

Javascript

INTRODUCCION JAVASCRIPT

Javascript es un lenguaje de programación creado por la empresa **Netscape** en **1995**. Es un **lenguaje de programación interpretado** a través de un navegador en el lado del cliente. Un lenguaje interpretado significa que las instrucciones las analiza y las procesa un navegador en el momento de ser ejecutadas.

Para aprender Javascript es imprescindible conocer previamente los otros dos “lenguajes” (no se consideran como tal aunque se utiliza de programación del lado cliente que conforman las capas para el desarrollo de páginas web:

- **HTML:** Para definir la capa de contenido y estructura de las páginas web.
- **CSS:** Para definir la capa de diseño o presentación de las páginas web.
- **Javascript:** Lenguaje para programar el comportamiento de las páginas web desde el cliente.

Javascript NO es Java, aunque el nombre sea muy similar. Se consideran lenguajes totalmente diferentes. Todos los lenguajes de programación comparten una serie de conceptos básicos como:

- Lenguaje de programación: Son una serie de instrucciones que interpreta el ordenador para realizar determinadas tareas. Cada lenguaje de **programación tiene su propia sintaxis**, su estructura y su forma de ser ejecutado. [IDIOMA].
- Algoritmo: A través del lenguaje de programación creamos algoritmos o rutinas que son los **pasos a seguir para resolver un problema**. Un algoritmo puede estar compuesto por varios algoritmos sin necesidad de ser un programa. [FRASE].
- Programa: Son un conjunto de instrucciones organizadas con la finalidad de resolver problemas en las aplicaciones. Cada problema se resuelve mediante su correspondiente algoritmo. Un **programa contiene miles de algoritmos interrelacionados** entre sí. [LIBRO]

Javascript fue creado por **Brendan Eich** en 1995 y se convirtió en un estándar en 1997. Es un lenguaje que se encuentra en constante evolución. Actualmente **ECMA-262** es el nombre oficial de la norma y ECMAScript es el nombre oficial del lenguaje. La versión **ECMAScript 5** (Junio 2011) es la más aceptada y asimilada por todos los navegadores. En la actualidad existen otras evoluciones como **ECMAScript 6** (Junio 2015) y **ECMAScript 7** (en desarrollo) que no son 100% compatibles con todos los navegadores modernos. La última versión publicada es de Junio 2019 y por convenio se actualiza anualmente.

Ed.	Fecha	Nombre formal / informal	Cambios significativos
1	JUN/1997	ECMAScript 1997 (ES1)	Primera edición
2	JUN/1998	ECMAScript 1998 (ES2)	Cambios leves
3	DIC/1999	ECMAScript 1999 (ES3)	RegExp, try/catch, etc...
4	AGO/2008	ECMAScript 2008 (ES4)	Versión abandonada.
5	DIC/2009	ECMAScript 2009 (ES5)	Strict mode, JSON, etc...
5.1	DIC/2011	ECMAScript 2011 (ES5.1)	Cambios leves

A partir del año 2015, se marcó un antes y un después en el mundo de Javascript, estableciendo una serie de cambios que lo transformarían en un lenguaje moderno, partiendo desde la especificación de dicho año, hasta la actualidad:

Ed.	Fecha	Nombre formal / informal	Cambios significativos
6	JUN/2015	ECMAScript 2015 (ES6)	Clases, módulos, generadores, hashmaps, sets, for of, proxies...
7	JUN/2016	ECMAScript 2016	Array includes(), Exponenciación **
8	JUN/2017	ECMAScript 2017	Async/await
9	JUN/2018	ECMAScript 2018	Rest/Spread operator, Promise.finally()...
10	JUN/2019	ECMAScript 2019	Flat functions, trimStart(), errores optionales en catch...
11	JUN/2020	ECMAScript 2020	Dynamic imports, BigInt, Promise.allSettled

Características importantes de JavaScript:

- **Javascript NO es Java**, aunque la similitud de los nombres genera muchas confusiones.
 - Java es un lenguaje de programación de bajo nivel (como C, C++, Delphi) y de propósito general, es decir, se pueden crear programas en Java para ser ejecutados en servidores, en equipos locales de sobremesa, en Tablets, en Smartphones o en navegadores (Applets). Java es un **lenguaje compilado**, es decir, se transforma a código máquina (binario) y empaquetado en un archivo ejecutable previo a su ejecución. Con Java se crean **aplicaciones nativas** específicas para cada uno de los sistemas operativos.
 - Javascript es un lenguaje de programación que **se ejecuta exclusivamente en un navegador web** (host), **multiplataforma, orientado a objeto** y ha sido diseñado únicamente para que se ejecute en el navegador. JS es un **lenguaje interpretado**, es decir, se compila cuando se ejecuta por medio de un programa que reside en la máquina donde se ejecuta el código fuente. El programa es el navegador web. Con Javascript se pueden crear **aplicaciones web y aplicaciones híbridas**.
- JS fue diseñado para **añadir interactividad y efectos visuales a las páginas web**. La estructura de una web la aporta HTML y el diseño el CSS. El estándar recomienda tenerlo todo bien separado.
- JS es un **lenguaje del lado cliente** para el diseño de páginas y aplicaciones web en internet y se ejecuta en el equipo local ganando en velocidad de respuesta.
- JS es un lenguaje **interpretado embebido** (incrustado) en el código de una página web HTML. Dicho de otra forma JS requiere de un intérprete en el navegador para poder ser ejecutado. Además no todos los navegadores utilizan el mismo intérprete por lo que puede haber pequeñas diferencias entre unos y otros.
- JS es un lenguaje de secuencias de comandos que puede **reaccionar a eventos**. Un evento es una acción como, por ejemplo, un clic de ratón, un movimiento de ratón, pulsar una tecla, un giro sobre una tableta, etc...
- JS puede **leer y modificar el contenido de un elemento HTML**.

