición.status==200).

Utilizando la función `callback` hemos conseguido que `getProd()` se encargue sólo de obtener los datos y cuando los tenga se los pasa a la función encargada de pintarlos en el documento de la página manipulando el DOM.

## Promesas

Sin embargo hay una forma más limpia de resolver una función asíncrona, y es que el código se parezca al primero que hicimos que no funcionaba, donde la función `getProd()` sólo debía ocuparse de obtener los datos y devolverlos a quien se los pidió:

```

...
let idProd = document.getElementById('id-prod').value;
if (isNaN(idProd) || idProd == '') {
  alert('Debes introducir un número');
} else {
  const datos = getProd(idProd);
  // y aquí usamos los datos recibidos para pintar los datos
}
...

```

La nueva forma es convirtiendo a `getProd()` en una **promesa**.

Cuando se realiza una llamada a una promesa, quien la llama puede usar métodos que NO SE EJECUTARÁN hasta que la promesa se haya resuelto, es decir, hasta que el servidor haya contestado. Estos métodos son:

- `.then(function(datos) { ... })`: se ejecuta si la promesa se ha resuelto satisfactoriamente. Su parámetro es una **función**. Esta función recibirá como parámetro los datos que haya devuelto la promesa (que serán los datos pedidos al servidor). En este caso la promesa se los envía con `resolve(...)`
- `.catch(function(datos) { ... })`: se ejecuta si se ha rechazado la promesa (normalmente porque se ha recibido una respuesta errónea del servidor). Ejecuta una función que recibe como parámetro la información pasada por la promesa al ser rechazada (que será información sobre el error producido). En este caso la promesa se los envía con `reject(...)`

De esta manera el código quedaría:

```

let idProd = document.getElementById('id-prod').value;
if (isNaN(idProd) || idProd == '') {
    alert('Debes introducir un número');
} else {
    getProd(idProd)
        // en el .then() estará el código a ejecutar cuando tengamos los datos
        .then((datos) => {
            document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
        })
        // en el .catch() está el tratamiento de errores
        .catch((error) => console.error(error))
}

```

Para convertir a `getProd()` en una promesa solo hay que “envolver” las instrucciones de la función en una promesa.

```

function getProd(idProd){
    return new Promise((resolve, reject)=>{
        // Aquí el contenido de GetProd()
    })
}

```

Esto hace que devuelva un objeto de tipo `Promise` (`return new Promise()`) cuyo único parámetro es una función que recibe 2 parámetros:

- `resolve`: función *callback* a la que se llamará cuando se resuelva la promesa satisfactoriamente.
- `reject`: función *callback* a la que se llamará si se resuelve la promesa con errores.

El funcionamiento es:

- cuando la promesa se resuelva satisfactoriamente `getProd` llama a la función `resolve()` y le pasa los datos recibidos por el servidor. Esto hace que se ejecute el método `.then()` de la llamada a la promesa que recibirá como parámetro esos datos. Se ejecuta `.then()` después de `resolve()`
- si se produce algún error se rechaza la promesa llamando a la función `reject()` y pasando como parámetro la información del fallo producido y esto hará que se ejecute el `.catch()` en la función que llamó a la promesa. Se ejecuta `.catch()` después de `resolve()`.

Por tanto, nuestra función `getProd` ahora quedará así:

```

function getPosts(idProd) {
    return new Promise((resolve, reject) => {
        const peticion = new XMLHttpRequest();
        peticion.open('GET', SERVER + '/productos?id=' + idProd);
        peticion.send();
        peticion.addEventListener('load', () => {
            if (peticion.status === 200) {
                resolve(JSON.parse(peticion.responseText));
            } else {
                reject("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
            }
        })
        peticion.addEventListener('error', () => reject('Error en la petición HTTP'));
    })
}

```

```

    }
}

```

Se observa que el único cambio es la primera línea donde se convierte la función `getProd()` en una **promesa**, y que luego para “devolver” los datos a quien llama a `getProd` en lugar de hacer un `return`, que ya se ha visto que no funciona, se hace un `resolve` si todo ha ido bien o un `reject` si ha fallado.

Desde donde llamamos a la promesa nos suscribimos a ella usando los métodos `.then()` y `.catch()` vistos anteriormente.

Básicamente, lo que nos va a proporcionar el uso de promesas es un código más claro y mantenible, ya que el código a ejecutar cuando se obtengan los datos asíncronamente estará donde se piden esos datos y no en una función escuchadora o en una función *callback*.

Utilizando promesas vamos a conseguir que la función que pide los datos sea quien los obtiene y los trate o quien informa si hay un error.

El código del ejemplo de obtener un producto desde su id usando promesas sería el siguiente:

El código HTML sería igual que antes:

```

<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>

```

Y el código Javascript sería:

```

const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      getProd(idProd)
        .then(function(datos) {
          // aquí pintamos los datos. Habrá casos que será muy extenso.
          document.getElementById('p1').innerHTML = datos[0].name+ " " +datos[0].descrip;
        })
        .catch(function(error) {
          console.error(error)
        })
    }
  })
}

function getProd(idProd) {
  return new Promise(function(resolve, reject) {
    let peticion = new XMLHttpRequest()

```

```

peticion.open('GET', SERVER + '/productos?id=' + idProd);
peticion.send()
peticion.addEventListener('load', () => {
  if (peticion.status === 200) {
    resolve(JSON.parse(peticion.responseText))
  } else {
    reject("Error " + this.status + " (" + this.statusText + ") en la petición")
  }
})
peticion.addEventListener('error', () => reject('Error en la petición HTTP'))
})
}

```

**Nota:** Los errores del servidor SIEMPRE llegan a la consola. En el ejemplo anterior si se produce un error de servidor aparecerá 2 veces en la consola: la primera que es el error original y la segunda donde está pintado con `console.error()`.

En este ejemplo, aunque pidamos un id de producto no existente en el servidor, devuelve un código 200 y un array vacío. El error que se ve en consola es porque no puede pintar un array vacío. Este error es capturado por `catch()`, y permite que la página siga funcionando.

Más sobre Promesas en [MDN web docs](#) y en los documentos anexos.

## Fetch

La [API Fetch](#) es una interfaz para manipular la información del servidor en forma de promesas.

Como el código que hay que escribir para hacer una petición Ajax es largo y repetitivo, la API-Fetch **permite realizar una petición Ajax genérica que directamente devuelve en forma de promesa**.

Básicamente **lo que hace es encapsular en una función todo el código que se repite siempre en una petición AJAX** (crear la petición, hacer el `open`, el `send`, escuchar los eventos, ...).

La función `fetch` es similar a la función `getProd` que hemos creado antes, pero es genérica para que sirva para cualquier petición pasándole la URL. Su código es algo similar a:

```

function fetch(url) {
  return new Promise((resolve, reject) => {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', url);
    peticion.send();
    peticion.addEventListener('load', () => {
      resolve(peticion.responseText);
    })
    peticion.addEventListener('error', () => reject('Network Error'));
  })
}

```

Hay 2 cosas que cambian respecto a nuestra función `getProd()`:

1. `fetch` devuelve los datos “en crudo” por lo que si la respuesta está en formato JSON habrá que convertirlos. Para ello dispone del método `.json()` que funciona como el `JSON.parse()`. Este método **devuelve una nueva promesa a la que nos suscribimos con un nuevo `.then`**. Es lo que se llama **promesas encadenadas**. Ejemplo.:

```

fetch('http://localhost:4000/productos?id=' + idProd)
  .then(response => response.json()) // los datos son una cadena JSON
  .then(myData => { // ya tenemos los datos en _myData_ como un objeto o array
    // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en el párrafo de la página)
    console.log(myData)
  })
  .catch(err => console.error(err));

```

2. `fetch` llama a `resolve` siempre que el servidor conteste, sin comprobar si la respuesta es de éxito (200, 201, 304,...) o de error (4xx, 5xx). Por tanto, siempre se ejecutará el `then` excepto si se trata de un error de red y el servidor no responde (no ha habido respuesta).

#### Propiedades y métodos de la respuesta

La respuesta devuelta por `fetch()` tiene las siguientes propiedades y métodos:

- **`status`**: el código de estado devuelto por el servidor (200, 404, ...)
- **`statusText`**: el texto correspondiente a ese código (Ok, Not found, ...)
- **`ok`**: booleano que vale `true` si el status está entre 200 y 299 y `false` en caso contrario
- **`json()`**: devuelve una promesa que se resolverá con los datos de la respuesta convertidos a un objeto (a los datos les realiza un `JSON.parse()`)
- otros métodos para convertir los datos según el formato que tengan: **`text()`, `blob()`, `formData()`, ...** Todos devuelven una promesa con los datos de distintos formatos convertidos.