- JS se utiliza para **la validación de contenido en los formularios** antes de ser enviados al servidor. Se consigue acelerar los tiempos de respuesta y no cargar al servidor con más tareas. Todo y con eso el estándar HTML5 de formulario ya realiza validaciones de forma automática.
- JavaScript **puede utilizarse para crear cookies**. Una cookie se utiliza para almacenar y recuperar información del usuario cuando está navegando. La finalidad de la cookie es acelerar el proceso de carga de la web como por ejemplo al llenar la información de un formulario que previamente ya ha sido llenado.
- Cualquier persona puede utilizar JavaScript **sin necesidad de adquirir una licencia**. Es un lenguaje gratuito y por eso se ha expandido tanto en los últimos años. Además, la tendencia actual es generar aplicaciones web universales y JS facilita la creación de software ya que permite la visualización multiplataforma.

Software de ejemplo (editores de pago y gratuito) para programar en JavaScript

De pago (profesionales)	Gratis
Dreamweaver CS5, CS6, CC.	Sublime Text (*)
Microsoft FrontPage.	Visual Studio Code (*)
Adobe PageMill.	Atom (*)
CutePage.	NetBeans
EditPlus	Notepad ++

* son los editores que "mayoritariamente" utilizan los programadores web.

Algunos navegadores actuales han sido diseñados para facilitar la tarea de creación a los desarrolladores web. Por este motivo los grandes navegadores tienen diferentes versiones para el desarrollo web como Google o Firefox que disponen de versiones específicas de sus navegadores para los desarrolladores como **Google Chrome Canary** y **Firefox Developer** respectivamente.



Estas versiones adaptadas tienen sus pros y contras.

- **Pros:** Se actualizan a diario y tienen funcionalidades que no están en las versiones más habituales. **Se pueden probar las nuevas características de los navegadores** sobre nuestras aplicaciones. Además, está disponible para un menor número de sistemas operativos.
- **Contras:** Son inestables y **pueden dejar de funcionar sin motivo aparente**. Las actualizaciones las añaden y las quitan sin previo aviso. El resultado de la aplicación puede no visualizarse como uno desearía debido a estas actualizaciones.



PARADIGMAS DE LA PROGRAMACION

Un paradigma en programación es un estilo de desarrollo de programas, es decir, **un modelo para resolver problemas computacionales**. Los lenguajes de programación, necesariamente, se encuadran en uno o varios paradigmas a la vez a partir del tipo de órdenes que permiten implementar, algo que tiene una relación directa con su sintaxis.

- **Programación Imperativa.** Los programas se componen de un conjunto de **sentencias una detrás de otra que cambian su estado**. Son secuencias de comandos que ordenan acciones a la computadora de forma secuencial, es decir una tras otra. El famoso GOTO que generaba 'código espagueti' surgió con este tipo de programación. A raíz de esta forma de programar se han desarrollado nuevos sistemas:
 - **Programación Estructurada:** El programador se centra en el algoritmo para la resolución de un problema. En este paradigma **se trabajan con funciones** que reciben recursos (parámetros) y devolviendo valores después de ser procesados.
 - **Programación Modular:** Surge a raíz que cada vez los programas son más extensos y el código se convierte ilegible debido a la gran cantidad de líneas de código. La programación estructurada no consigue gestionar tanto volumen de información. **Los programas están divididos en módulos independientes**, cada uno con un comportamiento bien definido, que se pueden reutilizar en otras aplicaciones.
 - **Programación Orientado a Objetos.** El comportamiento del programa es llevado a cabo por objetos, entidades que representan elementos del problema a resolver y tienen atributos y comportamientos. Los objetos son fácilmente exportables y reutilizables por otros desarrolladores.
 - **Programación dirigida a Eventos.** El flujo del programa está determinado por sucesos externos generados por los usuarios. Por ejemplo, la acción que realiza un usuario cuando hace clic con el mouse, al entrar en un elemento tipo <input>, o al posicionarnos sobre un elemento de la web). Javascript es un lenguaje especialmente diseñado para trabajar con eventos.

➤ **Programación Declarativa.** Opuesto al imperativo. Los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos.

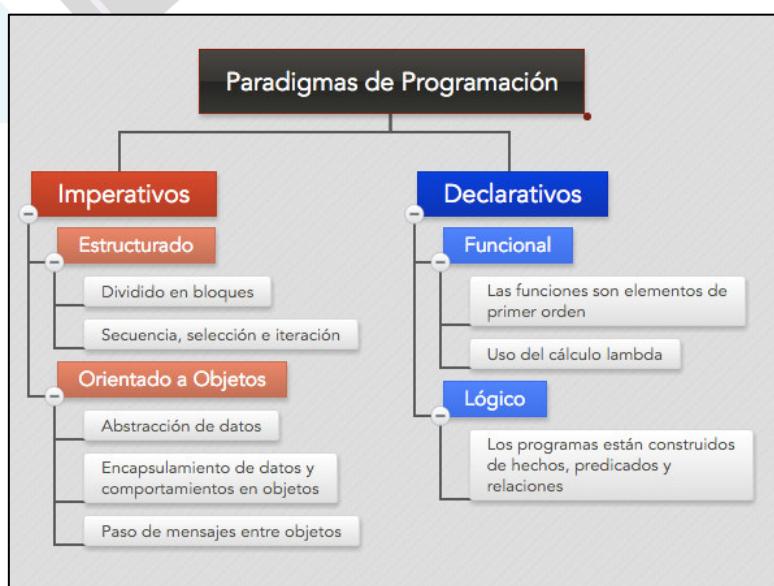
- **Lógico.** El problema se modela con enunciados de lógica de primer orden.
- **Funcional.** Los programas se componen de funciones, es decir, implementaciones de comportamiento que reciben un conjunto de datos de entrada y devuelven un valor de salida.

Javascript es uno de los muchos lenguajes de programación que **acepta trabajar con uno o varios paradigmas a la vez**, es más, a la hora de programar se utilizan varios paradigmas.

No hay un paradigma mejor que otro, pero sí que un **paradigma puede ser más adecuado para solucionar un tipo de problema que otro**. El programador a base de experiencia podrá decidir en cada momento cual es el paradigma que más le conviene en cada caso.

Cabe destacar que tanto en lenguajes de alto nivel o bajo nivel el paradigma que utilizan por excelencia debido a las propias características del mismo es el **paradigma orientado a objetos**. Hoy en día más del 95% del software que se crea utiliza el paradigma de la POO en combinación con otros paradigmas como la estructura o modular.

Todos los programadores de aplicaciones **antes de empezar a programar en POO es necesario que dominen otros paradigmas**. La POO es una forma de solucionar problemas más eficientes, pero requiere de un óptimo nivel de programación por parte del programador y un conocimiento exhaustivo del lenguaje y su funcionamiento.



SINTAXIS BÀSICA JAVASCRIPT

Añadiendo código JS dentro de etiquetas <head> o <body>

En HTML un código JavaScript debe ir encerrado entre las etiquetas `<script>` y `</script>`. Es la forma de indicar que empieza código JavaScript. El código JavaScript puede ir dentro de las etiquetas `<head>` o `<body>`, aunque lo más habitual es acceder a él a través de un **enlace externo al archivo** que contiene los procedimientos desde el documento HTML igual que si fuese un archivo CSS.

En los siguientes ejemplos se muestra como el código Javascript puede estar insertado en diferentes zonas del archivo principal .html.

Ejemplo 1 (<head>): En este primer caso el código es leído por el navegador y se ejecuta con la misma carga de la página. Si se opta por este sistema se debe introducir el código JS entre las etiquetas `<script>` para diferenciarlo de cualquier otro tipo de código. Este sistema aunque se sigue utilizando no es el más recomendable según el estándar HTML5 porque no separa los tres elementos principales de una página web (Estructura – Diseño – Interactividad).

```
// El código JS se introduce entre las etiquetas <head>
<html>
  <head>
    <script>
      document.getElementById("caja1").innerHTML = "JavaScript";
    </script>
  </head>
  <body>
    Contenido página web
    </body>
  </html>
```

Ejemplo 2 (<body>): Este sistema de código embebido en el <body> también se ejecuta en el momento que la página se carga. Una de las características más interesantes de Javascript es poder ejecutar código después de haber realizado la carga de la web. Por tanto, aunque este sistema sigue vigente y está permitido el estándar HTML5 no recomiendo su uso porque no separa los tres elementos principales de una página web (Estructura – Diseño – Interactividad).

```
// Código introducido entre las etiquetas <body>

<html>

  <head>
    </head>

  <body>
    <script>
      document.write('Hola a todos, soy yo mismo');

      document.write('<br>'); // Elemento HTML intercalado

      document.write('Programación en JavaScript');
    </script>
  </body>
</html>
```

Ejemplo 3 (llamada a función): Las funciones en JS se pueden declarar en el **<head>** de la web para posteriormente ser invocadas desde nuestro código utilizando el sistema de eventos. En este caso se crea una función que se invocará desde la etiqueta **<body>** mediante un evento. Las funciones y rutinas se suelen incluir en un archivo externo JavaScript con la extensión .js pero también es posible introducirlas dentro de la propia página.

```
// Función JS declarada el <head> se invoca desde HTML con un evento
// La llamada se realiza desde el botón pulsar que está en el <body>

<html>
  <head>
    <script>
      function miFuncion() {
        document.getElementById("caja1").innerHTML = "Cambia
        texto";}
    </script>
  </head>
  <body>
    <button type="button" onclick="miFuncion()"> Pulsar </button>
  </body>
</html>
```

JavaScript en archivo externo con extensión .js

La forma más habitual de enlazar el documento HTML con procedimientos en JS es **mediante hojas externas**, de la misma forma que se realiza con CSS. El enlace con el archivo externo de JavaScript se realiza siempre dentro de las etiquetas <head>. Las ventajas de utilizar hojas externas son:

- Mantiene separado la estructura HTML del código.
- Hace más fácil la lectura y el mantenimiento tanto del HTML como de JavaScript.
- Al estar cargados los archivos .js en caché aceleran notablemente la carga de la página.

Ejemplo 1: Enlace **especificando la URL completa** donde se encuentra el archivo de funciones.

```
<script src="https://www.miweb.com/js/myScript1.js"> </script>
```

Ejemplo 2: Enlace **especificando una ruta** desde el sitio actual. Es una referencia relativa. Los caracteres punto punto barra (..) se utilizan para subir un nivel en la estructura de árbol de las carpetas.

```
<script src="../js/myScript1.js"></script>
```

Ejemplo 3: Enlace de 2 archivos .js se cargan de diferente forma (async, defer)

```
<head>  
  <script src="file_js1.js" async> </script>  
  <script src="file_js2.js" defer> </script>  
</head>
```

async: La página se carga de forma asíncrona, es decir, el código se ejecuta mientras se procesa el resto del documento.

defer: El código .js se ejecuta después de que la página se haya cargado por completo. Solo funciona con scripts externos al HTML.

Es posible también enlazar desde las etiquetas <head> con las **librerías de Javascript**, como por ejemplo JQuery. Una librería es un archivo o conjunto de archivos (también llamado "framework") que se utiliza para facilitar la programación de un website. De esta forma es posible reutilizar código que ya está creado y que funciona correctamente en todos los navegadores.

Técnicamente una librería y un framework no son lo mismo ya que el segundo es un entorno de trabajo mientras que el primero es un conjunto de métodos agrupados bajo el nombre de librería. Para nuestro objetivo por ahora podemos entenderlos como un recurso externo al standard de Javascript.

```
<script src="Jquery/jquery-1.6.3.min.js"> </script>
```

Víctor Pérez
Programador Web & Formador

ALERTAS Y VENTANAS DE DATOS

En JavaScript existen **diferentes formas de enviar o recibir información** al equipo, y de modificar los datos que se muestran de la página web. Los más significativos y utilizados se muestran a continuación:

1. **document.write**: Añade texto en el **<body>** de una página. Borra el contenido de la web (si lo hubiera) y escribe el contenido pasado a través de los parámetros. Es utilizado para testear las aplicaciones, aunque para programadores es más útil 'console.log'. Es un sistema poco utilizado en la actualidad pero para empezar a conocer el lenguaje tiene su utilidad.