El ejemplo que hemos visto con las promesas, usando `fetch` quedaría:

El código HTML sería igual que antes:

```

<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>

```

Y el código Javascript sería:

```

const SERVER = 'http://localhost:4000';

window.addEventListener('load', () => {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value;
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      fetch(SERVER + '/productos?id=' + idProd)
        .then((response) => response.json())
        .then((datos) => {
          // aquí pintamos los datos. Habrá casos que será muy extenso.
          document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
        })
    }
  })
})

```

```
    })
    .catch((error) => console.error(error))
  }
})
})
```

Pero este ejemplo fallaría si hubiéramos puesto mal la *url* ya que contestaría con un 404 y fallaría al ejecutar el *then* intentando pintar un producto que no tenemos.

## *Gestión de errores con fetch*

Según [MDN](#), la promesa devuelta por la `API fetch` sólo es rechazada en el caso de un error de red, es decir, el `.catch` sólo saltará si no hemos recibido respuesta del servidor; en caso contrario la promesa siempre es resuelta.

Por tanto, para saber si se ha resuelto satisfactoriamente o no, debemos comprobar la propiedad `.ok` de la respuesta. El código correcto del ejemplo anterior gestionando los posibles errores del servidor sería:

```
// tratando los errores en fetch
const SERVER = 'http://localhost:4000';

window.addEventListener("load", () => {
  document.getElementById("addProduct").addEventListener("submit", (event) => {
    event.preventDefault();
    let idProd = document.getElementById("id-prod").value;
    if (isNaN(idProd) || idProd.trim() == "") {
      alert("Debes introducir un número");
    } else {
      fetch(SERVER + "/productos?id=" + idProd)
        .then((response) => {
          if (!response.ok) {
            // lanzamos un error que interceptará el .catch()
            throw `Error ${response.status} de la BBDD: ${response.statusText}`;
          }
          return response.json(); // devolvemos la promesa que hará el JSON.parse
        })
        .then((datos) => {
          // ya tenemos los datos formateados
          // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en la página)
          document.getElementById("p1").innerHTML =
            datos[0].name + " " + datos[0].descrip;
          console.log(datos);
        })
        .catch((error) => console.error(error));
    }
  });
});
```

En este caso, si la respuesta del servidor no es `ok` lanzamos un error que es interceptado por nuestro propio `catch`

**Otros métodos de petición POST, PUT...**

Los ejemplos anteriores hacen peticiones GET al servidor. Para peticiones que no sean GET la función `fetch()` admite un segundo parámetro: un objeto con la información a enviar en la petición HTTP. Ej.:

```
// Otros métodos de petición POST, PUT...
fetch(url, {
  method: 'POST', // o 'PUT', 'GET', 'DELETE'
  body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
  headers:{
    'Content-Type': 'application/json'
  }
}).then
```

**Ejemplo de una petición para añadir datos:**

```
fetch(url, {
  method: 'POST',
  body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
  headers:{
    'Content-Type': 'application/json'
  }
})
.then(response => {
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  return response.json()
})
.then(datos => {
  alert('Datos recibidos')
  console.log(datos)
})
.catch(err => {
  alert('Error en la petición HTTP: '+err.message);
})
```

Más ejemplos en [MDN web docs](#)

Realiza una página web para que cualquier usuario de la página pueda dar de alta nuevos productos en `productos.json` utilizando `fetch`. El usuario introducirá el nombre y la descripción del producto, la API asignará el id que corresponda.

En la solución del ejercicio, si añadimos al objeto producto que se va a enviar con POST la propiedad `id` con un valor puesto a mano, **puede ocurrir:**

- Si el número de id se repite, provocará un error 500, y el producto no se añadirá a la BBDD.
- Si el id no existe, se creará el producto con el valor id que se haya puesto a mano.
- Si no se pone id, la API colocará el número siguiente al último id del fichero. (Cuidado!!)



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 4 .....	1
async / await.....	1
Gestión de errores en async/await .....	3
Ejemplos de GET y POST con Fetch (utilizando y sin usar async-await) .....	4
Con nuestro fichero productos.json:.....	4
Index.html para GET.....	4
Código javascript.....	5
Index.html para POST.....	6
Código JavaScript .....	6
Hacer varias peticiones simultáneamente. Promise.all .....	8
Single Page Application .....	10
Resumen de llamadas asíncronas .....	10
CORS .....	11

## JavaScript - Ajax 4

Si en "Visual Studio Code" dejo abierta la carpeta desde la que he abierto (lanzado) la página con "Live Server" al hacer clic en el botón "submit", borra los inputs y el contenido de la consola, es como si reiniciase la página.

Para solucionar estos problemas lo más fácil es utilizar un servidor web como apache.

Si quisieramos seguir usando LiveServer: después de ejecutar la aplicación con LiveServer, será necesario cambiar la carpeta de Visual Studio Code (open folder → otra carpeta distinta). La aplicación que hemos dejado en ejecución ahora sí funcionará (comprobar que el servidor json está funcionando).

### async / await

Son instrucciones nuevas introducidas en ES2017 que permiten escribir el código de peticiones asíncronas como si fueran sincronas, lo que facilita su comprensión.

Hay que tener en cuenta que **NO** están soportadas por navegadores antiguos.

Si hubiéramos utilizado async/await en el primer ejemplo que hicimos, sí habría funcionado.

Se puede llamar a cualquier función asíncrona (por ejemplo, una promesa como `fetch`) **anteponiendo la palabra `await` a la llamada**. Esto provocará que la ejecución se "espere" a que se resuelva la promesa devuelta por esa función. Así nuestro código se asemeja a código sincrónico ya que no continúan ejecutándose las instrucciones que hay después de un `await` hasta que esa petición se ha resuelto.

Cualquier función que realice un `await` pasa a ser asíncrona ya que no se ejecuta en ese momento, sino que se espera un tiempo.

Para indicarlo debemos anteponer la palabra **async** a la declaración **function**. Al hacerlo, automáticamente se “envuelve” esa función en una promesa (o sea, que esa función pasa a devolver una promesa, a la que podríamos ponerle un **await** o un **.then()**).

Siguiendo con el ejemplo anterior, el código Javascript sería:

```
async function pideDatos() {
  const response = await fetch('http://localhost:4000/productos?id=' + idProd);
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`;
  }
  const myData = await response.json(); // recuerda que .json() tb es una promesa
  return myData;
}
...
// Y llamaremos a esa función con
const myData = await pideDatos();
```

**Nota:** solo se puede utilizar **await** dentro de funciones declaradas con la palabra reservada **async**.

Observa la diferencia: si hacemos:

```
const response = fetch('http://localhost:4000/productos?id=' + idProd);
```

obtenemos en **response** una promesa, y para obtener el valor se debería hacer **response.then()**. Pero si hacemos:

```
const response = await fetch('http://localhost:4000/productos?id=' + idProd);
```

lo que obtenemos en **response** es ya el valor devuelto por la promesa cuando se resuelve.

Con esto conseguimos que llamadas asíncronas se comporten como instrucciones síncronas, lo que aporta claridad al código.

Hay algunos ejemplos del uso de **async / await** en la [página de MDN](#).

Siguiendo con el ejemplo de obtener datos de un producto indicando su id:

El código HTML sería igual que antes:

```
<form id="getProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>
```

Y el código Javascript sería:

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', () => {
  document.getElementById('getProduct').addEventListener('submit', async (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
```

```

if (isNaN(idProd) || idProd.trim() == '') {
    alert('Debes introducir un número')
} else {
    const datos = await getData(idProd)
    // La ejecución se para en la sentencia anterior hasta que
    // contesta la función getData
    document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
}
})
})
}

async function getData(idProd) {
    const response = await fetch(SERVER + '/productos?id=' + idProd)
    if (!response.ok) {
        throw `Error ${response.status} de la BBDD: ${response.statusText}`
    }
    const datos = await response.json()
    return datos
}

```

### Gestión de errores en async/await

En el código anterior no estamos tratando los posibles errores que se pueden producir.

Con `async / await` los errores se tratan como en las excepciones, con `try ... catch`.

El código que resulta para el ejemplo de pintar los datos de un producto indicando su id:

El código HTML sería igual que siempre:

```

<form id="getProduct">
    <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
    <br>
    <button type="submit">Buscar</button>
    <p id="p1">Aquí vendrán los datos</p>
</form>

```

Y el código Javascript sería:

```

// fetch con async/await
// tratando errores con try-catch
const SERVER = 'http://localhost:4000'

window.addEventListener('load', () => {
    document.getElementById('getProduct').addEventListener('submit', async (event) => {
        event.preventDefault();
        let idProd = document.getElementById('id-prod').value
        if (isNaN(idProd) || idProd.trim() == '') {
            alert('Debes introducir un número')
        } else {
            try {
                const datos = await getData(idProd)
                // La ejecución se para en la sentencia anterior hasta que
                // contesta la función getData
            }
        }
    })
})
}

async function getData(idProd) {
    const response = await fetch(SERVER + '/productos?id=' + idProd)
    if (!response.ok) {
        throw `Error ${response.status} de la BBDD: ${response.statusText}`
    }
    const datos = await response.json()
    return datos
}

```

```

        document.getElementById('p1').innerHTML = datos.name+" "+datos.descrip;
    } catch (err) {
        console.log("mal");
        console.error(err);
        return;
    }
}
})
}

async function getData(idProd) {
const response = await fetch(SERVER + '/productos?id=' + idProd)
if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
}
const datos = await response.json()
return datos
}

```

También podemos tratar los errores sin usar *try...catch*, porque como una función asíncrona devuelve una promesa, podemos suscribirnos directamente a su *.catch*

### Ejemplos de GET y POST con Fetch (utilizando y sin usar async-await)

*Con nuestro fichero productos.json:*

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    },
    {
      "id": 2,
      "name": "Monitor Ph-21",
      "descrip": "Monitor Phillips SVGA 21 \\""
    },
    {
      "id": 3,
      "name": "Ratón Logi-Laser",
      "descrip": "Tatón Láser Logitech Pro USB"
    }
  ]
}
```

*Index.html para GET*

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
```

```

<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
  <form id="getProduct">
    <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
    <br>
    <button type="submit">Buscar</button>
    <p id="p1">Aquí vendrán los datos</p>
  </form>
  <script src="js/get.js"></script>

</body>
</html>

```

### Código javascript

<pre> // GET con fetch const SERVER= 'http://127.0.0.1:4000'; window.addEventListener("load", ()=&gt;{   document.getElementById('getProduct')     .addEventListener('submit', (event)=&gt;{       event.preventDefault();       let idProd=document.getElementById('id-prod').value;       let promesa=fetch(SERVER + '/productos?id=' + idProd);         //obtenemos una promesa       promesa         .then((dato1)=&gt;dato1.json())           //ejecuta lo de dentro, dato1 es lo que then ha         recibido         .then((dato2)=&gt;{ // dato2 es lo que recibe del         anterior then           console.log(dato2);           document.getElementById('p1').innerText=dato2[0].nam         e;         })         .catch((problema)=&gt; document.getElementById('p1')           .innerText="Ha habido error: "+problema);       });     });   }); </pre>	<pre> // fetch con async await // se trata de separar la zona donde // se pinta, de la petición a la API. // ya no usamos then, porque await garantiza // que ya llegó la respuesta const SERVER= 'http://127.0.0.1:4000'; window.addEventListener("load", ()=&gt;{   document.getElementById('getProduct')     .addEventListener('submit', async   (event)=&gt;{     event.preventDefault();     const idProd=document.getElementById('id-   prod').value;     const dato1=await getData(idProd);     document.getElementById('p1').innerText=dato1[0]       .name + " " +dato1[0].descrip;   }); });  async function getData(idProd){   const dato1 = await fetch(SERVER+'/productos?id='+idProd);   // lo normal sería mirar si hubo error:   /* if (!dato1.ok) {     throw `error\${dato1.statusText} \${dato1.statusText}` */   */   console.log(dato1);   const dato2 = await dato1.json();   console.log(dato2);   return dato2; } </pre>
--	--

<pre>&gt; JS get.js &gt; ... 1 // fetch con 2 const SERVER= 'http://127.0.0.1:4000'; 3 window.addEventListener("load", ()=&gt;{ 4   document.getElementById('getProduct').addEventListener('submit', (event)=&gt;{ 5     event.preventDefault(); 6     let idProd=document.getElementById('id-prod').value; 7     let promesa=fetch(SERVER + '/productos?id=' + idProd); //obtenemos una promesa 8     promesa 9     .then((dato1)=&gt;dato1.json()) //ejecuta lo de dentro, dato1 es lo que then ha recibido 10     .then((dato2)=&gt;{ // dato2 es lo que recibe del anterior then 11       console.log(dato2); 12       document.getElementById('p1').innerText=dato2[0].name; 13     }) 14     .catch((problema)=&gt; document.getElementById('p1').innerText="Ha habido error: "+problema); 15 }); 16 }); 17 })</pre>	<pre>js &gt; JS get.js &gt; ⚡ getData 1 // fetch con async await 2 // se trata de separar la zona donde se pinta, de la petición a la API. 3 // ya no usamos then, porque await garantiza que ya llegó la respuesta 4 const SERVER= 'http://127.0.0.1:4000'; 5 window.addEventListener("load", ()=&gt;{ 6   document.getElementById('getProduct').addEventListener('submit', async (event)=&gt;{ 7     event.preventDefault(); 8     const idProd=document.getElementById('id-prod').value; 9     const dato1=await getData(idProd); 10     document.getElementById('p1').innerText=dato1[0].name + " " +dato1[0].descrip; 11   }); 12 }); 13 14 async function getData(idProd){ 15   const dato1 = await fetch(SERVER + '/productos?id=' + idProd); 16   // lo normal sería mirar si hubo error: 17   // if (!dato1.ok) { throw `error\${dato1.status} \${dato1.statusText}`} 18   console.log(dato1); 19   const dato2 = await dato1.json(); 20   console.log(dato2); 21   return dato2; 22 }</pre>

**Index.html para POST**

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="shortcut icon" href="#" type="image/x-icon">
    <title>Document</title>
</head>
<body>
    <form id="addProduct">
        <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
        <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>

        <button type="submit">Agregar Producto</button>
        <p id="p1"> ...</p>
    </form>
    <script src="js/post.js"></script>
</body>
</html>
```

**Código JavaScript**

<pre>// POST con fetch  const SERVER = 'http://localhost:4000';  window.addEventListener('load', ()=&gt;{     document.getElementById('addProduct').addEventListener('submit', (event)=&gt;{         event.preventDefault();         const nuevoProducto={</pre>	<pre>// POST con fetch y asunc await  const SERVER = 'http://localhost:4000';  async function anadirProducto(nuevoProducto){     const dato1 = await fetch(SERVER+ '/productos',     {         method: 'POST',         body: JSON.stringify(nuevoProducto),         headers: {</pre>
--	--

```
        id:"",
        name:document.getElementById('name').value,
        descrip:document.getElementById('descrip').value
    };
}

const promesa=fetch(SERVER+ '/productos', {
    method: 'POST',
    body: JSON.stringify(nuevoProducto),
    headers: {
        'Content-Type': 'application/json'
    }
})
promesa
    .then((dato1)=>dato1.json())
    .then((dato2)=>{
        //pintamos lo que queramos
        console.log(dato2);
        document.getElementById('p1').innerText=`${dato2.name}
${dato2.descrip}`;
    })
    .catch();
});
});

        }
    });
    const dato2= await dato1.json();
    return dato2;
}

window.addEventListener('load', ()=>{
    document.getElementById('addProduct')
        .addEventListener('submit', async
(event)=>{
        event.preventDefault();
        const nuevoProducto={
            id:"",
            name:document.getElementById('name').value,
            descrip:document.getElementById('descrip').value
        };
        const dato2 = await
anadirProducto(nuevoProducto);

        //pintamos lo que queramos
        console.log(dato2);
        document.getElementById('p1')
            .innerText=`${dato2.name}
${dato2.descrip}`;
    });
});
```

```
> JS post.js > ...
1 // POST con fetch
2 const SERVER = 'http://localhost:4000';
3 window.addEventListener('load', ()=>{
4   document.getElementById('addProduct')
5     .addEventListener('submit', (event)=>{
6     event.preventDefault();
7     const nuevoProducto={
8       id:"",
9       name:document.getElementById('name').value,
10      descrip:document.getElementById('descrip').value
11    };
12
13    const promesa=fetch(SERVER+'/productos', {
14      method: 'POST',
15      body: JSON.stringify(nuevoProducto),
16      headers: {
17        'Content-Type': 'application/json'
18      }
19    })
20    promesa
21      .then((dato1)=>dato1.json())
22      .then((dato2)=>{
23        //pintamos lo que queramos
24        console.log(dato2);
25        document.getElementById('p1')
26          .innerText=`${dato2.name} ${dato2.descrip}`;
27      })
28      .catch();
29  });
30});
```

```
; > JS post.js > ...
1 // POST con fetch y asunc await
2 const SERVER = 'http://localhost:4000';
3 async function anadirProducto(nuevoProducto){
4   const dato1 = await fetch(SERVER+'/productos', {
5     method: 'POST',
6     body: JSON.stringify(nuevoProducto),
7     headers: {
8       'Content-Type': 'application/json'
9     }
10   });
11   const dato2= await dato1.json();
12   return dato2;
13 }
14
15 window.addEventListener('load', ()=>{
16   document.getElementById('addProduct')
17     .addEventListener('submit', async (event)=>{
18     event.preventDefault();
19     const nuevoProducto={
20       id:"",
21       name:document.getElementById('name').value,
22       descrip:document.getElementById('descrip').value
23     };
24
25     const dato2 = await anadirProducto(nuevoProducto);
26
27     //pintamos lo que queramos
28     console.log(dato2);
29     document.getElementById('p1')
30       .innerText=`${dato2.name} ${dato2.descrip}`;
31   });
32});
```

## Hacer varias peticiones simultáneamente. Promise.all

En ocasiones necesitamos hacer más de una petición al servidor.