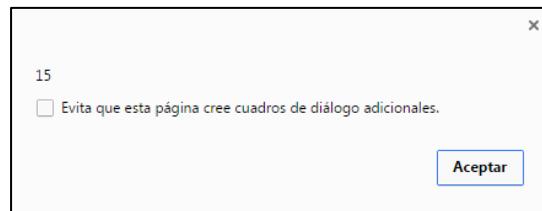
```
// Escribe el número 15 en el navegador directamente
```

```
<script>  
    document.write(5 + 10);  
</script>
```



2. **windows.alert()**: Muestra un mensaje de alerta para mostrar información o datos. Es una ventana modal que se ejecuta en primer plano, detiene el flujo del programa y no deja manipular nada en segundo plano. Se puede escribir sin '**window**' ya que depende del objeto principal dentro de una web que siempre es '**window**'. En las páginas web actuales se intenta no abusar en exceso de este recurso debido al bloqueo que realiza sobre la web.

```
<script>  
    alert(5 + 10);  
</script>
```



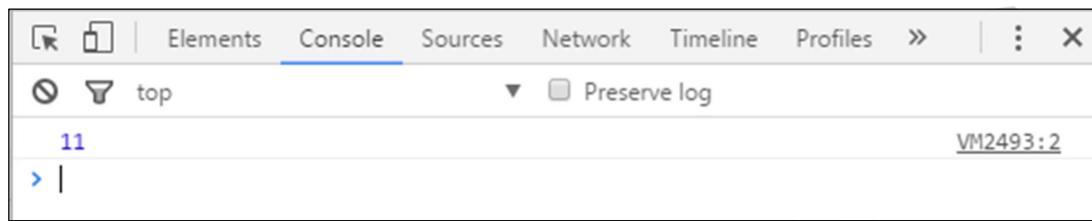
3. **console.log()**: Se utiliza con propósitos de depuración. Crea un objeto del tipo Console() que nos permite imprimir un mensaje en la consola en cualquier momento sin interrumpir la ejecución del código. Es necesario abrir el navegador en modo de consola (tecla F12). Esta herramienta es muy útil para mostrar mensajes cuando se produce un error en la aplicación sin necesidad de mostrar esa información a través de la web.

```
// Muestra el mensaje a través de la consola el navegador
```

```
<script>
```

```
    console.log(5 + 6);
```

```
</script>
```



4. **prompt:** Captura a través de una ventana modal la información que desea introducir el usuario.

El valor capturado se almacena en una variable y se convertirá en otro tipo si fuese preciso ya que el valor recibido será **siempre de tipo texto (string)**.

Si intentamos asignar un valor a una variable que no está creada se mostrará el valor de '*Undefined*' (no definida) como resultado mostrado por pantalla. Si el prompt se cancela el valor devuelto por la ventana será 'null'.

```
<variable que almacena el dato> =  
prompt (<msg a mostrar en pantalla>, <valor inicial por defecto>)
```

```
// 'prompt' captura órdenes por teclado y las guarda en variables.
```

Ejemplo: `nombre = prompt('Cual es tu edad:','20');`

```
<script type="text/javascript">
```

```
    var nombre = prompt('Introduce tu nombre:','Pep');
```

```
    var ciudad = prompt('Introduce tu ciudad:','BCN');
```

```
    document.write('Hola ' + nombre);
```

```
    document.write(' Vivo en ' + ciudad);
```

```
</script>
```

5. **innerHTML**: Escribe en un documento HTML o inserta contenido dentro de una etiqueta HTML.

Esta propiedad permite incrustar etiquetas HTML siempre que se encuentren en formato texto ya que las reconoce y las interpreta como tal.

```
// Escribe el número 10 dentro del elemento HTML especificado
```

```
<script>  
    document.getElementById("resul").innerHTML = 10;  
</script>
```

```
// Escribe el nombre 'Homer' incluyendo código HTML en el mensaje.
```

```
<script>  
    document.getElementById("resul").innerHTML = "<h1>Homer</h1>";  
</script>
```

VARIABLES Y OPERADORES

Los lenguajes de programación como JavaScript (pseudo-lenguajes de alto nivel) ejecutan el código de programación de forma secuencial, es decir, una línea tras otra. La sintaxis de JavaScript no difiere demasiado con otros lenguajes, pero si la forma de escribirlo y representarlo en pantalla.

Las variables son elementos que permiten almacenar información para poder ser utilizada a lo largo de la ejecución del programa. Las variables en **lenguajes dinámicos** como Javascript se adaptan al tipo de valor introducido, es decir, si introducimos un número la variable es de tipo Number, y si introducimos un texto la variable será de tipo String. Además una misma variable puede cambiar de tipo sin verse afectada.

Las variables en los denominados **lenguajes estáticos** (C, Java u otros) se debe especificar que tipo de valor almacenará la variable y no podrá introducirse un valor de otro tipo ya que provocaría un error en tiempo de ejecución.

Los diferentes tipos de variables que se utilizan en Javascript son:

Tipo de dato	Descripción	Ejemplo básico
number	Valor numérico (enteros, decimales, etc...)	42
string	Valor de texto (cadenas de texto, caracteres, etc...)	'MZ'
boolean	Valor booleano (valores verdadero o falso)	true
undefined	Valor sin definir (variable sin inicializar)	undefined
function	Función (función guardada en una variable)	function() {}
object	Objeto (estructura más compleja)	{}

Los lenguajes como Javascript donde no hay que ser tan específico con la declaración de las variables se denominan lenguajes **débilmente tipados**, en cambio, otros lenguajes que son mucho mas estrictos con las declaraciones de los diferentes elementos y variables se consideran **fuertemente tipados**.

Las variables se pueden escribir de diferentes formas en función de la forma seleccionada por el usuario.

No hay una forma mejor que otra pero siempre es recomendable seguir alguno de los sistemas estandarizados entre desarrolladores.

Nombre	Descripción	Ejemplo
camelCase	Primera palabra, minúsculas. El resto en minúsculas, salvo la primera letra. La más utilizada en Javascript.	precioProducto
PascalCase	Ídem a la anterior, pero todas las palabras empiezan con la primera letra mayúscula. Se utiliza en las Clases.	PrecioProducto
snake_case	Las palabras se separan con un guion bajo y se escriben siempre en minúsculas.	precio_producto
kebab-case	Las palabras se separan con un guion normal y se escriben siempre en minúsculas.	precio-producto

Variables

Para poder realizar rutinas o procedimientos es necesario utilizar **variables** que son los elementos que nos permiten **almacenar valores** y realizar cálculos. Algunas características de las variables son:

- Las variables son elementos que permiten **almacenar información de diferente tipo**. Disponen de un espacio de memoria (RAM) reservada en el ordenador, y almacenan valores que podrán cambiar durante la ejecución del programa.
 - Valores enteros (5, 8, 25, etc...). Número solo parte entera.
 - Valores reales (4.21, 8.35, etc...). Número con parte entera y decimal.
 - Cadenas de caracteres ('Juan', 'Pedro', 'Ana'). Las dobles comillas son aceptadas.
 - Valores Lógicos (true o false).
- **Las variables se declaran** escribiendo la palabra reservada **var**, **let**, **const** (siempre en minúsculas) delante del nombre de la variable. En su inicialización, una variable es de un tipo u otro cuando se le asigna un valor. Es recomendable que en la declaración de las variables estas sean inicializadas (darles un valor) para evitar problemas o errores en tiempo de ejecución.

- `var sexo = "mujer"; /* Texto siempre entre dobles comillas */`
- `var talla = 1.72; /* Para poner la coma es un punto */`

- La declaración de variables en Javascript es siempre con las palabras reservadas, 'var', 'let', o 'const'. Otros lenguajes tienen declaraciones específicas si los valores almacenados serán números (int, float, double, etc..) o textos (char, string, etc..).
- Los nombres de las variables deben ser **representativos de su contenido** (nombre, edad, etc...) para realizar un mejor seguimiento del código en modo depuración.
- Los valores de las variables pueden ser **fijos [literales]** o **variables [variables]**. Hay variables que NO modifican su valor y otras variables que lo van modificando con el desarrollo del programa. Las variables fijas llamadas constantes se suelen escribir en mayúsculas para diferenciarlas del resto de las variables del código.
- Las variables comienzan por una letra o por guion bajo (_) o un signo dólar \$, **nunca por un número** ya que la podría confundir por un valor numérico.
- Las palabras propiamente reservadas por el lenguaje **no se pueden utilizar como nombres de variables**. Por ejemplo: var, break, if, while, case, for, function, foreach, etc... no serían validas.
- La variable **debe estar inicializada con un valor** en el momento de su declaración. Si no se declara esta adopta el valor 'undefined'. Este valor puede provocar problemas en tiempo de ejecución por no estar definida cuando se necesite para realizar operaciones.
 - `var edad; // Si no se inicializa adopta el valor 'undefined'`
 - `var edad = 35; // Inicializada correctamente`
- Los guiones (-) NO se aceptan para declarar nombres de variables. Nombres como 'dia-alta' o 'fecha-nac' no serían válidos.
- Los nombres de variables diferencian entre mayúsculas y minúsculas [**case sensitive**]. En palabras compuestas se recomienda utilizar el sistema [**camel case**] o [**lowe camel case**] donde la primera letra de la palabra compuesta es en mayúsculas, como por ejemplo 'DiaSemana', 'NombrePersona', 'colorCamisa', 'tipoAnimal'.