Por ejemplo; para obtener los productos y sus categorías podríamos hacer:

```
function getTable(table) {
  return new Promise((resolve, reject) => {
    fetch(SERVER + table)
      .then(response => {
        if (!response.ok) {
          throw `Error ${response.status} de la BBDD: ${response.statusText}`
        }
        return response.json()
      })
      .then((data) => resolve(data))
      .catch((error) => reject(error))
  })
}

function getData() {
  getTable('/categories')
    .then((categories) => categories.forEach((category) => renderCategory(category)))
    .catch((error) => renderErrorMessage(error))
  getTable('/products')
    .then((products) => products.forEach((product) => renderProduct(product)))
    .catch((error) => renderErrorMessage(error))
}
```

Pero si para renderizar los productos necesitamos tener las categorías, este código no nos lo garantiza ya que el servidor podría devolver antes los productos, aunque los hemos pedido después.

Una solución sería no pedir los productos hasta tener las categorías:

```
function getData() {
  getTable('/categories')
    .then((categories) => {
      categories.forEach((category) => renderCategory(category))
      getTable('/products')
        .then((products) => products.forEach((product) => renderProduct(product)))
        .catch((error) => renderErrorMessage(error))
    })
    .catch((error) => renderErrorMessage(error))
}
```

pero esto hará más lento nuestro código al no hacer las 2 peticiones simultáneamente.

La solución es usar el método `Promise.all()` al que se le pasa un array de promesas a hacer y devuelve una promesa que:

- se resuelve en el momento en que todas las promesas se han resuelto satisfactoriamente o
- se rechaza en el momento en que alguna de las promesas es rechazada

El código anterior de forma correcta sería:

```
function getData() {
  Promise.all([
    getTable('/categories'),
    getTable('/products')
  ])
    .then(([categories, products]) => {
      categories.forEach((category) => renderCategory(category))
      products.forEach((product) => renderProduct(product))
    })
    .catch((error) => renderErrorMessage(error))
}
```

Lo mismo pasa si en vez de promesas usamos `async/await`. Si hacemos:

```
async function getTable(table) {
  const response = await fetch(SERVER + table)
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const data = await response.json()
  return data
}

async function getData() {
  const responseCategories = await getTable('/categories');
```

```
const responseProducts = await getTable('/products');
categories.forEach((category) => renderCategory(category))
products.forEach((product) => renderProduct(product))
}
```

tenemos el problema de que no comienza la petición de los productos hasta que se reciben las categorías. La solución con `Promise.all()` sería:

```
async function getData() {
  const [categories, products] = await Promise.all([
    getTable('/categories')
    getTable('/products')
  ])
  categories.forEach((category) => renderCategory(category))
  products.forEach((product) => renderProduct(product))
}
```

## Single Page Application

Ajax es la base para construir SPAs que permiten al usuario interactuar con una aplicación web como si se tratara de una aplicación de escritorio (sin “esperas” que dejen la página en blanco o no funcional mientras se recarga desde el servidor).

En una SPA sólo se carga la página de inicio (es la única página que existe) que se va modificando y cambiando sus datos como respuesta a la interacción del usuario.

Para obtener los nuevos datos se realizan peticiones al servidor (normalmente Ajax). La respuesta son datos (JSON, XML, ...) que se muestran al usuario modificando mediante DOM la página mostrada (o podrían ser trozos de HTML que se cargan en determinadas partes de la página, o ...).

## Resumen de llamadas asíncronas

Una llamada Ajax es un tipo de llamada asíncrona que podemos hacer en Javascript.

Aunque hay otros tipos de llamadas asíncronas, como un `setTimeout()` o las funciones manejadoras de eventos.

Como hemos visto, para la gestión de las llamadas asíncronas tenemos varios métodos y los más comunes son:

- funciones `callback`
- `promesas`
- `async / await`

Cuando se produce una llamada asíncrona el orden de ejecución del código no es el que vemos en el programa, ya que el código de respuesta de la llamada no se ejecutará hasta completarse ésta.

Además, si hacemos varias llamadas tampoco sabemos en qué orden se ejecutarán sus respuestas, ya que depende de cuándo finalice cada una.

Si usamos funciones `callback` y necesitamos que cada función no se ejecute hasta que haya terminado la anterior, debemos llamarla en la respuesta a la función anterior, lo que provoca un tipo de código difícil de leer llamado “callback hell”.

Para evitar estos problemas surgieron las **promesas** que permiten evitar las funciones **callback** tan difíciles de leer. Y si necesitamos ejecutar secuencialmente las funciones evitaremos la pirámide de llamadas **callback**.

Aun así, el código no es muy claro. Para mejorarlo tenemos **async y await**. Estas funciones forman parte del estándar ES2017 por lo que no están soportadas por navegadores muy antiguos (aunque siempre podemos transpilar con *Babel*).

## CORS

*Cross-Origin Resource Sharing (CORS)* es un mecanismo de seguridad que incluyen los navegadores y que por defecto impiden que se puedan realizar peticiones Ajax desde un navegador a un servidor con un dominio diferente al de la página cargada originalmente.

Si necesitamos hacer este tipo de peticiones necesitamos que el servidor al que hacemos la petición añada en su respuesta la cabecera **Access-Control-Allow-Origin** donde indiquemos el dominio desde el que se pueden hacer peticiones (o \* para permitirlas desde cualquier dominio).

El navegador comprobará las cabeceras de respuesta y si el dominio indicado por ella coincide con el dominio desde el que se hizo la petición, ésta se permitirá.

Como en desarrollo normalmente no estamos en el dominio de producción (para el que se permitirán las peticiones) podemos instalar en el navegador la extensión **allow CORS** que al activarla deshabilita la seguridad CORS en el navegador.

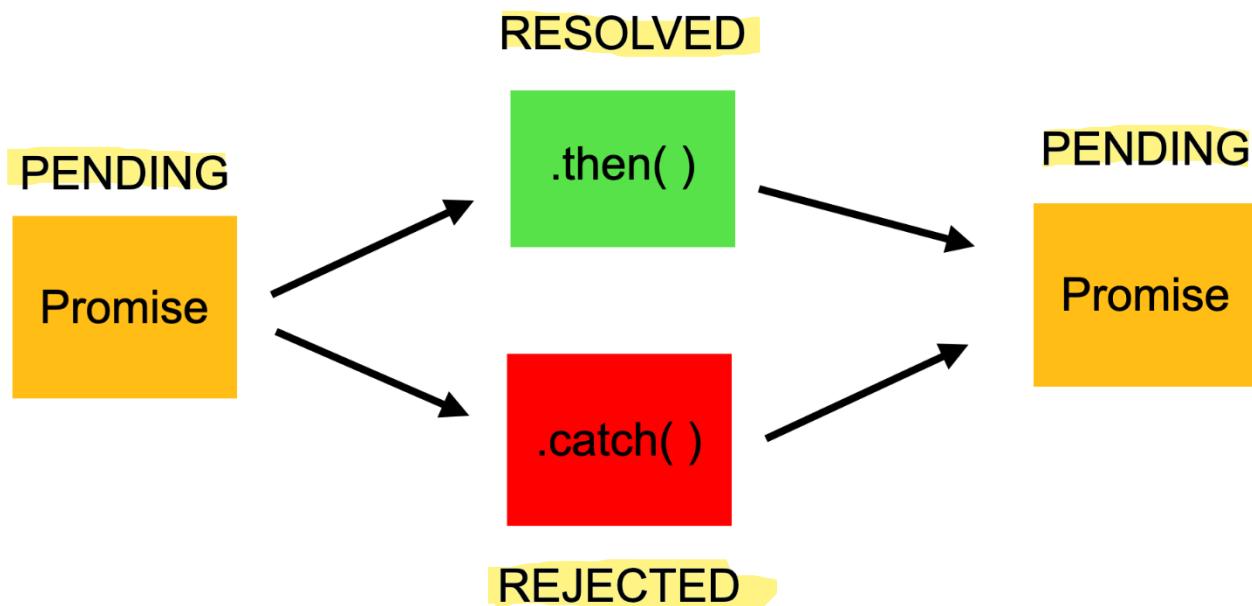
## JavaScript - Promesas

### Promesas en JavaScript

Una Promesa es un objeto. Hay 3 estados del objeto Promesa:

- **Pendiente**: estado inicial, antes de que la promesa tenga éxito o falle
- **Resuelto**: promesa completada
- **Rechazado**: promesa fallida

Veamos una representación de los procesos de Promesas:



Cuando solicitamos datos del servidor mediante una Promesa, estará en modo pendiente hasta que recibamos nuestros datos.