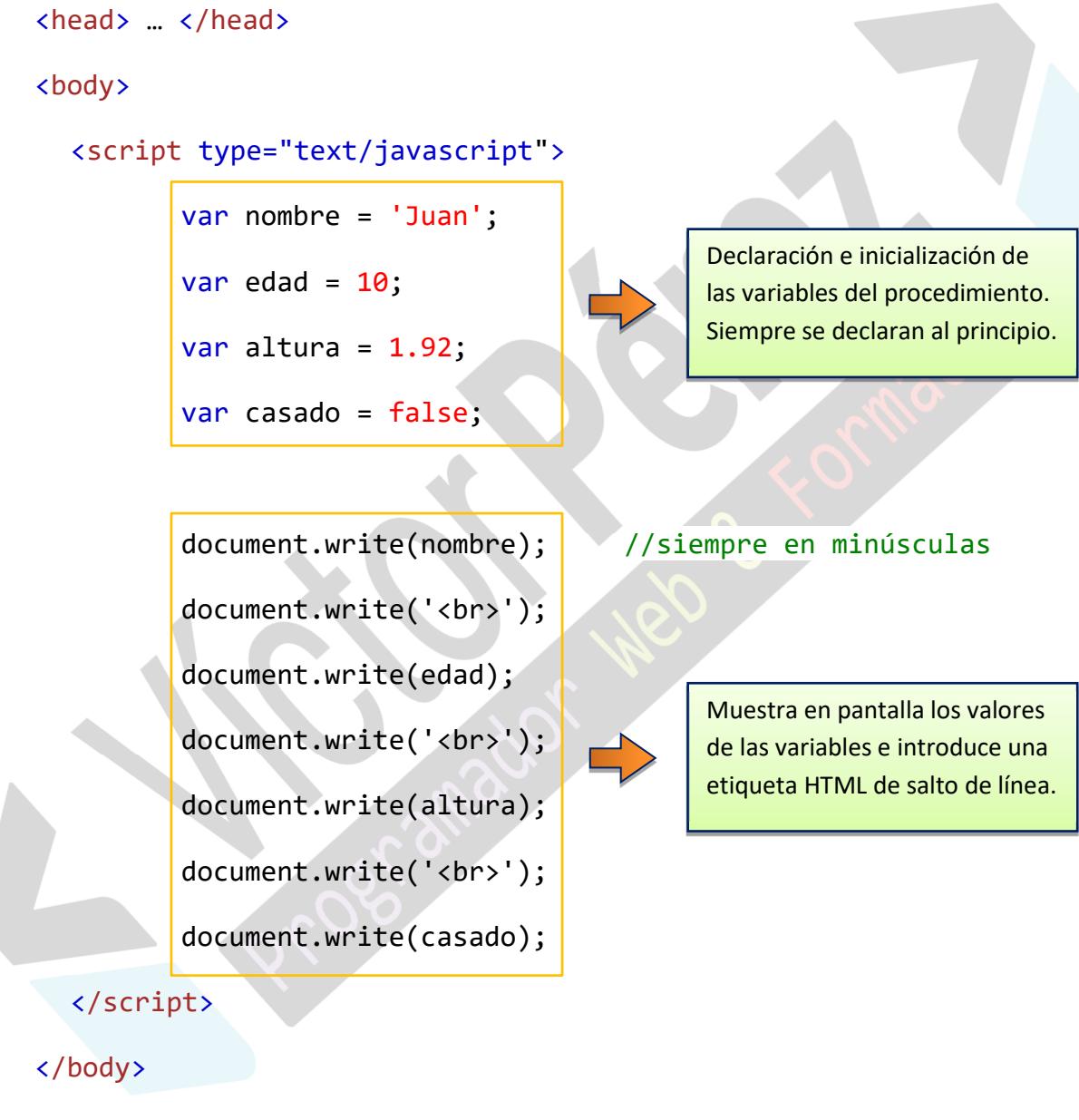
Ejemplo 1: Se crea un script con una declaración de variables que están inicializadas para posteriormente mostrarlas por la pantalla del navegador. En este caso el nombre de la variable dentro del método write no se escribe con comillas simples, porque sino provocaría que se mostrase el nombre de la variable, pero no su valor.

```
<html>
  <head> ... </head>

  <body>
    <script type="text/javascript">
      var nombre = 'Juan';
      var edad = 10;
      var altura = 1.92;
      var casado = false;

      document.write(nombre);
      document.write('<br>');
      document.write(edad);
      document.write('<br>');
      document.write(altura);
      document.write('<br>');
      document.write(casado);

    </script>
  </body>
</html>
```



Declaración e inicialización de las variables del procedimiento. Siempre se declaran al principio.

//siempre en minúsculas

Muestra en pantalla los valores de las variables e introduce una etiqueta HTML de salto de línea.

Ejemplos de declaración de variables

```
var nombreMarca;      // Declaración de la variable

var nombreMarca = "Audi"; // Asigna un valor a variable declarada

var persona = "Pepe", price = 200; // Asignación multivariable 1 línea

document.write(persona); // Mostrar por la pantalla del navegador
```

```
var precio1 = 5;
var precio2 = 6;
var total = precio1 + precio2;
```

```
document.write(total);
```

Las variables 'precio1' y 'precio2' se inicializan y almacenan un valor. La variable 'total' guarda el cálculo para finalmente mostrarlo por pantalla del navegador.

Las variables de escape se utilizan para poder utilizar caracteres reservados como si no lo fuesen o para realizar determinadas acciones específicas sobre el documento JavaScript.

Si se quiere incluir...	Se debe incluir...
Una nueva línea	\n
Devolver cursor inicio línea	\r
Un tabulador	\t
Una comilla simple	\'
Una comilla doble	\"
Una barra inclinada	\\"

El siguiente ejemplo contiene la variable 'dato' que almacena un texto. Al ser mostrado a través de la ventana del navegador, la línea se convierte en dos debido a las variables de escape que contiene dentro de la declaración fuerzan el ajuste de línea.

```
var dato = "El siguiente texto \r\n se escribirá en dos líneas";
```

Ámbito de las variables

Las variables, igual que en todos los lenguajes de programación, tienen un **ámbito de acción (scope)**. El **ámbito de las variables se determina en el momento de su declaración**. Por tanto, es posible disponer de los siguientes tipos:

Variables locales

Las variables declaradas dentro de una función se consideran locales y solo se puede acceder a ellas desde dentro de la misma función. Es posible utilizar variables con el mismo nombre en diferentes funciones. Se eliminan de memoria cuando se dejan de utilizar. Si no se utiliza el prefijo '**var**', '**let**' o '**const**' para la declaración de la variable esta se convertirá en una variable global.

```
function miFuncion() {  
    var coche = "Volvo"; // Solo afecta dentro de la función  
}  
miFuncion(); // Utiliza la variable local
```

Variables globales

Las variables globales están declaradas fuera de las funciones de JavaScript. Esto significa que todas las funciones y procedimientos pueden acceder a esta variable y manipularla. En la POO no se recomienda en absoluto trabajar utilizando variables globales.

```
<script>  
    var coche = "Volvo"; // Variable global declarada externamente  
    myFunction(); // Llamada a la función  
  
    function myFunction() {  
        document.getElementById("demo").innerHTML =  
            "El coche es" + coche; // Utiliza la variable global  
    }  
</script>
```

Las variables que no se hayan declarado correctamente se convierten de forma automática en variables '*globales*' incluso aun estando dentro de una función. En HTML todas las variables globales pertenecen al objeto ventana (**window**).

```
function miFuncion() {  
    coche = "Volvo"; // Variable mal declarada, ahora es global  
}  
  
window.coche = "Mercedes";
```

Si dentro de una función declaramos una variable sin introducir el prefijo '**var**' entonces se convierte en una variable global de forma automática que se puede acceder desde cualquier parte de la aplicación.