Si logramos obtener la información del servidor, la Promesa se resolverá con éxito. Pero si no obtenemos la información, entonces la Promesa estará en el estado rechazado.

Además, si hay múltiples solicitudes, luego de que se resuelva (o rechace la primera Promesa), comenzará un nuevo proceso al que podemos adjuntarlo directamente mediante un método llamado **encadenamiento**.

La principal **diferencia** entre las funciones de **retrollamadas** y las **promesas** es que adjuntamos una retrollamada a una promesa en lugar de pasársela. Así que todavía usamos funciones de retrollamada (Callback) con Promesas, pero de una manera diferente (**encadenamiento**).

## Encadenamiento:

Las funciones **Callback** se han utilizado solas para operaciones asincrónicas en JavaScript durante muchos años. Pero en algunos casos, usar **Promesas** puede ser una mejor opción.

Si hay varias operaciones asincrónicas por hacer y si tratamos de usar retrollamadas antiguas para ellas, nos encontraremos rápidamente dentro de una situación llamada "**Infierno Retrollamada**":

```
firstRequest(function(response) {
    secondRequest(response, function(nextResponse) {
        thirdRequest(nextResponse, function(finalResponse) {
            console.log('Final response: ' + finalResponse);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Sin embargo, si manejamos la misma operación con **Promesas**, ya que podemos adjuntar retrollamadas en lugar de pasárlas, esta vez el mismo código anterior parece mucho más limpio y fácil de leer:

```
firstRequest()
  .then(function(response) {
    return secondRequest(response);
}).then(function(nextResponse) {
  return thirdRequest(nextResponse);
}).then(function(finalResponse) {
  console.log('Final response: ' + finalResponse);
}).catch(failureCallback);
```

El código anterior muestra cómo se pueden encadenar múltiples retrollamadas una tras otra. **El encadenamiento es una de las mejores características de Promesas**.

## Creación y uso de una promesa paso a paso

En primer lugar, usamos un constructor para crear un objeto **Promesa**:

```
const myPromise = new Promise();
```

Se necesitan dos parámetros, uno para el éxito (**resolver**) y otro para el error (**rechazar**):

```
const myPromise = new Promise((resolve, reject) => {
  // condition
```

```
});
```

Finalmente, habrá una condición. Si se cumple la condición, la Promesa se resolverá, de lo contrario será rechazada:

```
const myPromise = new Promise((resolve, reject) => {
  let condition;

  if(condition is met) {
    resolve('Promise is resolved successfully.');
  } else {
    reject('Promise is rejected');
  }
});
```

Ya está creada una promesa, vamos a usarla:

## then( ) para promesas resueltas:

En la imagen del principio se observa que hay 2 casos: uno para promesas resueltas y otro para rechazadas. Si la Promesa se resuelve (caso de éxito), algo sucederá después (depende de lo que hagamos con la Promesa exitosa).

```
myPromise.then();
```

Se llama al método then( ) después de que se resuelva la Promesa . Entonces podemos decidir qué hacer con la Promesa resuelta.

Por ejemplo, registremos el mensaje en la consola lo que obtuvimos de la Promesa:

```
myPromise.then((message) => {
  console.log(message);
});
```

## catch( ) para Promesas rechazadas:

Sin embargo, el método then( ) es solo para Promesas resueltas. ¿Qué pasa si la Promesa falla? Entonces, necesitamos usar el método catch( ).

Del mismo modo que adjuntamos el método then( ). También podemos adjuntar directamente el método catch() justo después de then():

```
myPromise.then((message) => {
```

```
    console.log(message);
}).catch((message) => {
    console.log(message);
});
```

Entonces si la promesa es rechazada, saltará al método `catch()` y esta vez veremos un mensaje diferente en la consola.

Traducido y adaptado del artículo de Cem Eygi - JavaScript Promise Tutorial: Resolve, Reject, and Chaining in JS and ES6.

## JavaScript - Promesas ejemplos

### Promesas, ejemplo en función mitad(numerito)

```
ls script.js >  mitad
1  function mitad (numerito){
2    return new Promise((resolver, rechazar) => {
3      if(numerito % 2 === 0) {
4        resolver(numerito /2);
5      } else {
6        //rechazar(); //la consola avisa de 'undefined'
7        rechazar('error, el número no es par'); //es mejor pasar mensaje aclaratorio
8      }
9    });
10 }
11
12 mitad(4).then(console.log);
13 mitad(3).then();
14 mitad(6).then(console.log);
15 mitad(3).catch(nopar => console.log(nopar));
16 mitad(3).then(console.log);
17 mitad(28).then(console.log);
--
```

```
2
3
error, el número no es par          script.js:15
14
✖ ▾ Uncaught (in promise) error, el número no es par          index.html:1
  Promise.then (asíncrono)
  (anónimo)          @ script.js:13
✖ ▾ Uncaught (in promise) error, el número no es par          index.html:1
  Promise.then (asíncrono)
  (anónimo)          @ script.js:16
```

Se observa que las líneas 12, 14 y 17 al llamar a la función con número par, se devuelve la mitad (2, 3 y 14 en la consola)

Al llamar con then() a la función, y no cumplir la promesa, las líneas 13 y 16 dan error Uncought. Aparece fuera de orden (asíncrono).

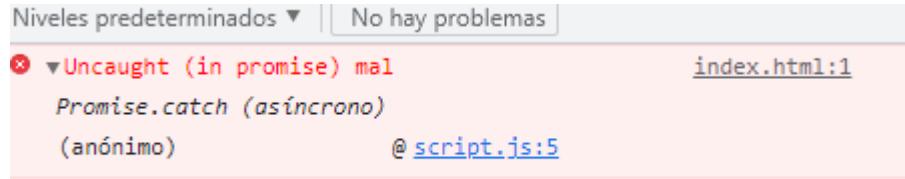
La línea 15 saca el mensaje correctamente al usar el método catch()

## Atributos *resolve* y *reject*

El objeto Promise tiene dos métodos: resolve y reject

Podemos pasar estos métodos como funciones a constantes (o variables):

```
JS script.js > ...
1 const promesaResuelta = Promise.resolve('hola');
2 const promesaRechazada = Promise.reject('mal');
3
4 promesaResuelta.then();
5 promesaRechazada.catch();
```



&gt;

- Cuando es resuelto (**resolve**) se utiliza el método **then()**
- Cuando es rechazado (**reject**) se utiliza el método **catch()**

El método then() no hace nada, y hay error en consola

El método catch() muestra error en consola porque no se ha atrapado el error.

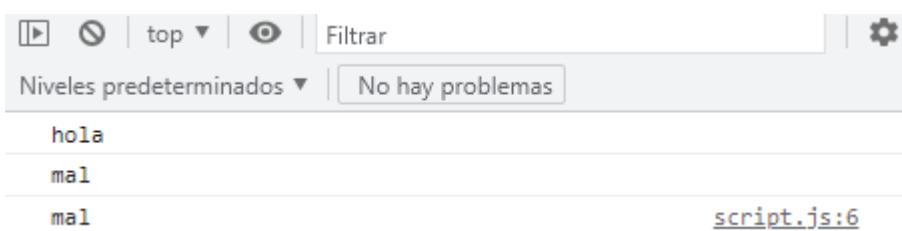
Vamos a resolver ambos casos:

Indicamos una acción para el caso resuelto (línea 4)

Indicamos una acción para el caso rechazado (línea 5)

Capturamos el error y lo sacamos por consola. (línea 6)

```
JS script.js > ...
1 const promesaResuelta = Promise.resolve('hola');
2 const promesaRechazada = Promise.reject('mal');
3
4 promesaResuelta.then(console.log);
5 promesaRechazada.catch(console.log);
6 promesaRechazada.catch((error2)=> console.log(error2));
7
```



Hay una pequeña diferencia en la consola, porque indica que es una salida de la línea 6 del script.

Cosa que no ocurre con la línea 5

## El caso de fetch()

Fetch es una API (interfaz de programación de aplicaciones) para acceder y manipular peticiones y respuestas desde HTTP, devolviendo un objeto Promise (una promesa) que :

- Se resolverá, normalmente. con un estado de ok a true,
- Será rechazado cuando haya un fallo de red o algo que impidiera completar la solicitud

Ejemplo de dos promesas con `fetch()`, una exitosa y otra que no:

```
JS script.js > ...
1 const promesa1 = fetch('http://127.0.0.1:5500/index.html')
2 const promesa2 = fetch('http://127.0.0.1:5500/pepe.html')
3 promesa1.then(respuesta=>console.log(respuesta));
4 promesa2.then(respuesta=>console.log(respuesta));
5
```



## Async

Async: Palabra reservada que colocada delante de una función, nos permite hacer que esa función sea asíncrona.

Async indica a la función que ha de colocar el resultado de la misma dentro de una promesa.

Ejemplo:

```
JS script.js > ...
1 //Async
2 async function saludar(){
3   return 'Hola majo';
4 }
5
6 saludar().then(console.log);
```

Niveles predeterminados ▾ | No hay pi  
Hola majo  
>

También se puede usar `async` tanto en expresiones como en funciones flecha:

Recuerda que hay que llamar a las funciones con los paréntesis.

```
JS script.js > ...
1 //Async
2 async function saludar(){
3   return 'Hola majo';
4 }
5
6 saludar().then(console.log);
7
8 let saludo2 = async () => 'Buenos días';
9 let saludo3 = async () => '¿Qué tal está?';
10 saludo2().then(console.log);
11 saludo3().then(console.log);
```

Niveles predeterminado  
Hola majo  
Buenos días  
¿Qué tal está?  
>

## Await

Hasta ahora no parece que sea de gran utilidad lo que estamos viendo de `async`, pero en conjunción con `await` es otra cosa.

`Await` es colocado delante de una función asíncrona con el objetivo de pausar la ejecución del programa en esa línea hasta que la promesa se resuelva satisfactoriamente y entonces devuelva el valor.

Tenemos una función asíncrona llamada `saludar()` que devuelve un saludo. Pero por ser asíncrona, lo que devuelve es una promesa. Más concretamente, lo que devuelve lo mete en una promesa.

```
$ script.js > ejecutar
1  async function saludar(){
2    return 'Hola majo';
3  }
4
5  let promesa = saludar();
6  console.log(promesa);
7  promesa.then(console.log);
```

Console output:

- Promise {<fulfilled>: 'Hola majo'} script.js:6
- Hola majo

Pero si antecedemos con la palabra await antes de la función

Ahora creamos la función asíncrona ejecutar() y llamamos a la función saludar desde dentro de ejecutar:

```
JS script.js > ...
1  async function saludar(){
2    return 'Hola majo';
3  }
4
5  async function ejecutar (){
6    let promesa = saludar();
7    console.log(promesa);
8    promesa.then('NO HACE NADA');
9  }
10
11 ejecutar();
```

Console output:

- Promise {<fulfilled>: 'Hola majo'} script.js:7
- >

Vamos a hacer un pequeño cambio: ponemos la palabra **await** delante de la llamada a `saludar()`. Para que espere a que se devuelva (no la promesa) si no el valor satisfactorio. Sería más preciso cambiar el nombre de la variable que recoge el valor, porque es un valor, no es una promesa.

```
JS script.js > ⚙ ejecutar
1  async function saludar(){
2    return 'Hola majo';
3  }
4
5  async function ejecutar (){
6    let promesa = await saludar();
7    console.log(promesa);
8    //promesa.then('NO HACE NADA');
9  }
10
11 ejecutar();
```

Niveles predeterminados ▾ | No hay problemas

Hola majo script.js:7

Promesa.then() da error porque no es una promesa.

Ojo!! : await solo se puede poner dentro de Funciones async

Await pausa el programa hasta que la promesa se resuelve. Como no hay operaciones asíncronas, se resuelve inmediatamente.

Await solo detiene esa función, el resto del programa continúa su ejecución. Veamos una prueba:

```
JS script.js > ⚙ ejecutar
1  async function saludar(){
2    return 'Hola majo';
3  }
4
5  async function ejecutar (){
6    let promesa = await saludar();
7    console.log(promesa);
8    console.log('fin de ejecutar');
9  }
10
11 ejecutar();
12 console.log('prueba de parada');
```

prueba de parada script.js:12

Hola majo script.js:7

fin de ejecutar script.js:8

## Fetch (url)

Podemos poner await delante de fetch() porque esta función devuelve una promesa.

```
JS script.js > ⚙ ejecutar
1  /*async function ejecutar(){
2    const respuesta= await fetch('http://127.0.0.1:5500/imagenes/logoiesfse3.png');
3    console.log(respuesta);
4  }
5
6  ejecutar();
7 */
8  async function ejecutar(url){
9    const respuesta= await fetch(url);
10   console.log(respuesta);
11   console.log([respuesta.ok]);
12 }
13
14 ejecutar('http://127.0.0.1:5500/imagenes/logoiesfse3.png'); //recurso existente
15 ejecutar('http://127.0.0.1:5500/imagenes/noexiste.png'); // recurso que NO existe
```

Niveles predeterminados ▾ | No hay problemas

```
script.js:10
▶ Response {type: 'basic', url: 'http://127.0.0.1:5500/imagenes/
  logoiesfse3.png', redirected: false, status: 200, ok: true, ...}
  true
script.js:11
x GET http://127.0.0.1:5500/imagenes/noexiste.png script.js:9
  404 (Not Found)
script.js:10
▶ Response {type: 'basic', url: 'http://127.0.0.1:5500/imagenes/
  noexiste.png', redirected: false, status: 404, ok: false, ...}
  false
script.js:11
```

Observando las respuestas en la consola, el atributo `ok` es el que nos dice si es correcto o no. En este ejemplo no utilizamos este atributo, únicamente para mostrar su valor.

Una respuesta (`Response`) tiene muchos atributos y métodos que podemos observar en consola si desplegamos:

```
script.js:10
Response {type: 'basic', url: 'http://127.0.0.1:5500/imagenes/logoiesfse3.png', redirected: false, status: 200, ok: true, ...} ⚡
  body: (...)

  bodyUsed: false
  ▼ headers: Headers
    ▶ [[Prototype]]: Headers
      ▶ append: f append()
      ▶ delete: f delete()
      ▶ entries: f entries()
      ▶ forEach: f forEach()
      ▶ get: f ()
      ▶ has: f has()
      ▶ keys: f keys()
      ▶ set: f ()
      ▶ values: f values()
      ▶ constructor: f Headers()
      ▶ Symbol(Symbol.iterator): f entries()
      ▶ Symbol(Symbol.toStringTag): "Headers"
      ▶ [[Prototype]]: Object
    ok: true ←
    redirected: false
    status: 200
    statusText: "OK"
    type: "basic"
    url: "http://127.0.0.1:5500/imagenes/logoiesfse3.png"
  ▼ [[Prototype]]: Response
    ▶ arrayBuffer: f arrayBuffer()
    ▶ blob: f blob() ←
    ▶ body: (...)

    ▶ bodyUsed: (...)

    ▶ clone: f clone()
    ▶ formData: f formData()
    ▶ headers: (...)

    ▶ json: f json()
    ▶ ok: (...)

    ▶ redirected: (...)

    ▶ status: (...)

    ▶ statusText: (...)

    ▶ text: f text()
    ▶ type: (...)

    ▶ url: (...)