Las variables en Javascript pueden ser de tres tipos en función de su ámbito de acción (scope).

var: Es de ámbito más genérico adquiriendo la característica de variable global que *puede ser accesible desde cualquier parte del código*. Este tipo de variable se pasa a scopes descendientes o herederos y pueden ser modificadas por las funciones de la aplicación. Una variable de este tipo declarada en una función tiene un scope que afecta exclusivamente a toda la función.

let: Es un tipo de declaración que *afecta exclusivamente al bloque de código donde ha sido declarada*, por ejemplo, un *for*, *while*, *if*, etc... Una vez se sale del bloque de código la variable desaparece liberando espacio en memoria. Las variables de este tipo al ser declaradas no se ven afectadas por el hoisting (alzamiento en la declaración) como le sucede con las de tipo 'var'.

const: Este tipo de declaración *NO permite la reasignación de valores*, es decir, es una variable de solo lectura. Se acostumbran a escribir en mayúsculas para diferenciarlos de los otros tipos de variables. Existe la posibilidad de cambiar el valor de la variable si este formase parte de un objeto, pero es un caso que sale de la finalidad de esta variable. Las variables de tipo 'const' no se ven afectadas por el hoisting.

Tipos de datos

Los valores en JS se almacenan siempre en variables mediante una de las palabras reservadas '**var**', '**let**' o '**const**'. Cada variable puede almacenar un tipo diferente de valor y los más habituales son:

```
var edad = 16;           // Número entero
var descuento = 15.25;    // Número con decimales
var nombre = "Pep";      // String o cadena de texto
var x = true;            // Booleano verdadero, no se pone entre comillas
var x = false;           // Booleano falso. True y false ya son valores
var cars = ["Saab", "Volvo", "BMW"]; // Array o matriz de valores

// Objeto que almacena 4 propiedades.
var person = { Nombre:"Pep", Ape1:"Lopez", Edad:50, Sexo:"H" };

var person;              // Devuelve 'undefined', sin valor y sin tipo
var person="";           // Valor vacío, no tiene nada que ver con 'undefined'
var person = null;        // Valor nulo, pero tipo sigue siendo un objeto
```

Función `typeof()`: Es un método global de JavaScript que **permite averiguar el tipo de dato** que almacena una variable. En muchos casos `typeof()` resulta insuficiente porque trabajamos con objetos o elementos más avanzados. En estos casos la sentencia `constructor.name` nos proporciona el tipo de constructor que utiliza.

```
typeof "John"           // Devuelve tipo "string"
typeof 3.14              // Devuelve tipo "number"
typeof true               // Devuelve tipo "boolean"
typeof [1,2,3,4]          // Devuelve tipo "object"
typeof {nombre:'John', edad:34} // Devuelve tipo "object"
typeof function myFunc(){} // Devuelve tipo "function"

console.log(nomPersona.constructor.name) // Devuelve tipo "string"
```

Función `instanceOf()`: Indica si un valor es una instancia a un tipo de objeto.

Comentarios

En JavaScript es posible **comentar líneas de código o bloques de código**. Para comentar líneas se realiza con la doble contra barra mientras que para comentar un bloque se realiza con asterisco-contra barra.

Los **comentarios son imprescindibles** para explicar el código que se está realizando. Esto es necesario para poder leer y entender el código que se ha realizado en otro momento.

```
var x = 5;          // Comentario para una línea (Comentario de línea)
```

```
/* Párrafo comentado que no se ejecutará (Comentario de bloque)
document.getElementById("myH").innerHTML = "My First Page";*/
```

Operadores

Son los símbolos o caracteres que permiten realizar operaciones matemáticas. Existen diferentes tipos de operadores en función del uso.

- Operadores aritméticos

Para realizar operaciones matemáticas básicas.

Operador	Descripción
+	Suma (o concatena)
-	Resta
*	Multiplicación
**	Potencia (ES6)
/	División
%	Resto división (módulo)
++	Incrementar en 1
--	Decrementar en 1

Los **operadores tienen prioridad** de unos respecto a otros. La multiplicación y la división tienen preferencia respecto a la suma y la resta. En caso de igualdad de prioridad entre dos operadores se ejecutan de izquierda a derecha.

```
var x = 5, var y = 2;    // Declaración múltiple misma línea.  

var z = x + y;          // Suma, return 7  

var m = x * y;          // Multiplicación, return 10  

var x++;                // Incremento de 1, return 6
```

- Operadores de asignación

Para dar valor a las diferentes variables.

Operador	Ejemplo (método abreviado)	Lo mismo que... (método expandido)
=	x = y	x = y (asignación)
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Ejemplos

```
var x = 10;
x += 5;
```

Se asigna el valor 10 a la variable 'x' y se le suma 5. Resultado final 15.

```
x = 5 + 5; // Asigna la suma como valor numérico (55)
y = "5" + 5; // Si hay texto el cálculo devuelve texto ('55')
z = "Hello" + 5; // Texto y numero devuelve texto ('hello5')
```

```
var txt1 = "Programación";
```

```
var txt2 = "JavaScript";
```

```
var txt3 = txt1 + " " + txt2; // Concatena dos textos con espacio
```

```
txt1 += "en JavaScript"; //Concatena texto con operador asignación
```

```
var x = 10;
```

```
x *= 5; // El resultado final será 50 (10*5)
```

```
var x = 10;
```

```
x /= 5; // Devuelve la parte entera de la división (10/5)=2
```

- Operadores de comparación

Para realizar comparaciones entre variables.