    ▶ constructor: f Response()
    ▶ Symbol(Symbol.toStringTag): "Response"
    ▶ get body: f body()
```

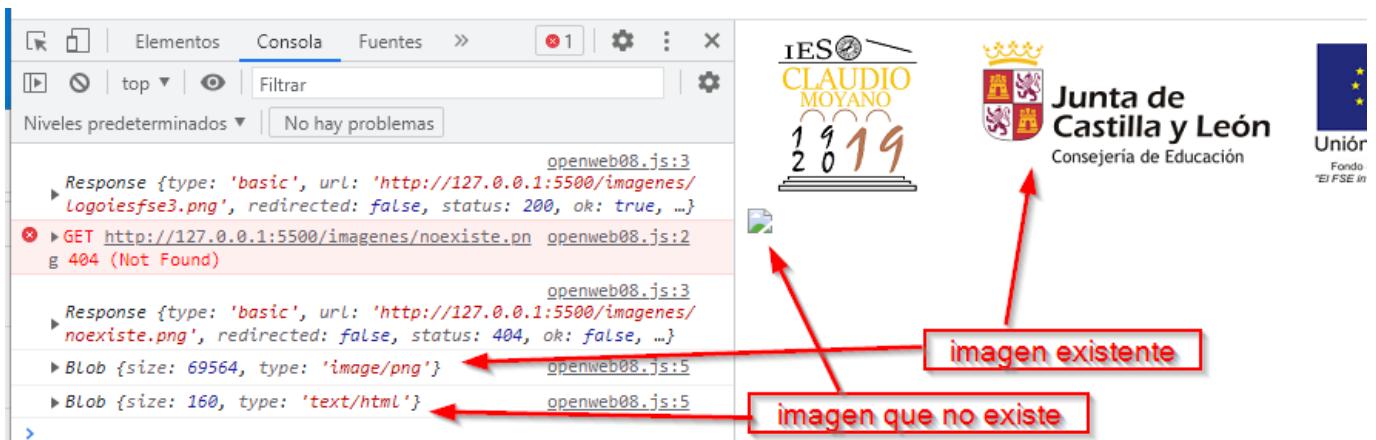
Destacamos el constructor `blob()` de un objeto `Blob`, el cual representa datos inmutables (en el caso que veremos, es un archivo de imagen png).

`respuesta.blob()` obtiene una promesa de fichero binario, con lo que si en la llamada precedemos de `await`, devuelve el valor binario de, en este caso, la imagen.

Agregamos al ejemplo `fetch()` el código para que se visualice la respuesta `fetch()` con el método `blob()`.

Debemos esperar a que el método blob() devuelva el valor (poniendo await delante de la llamada en línea 4) antes de crear el objeto imagen en la línea 7.

```
JS openweb08.js > ⚙ ejecutar > [↻] blob1
1  async function ejecutar(url){
2      const respuesta= await fetch(url);
3      console.log(respuesta);
4      let blob1= await respuesta.blob(respuesta);
5      console.log(blob1);
6      let imagen= new Image();
7      imagen.src=URL.createObjectURL(blob1);
8      document.body.appendChild(imagen);
9
10
11  ejecutar('http://127.0.0.1:5500/imagenes/logoiesfse3.png'); //recurso existente
12  ejecutar('http://127.0.0.1:5500/imagenes/noexiste.png'); // recurso que NO existe
```



Refactorizar (versión 1) de

## Refactorizar código async-await a promesas

Refactorizar es crear un código (a partir de otro existente) que se entienda mejor y que sea más fácil de mantener y escalar.

En este caso tenemos traemos una imagen de una url remota y la insertamos en nuestro documento:

JS script.js &gt; ...

```

1  fetch ('http://127.0.0.1:5500/imagenes/logoiesfse3.png')
2      .then (respuesta => respuesta.blob())
3      .then (blob1 =>{
4          let imagen1 = new Image();
5          imagen1.src = URL.createObjectURL(blob1);
6          return imagen1;
7      })
8      .then(imagen => {
9          document.body.appendChild(imagen);
10     })
11     .catch(ex => console.error(ex)); // tratamiento de la exception
12

```



En el código anterior:

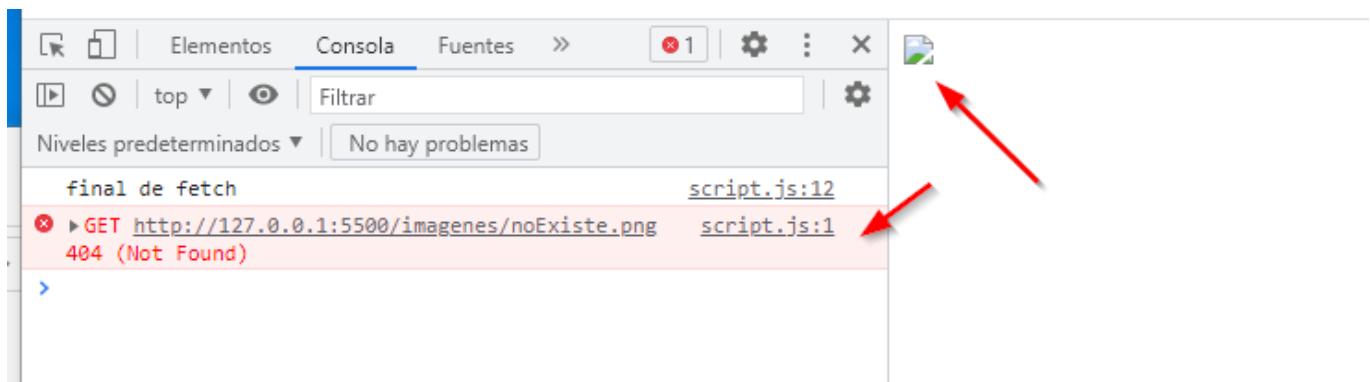
`fetch()` devuelve una promesa que es tratada con `.then()` y `.catch()`

Se pueden poner muchos `.then()` seguidos que se ejecutarán en ese orden (líneas 2, 3 y 8) esperando a que cada uno reciba un valor.

- Fetch devuelve una respuesta tipo promesa que es tratada en la línea 2,
- La línea 2 devuelve el objeto blob que es tratado en la línea 3.
- Al finalizar este segundo `then()` devuelve una imagen que es tratada en el siguiente `then()`, que está en la línea 8.

Podemos añadir `.finally()` para ver que aunque se produzca la excepción, se continua con la ejecución:

```
fetch ('http://127.0.0.1:5500/imagenes/noExiste.png')
  .then (respuesta => respuesta.blob())
  .then (blob1 =>{
    let imagen1 = new Image();
    imagen1.src = URL.createObjectURL(blob1);
    return imagen1;
  })
  .then(imagen => {
    document.body.appendChild(imagen);
  })
  .catch(ex => console.error(ex)) // tratamiento de la exception
  .finally (console.log("final de fetch"));
```



## Refactorizar código con promesas a código async-await

Vamos a duplicar el mismo código en las dos versiones.

```

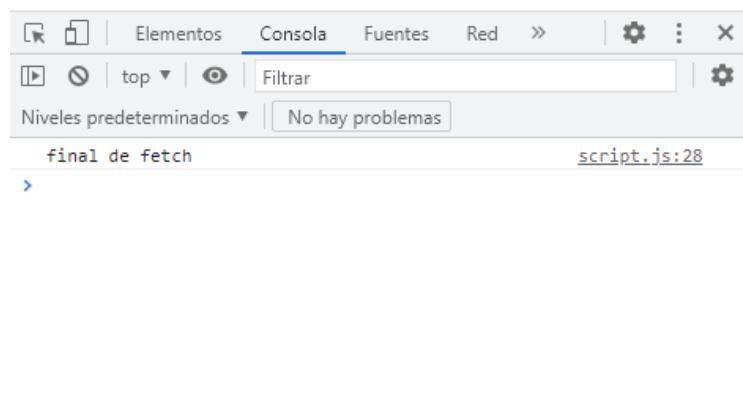
5 script.js > ...
1 // refactorizar
2 const crearImagenDesdeBlob = blob1 =>{
3     let imagen1= new Image();
4     imagen1.src = URL.createObjectURL(blob1);
5     return imagen1;
6 }
7
8 async function imagenFetch(url) {
9     const respuesta = await fetch(url);
10    const blob = await respuesta.blob();
11    const imagen = crearImagenDesdeBlob(blob);
12    document.body.appendChild(imagen);
13 }
14
15 imagenFetch ('http://127.0.0.1:5500/imagenes/logoiesfse3.png');
16

```

```

17 fetch ('http://127.0.0.1:5500/imagenes/logoiesfse3.png')
18 .then (respuesta => respuesta.blob())
19 .then (blob1 =>{
20     let imagen1 = new Image();
21     imagen1.src = URL.createObjectURL(blob1);
22     return imagen1;
23 })
24 .then(imagen => {
25     document.body.appendChild(imagen);
26 })
27 .catch(ex => console.error(ex)) // tratamiento de la exception
28 .finally (console.log("final de fetch"));
29

```



Habrás observado que hemos perdido la gestión de errores con la refactorización 1.

Para resolverlo, hay dos maneras:

1. Se envuelve todo el código que puede contener errores en un bloque try-catch
2. Con promesas

## Control de errores con try-catch

Se envuelve todo el código que puede contener errores en un bloque try-catch

```
JS script.js > ⚙️ imagenFetch
1 // refactorizar
2 const crearImagenDesdeBlob = blob1 =>{
3     let imagen1= new Image();
4     imagen1.src = URL.createObjectURL(blob1);
5     return imagen1;
6 }
7
8 async function imagenFetch(url) {
9     try{
10         const respuesta = await fetch(url);
11         const blob = await respuesta.blob();
12         const imagen = crearImagenDesdeBlob(blob);
13         document.body.appendChild(imagen);
14     }
15     catch (ex) {
16         console.error(ex);
17     }
18 }
```

## Control de errores con promesas

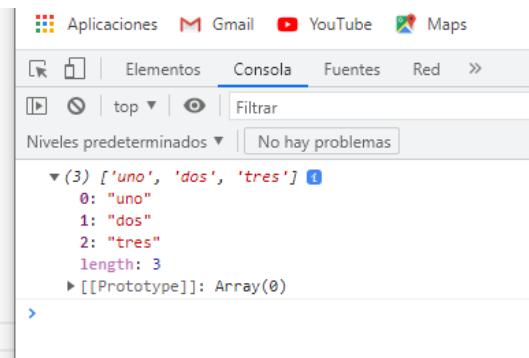
Como async hace que se devuelva el resultado de una función en una promesa, usaremos el catch para estas promesas, añadiendo en la llamada el tratamiento para esas excepciones en el catch()

```
JS script.js > ...
1 // refactorizar
2 const crearImagenDesdeBlob = blob1 =>{
3     let imagen1= new Image();
4     imagen1.src = URL.createObjectURL(blob1);
5     return imagen1;
6 }
7
8 async function imagenFetch(url) {
9     const respuesta = await fetch(url);
10    const blob = await respuesta.blob();
11    const imagen = crearImagenDesdeBlob(blob);
12    document.body.appendChild(imagen);
13 }
14 imagenFetch('http://127.0.0.1:5500/imagenes/logoiesfse3.png').catch(ex => console.error(ex));
```

Podríamos haber añadido a `imagenFetch(...).then()` para seguir encadenando con otras promesas.

## Await delante de Promise.all

**Promise.all:** Es un array de promesas que devolverá un array con los resultados de todas esas promesas cuando todas se hayan cumplido. Evita la necesidad de encadenar promesas.



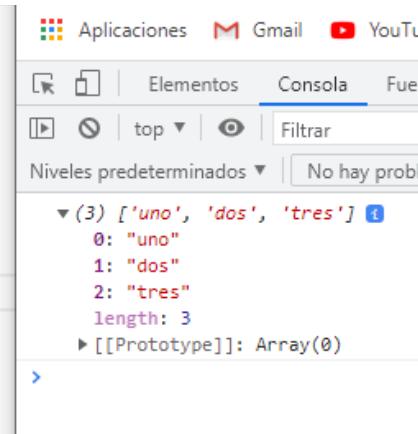
The screenshot shows a browser's developer tools console tab. It displays the result of a JavaScript execution. The code in the editor is:

```
js script.js > ...
1 const p1= Promise.resolve('uno');
2 const p2= Promise.resolve('dos');
3 const p3= Promise.resolve('tres');
4
5 const ejecutar = async () => {
6     let promesaCompleta = Promise.all([p1, p2, p3]);
7     promesaCompleta.then(console.log);
8 }
9
10 ejecutar();
```

The console output shows the resolved values from the promises:

```
(3) ['uno', 'dos', 'tres']
  0: "uno"
  1: "dos"
  2: "tres"
  length: 3
  [[Prototype]]: Array(0)
```

Vamos a aprovechar esto para poner `await` y recibir los valores directamente:



The screenshot shows a browser's developer tools console tab. It displays the result of a JavaScript execution. The code in the editor is:

```
js script.js > [e] ejecutar
1 const p1= Promise.resolve('uno');
2 const p2= Promise.resolve('dos');
3 const p3= Promise.resolve('tres');
4
5 const ejecutar = async () => {
6     let valores = await Promise.all([p1, p2, p3]);
7     console.log(valores);
8 }
9
10 ejecutar();
```

The console output shows the resolved values from the promises:

```
(3) ['uno', 'dos', 'tres']
  0: "uno"
  1: "dos"
  2: "tres"
  length: 3
  [[Prototype]]: Array(0)
```

## Await delante de setTimeout

- (1) Hacemos 3 promesas p1, p2 y p3 que las genera una función que llamamos `espera`.
- (2) `espera(tiempo, valor) =>` después del tiempo especificado devuelve una promesa (que dentro tiene el valor).
- (3) Cuando pase el tiempo especificado se llama a `resolver` con el valor que se le pasa, que es lo mismo que meter ese valor dentro de una promesa.
- (4) `await Promise.all()` hace que se espere a que todas las promesas del array se resuelvan satisfactoriamente. Si no usáramos `Promise.all` habría que coger todas las promesas y encadenarlas.
- (5) Se para la ejecución de la función en esa línea hasta que no pasen los segundos especificados. Pasado ese tiempo muestra los valores. ¿Cuánto es ese tiempo? El tiempo de espera máximo. En este ejemplo serán 5000 milisegundos. **Atención:** Como este código con `async await` parece visualmente más síncrono, puede ocasionarnos sorpresa si las tareas tardan mucho (como una descarga de vídeo), ya que el resto de la función asíncrona está detenida. No ocurriría lo mismo con `promesas`, en las que todo es asíncrono, no se paraliza nada, simplemente el motor JavaScript va llamando a los `Callbakc` de los `Then()` o los `catch()` cuando todo vaya ocurriendo.

(6) Comprobamos que solo se detiene la función asíncrona.

(7) Final de la función asíncrona

JS script.js > [ej] p3

```

1  const espera = (tiempo, valor) => new Promise( 2
2    |   (resolver)=>{setTimeout(() => resolver(valor), tiempo), 3
3    |   (rechazar) => {console.log('nada')} // no hace falta para el ejemplo
4  })
5
6  const p1= espera(3000,'uno'); 1
7  const p2= espera(5000,'dos');
8  const p3= espera(4000,'tres');
9
10 /*const p1= Promise.resolve('uno');
11 const p2= Promise.resolve('dos');
12 const p3= Promise.resolve('tres');
13 */
14 const ejecutar = async () => {
15  let valores = await Promise.all([p1, p2, p3]); 5
16  console.log(valores);
17  console.log('después'); 7
18 }
19
20 ejecutar();
21 console.log('antes'); 6

```

antes	6	script.js:21
▶ (3) ['uno', 'dos', 'tres']	5	script.js:16
después	7	script.js:17

**Promise.all** espera hasta que todas las promesas hayan finalizado, y cuando están todas (ha tardado la suma de todos los tiempos de espera), entonces asigna el valor a la variable **valores**.

En este caso no hay que concatenar las promesas.

**Nota:** El uso de **await** hace que el código sea “más síncrono” y que sea más fácil, pero hay que tener en cuenta que, si usamos **await** en llamadas a servicios que pueden tardar mucho, esas funciones estarán paradas mucho tiempo.

Con promesas todo es asíncrono, no se paraliza nada, el motor de JS va llamando a los **callback** de los **then()** o **catch()** cuando todo vaya ocurriendo.

Aunque **await** simplifica el uso de **promesas**, al poner delante un **await** estamos pausando la ejecución de esa función. El resto del código se sigue ejecutando. Veamos en el ejemplo:

## Resumen

- Las promesas son cajas que, si las desempaquetamos, obtenemos el valor.
- Hemos metido promesas en una función que aparentemente no era asíncrona (la función mitad)
- Hemos usado Promise.resolve() y Promise.reject() para obtener una promesa directamente.
- Async delante de cualquier función hace que el resultado de esta función lo mete dentro de una promesa
- Async con Await hace que el código sea más simple.
- Await hace esperar o pausar la ejecución de esa función asíncrona hasta obtener el resultado de la promesa.
- Hemos pasado código inicialmente programado con promesas a código con async-await, sin olvidarnos del control de errores.
- Hemos usado Promise.all en conjunción con async-await (que en realidad funcionan con promesas)

**Concluyendo:** las **promesas** y **async await** resuelven la asincronía de distinta forma. Con las promesas no sabemos cuándo se va a resolver y con async await forzamos una espera en la función.

Unas veces será mejor utilizar un método y otras veces, otro.