Operador	Descripción
<code>==</code>	Igual a
<code>====</code>	Igual valor y igual tipo
<code>!=</code>	Diferente (no igual)
<code>!==</code>	Diferente valor o diferente tipo
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual que
<code><=</code>	Menor o igual que
<code>?</code>	Condicional abreviado (op. ternario)

Ejemplos

```

var x = 10;           // Declaración de la variable.

x == "10";           // Devuelve True, mismo valor sin valorar tipo.

x > 20;              // Devuelve False

x != 20;              // Devuelve True

x >= 10;              // Devuelve True

x < 10;               // Devuelve False

x !== "10";            // Devuelve False, mismo valor diferente tipo.

x === 20;              // Devuelve False, diferente valor mismo tipo.
  
```

- Operadores de lógicos

Operador	Descripción
&&	Condicional Y. Todas las condiciones se deben cumplir
	Condicional O. Una o más condiciones se deben cumplir
!	Negación. Para negar el resultado de una condición.

// Si las 2 condiciones se cumplen entra en el bucle.

```
if (num1 > num2 && num1 > num3) {
    document.write('el mayor es el '+ num1);
}
```

// Si una de las 2 condiciones o las 2 se cumplen entra en el bucle.

```
if (num1 > num2 || num1 > num3) {
    document.write('El mayor es el '+ num1);
}
```

- Operador ternario

El operador ternario (o condicional) permite realizar una operación condicional de forma más abreviada y aprovechar el resultado para asignárselo a una variable.

La sintaxis del operador ternario siempre devuelve algo que se almacena en una variable:

```
< Condición ? True : False >
```

// En este caso la variable 'resultado' almacena el resultado de la condición utilizando el operador ternario.

```
var resultado = num1 > num2 ? "Num1 es mayor" : "Num2 es mayor";
```



Prueba lógica o
comparación.



Si se cumple
condición Verdadero



Si NO se cumple
condición Falso

USE STRICT

Para no tener problemas con variables locales que se declaran sin querer como globales (entre otros posibles errores) existe la directiva '*use strict*' que obliga, entre otras cosas, a declarar todas las variables dentro de las funciones ya que sino produce un error desde (ES6).

El modo estricto se puede aplicar al código Javascript por completo o a una función. El 'modo estricto' es una variante del lenguaje que es menos permisiva con ciertos comportamientos de tal forma que posibles warnings en modo normal se convierten en errores de Javascript en 'modo estricto'. Los programadores profesionales utilizan el '*strict mode*' para minimizar los posibles errores al programar.

La directiva de modo estricto emplea las palabras claves '***use strict***' entre comillas simples o dobles seguido de punto y coma. Los navegadores modernos implementan los análisis de modo estricto. La razón de disponer la directiva entre comillas y con un punto y coma es para que navegadores antiguos pasen por alto el string.

Ejemplo: El siguiente código tiene declarado antes de empezar a ejecutar ningún código el método '***use strict***'. Este puede estar en un ámbito global o dentro de una función.

De esta forma la variable 'x' que se encuentra declarada dentro de la función **imprimir()** dará como resultado un error en tiempo de ejecución porque ahora es obligatorio declararla.

Sin esta línea de código en modo estricto la variable 'x' simplemente hubiese sido considerada como una variable global y por tanto podría ser accesible desde cualquier zona del código.

```
<script type="text/javascript">  
  "use strict";  
  /* Modo estricto que obliga a declarar todas las */  
  function imprimir(){  
    // Da error, mal declarada, está en modo 'use strict'  
    x = 100;    document.write( x + '<br>' );  
  }  
  imprimir();  document.write( x + '<br>' );  
</script>
```

ESTRUCTURAS DE CONTROL DE FLUJO

Una estructura secuencial es un algoritmo que dispone de un único hilo o camino para ejecutarse. Un ejemplo de estructura secuencial sería pedir datos al usuario y mostrarlos por pantalla. Este algoritmo tiene un único recorrido, por tanto, se considera una estructura secuencial.

Los operadores aritméticos (+ , - , * , / , %) permiten realizar operaciones matemáticas. Los valores capturados por la aplicación se deberán convertir a número ya que por defecto la información es texto. Las funciones que permiten la conversión de texto a número son:

- **parseInt()**: Convierte un texto que representa un número entero a número entero.
- **parseFloat()**: Convierte un texto que representa un número decimal a número decimal.
- **Number()**: Convierte cualquier número (entero o decimal), a valor numérico. Es el método que se recomienda utilizar porque engloba los dos anteriores y además tiene otras funciones extra.

```
<script type="text/javascript">

    var num1, num2; //declaración de las variables

    num1 = prompt('Ingrese primer valor:','');
    num2 = prompt('Ingrese segundo valor:','');

    // Realiza la suma convirtiendo los valores a enteros
    var suma = parseInt(num1)+ parseInt(num2);

    // Realiza el producto convirtiendo valores a números
    var producto = Number(num1) * Number(num2);

    document.write('La suma es: ' + suma);
    document.write('El producto es: ' + producto);

</script>
```

num1 = prompt('Ingrese primer valor:','');
num2 = prompt('Ingrese segundo valor:','');

Solicita valor con
'prompt' y lo guarda
en la variable.

// Realiza la suma convirtiendo los valores a enteros
var suma = parseInt(num1)+ parseInt(num2);
// Realiza el producto convirtiendo valores a números
var producto = Number(num1) * Number(num2);

Escribe el texto y el
contenido de las
variables por pantalla.

IF (Estructura condicional Simple)

Las condiciones son las sentencias mediante las cuales se puede determinar que un proceso realice una acción u otra en función de un criterio establecido.

Este tipo de estructuras se utilizan cuando tenemos más de una solución o camino para ejecutarse nuestra aplicación JavaScript. Un ejemplo condicional puede ser si un alumno ha sacado más de un 5 entonces está aprobado, sino está suspendido.

Para utilizar estructuras condicionales es necesario conocer los operadores de comparación:

- > Mayor.
- >= Mayor o igual.
- < Menor.
- <= Menor o igual.
- != Distinto o Diferente.
- !== Diferente valor y tipo de dato.
- == Igual (de comparación, porque un solo igual es asignación).
- === Igualdad (mismo valor y mismo tipo).

En JavaScript, como en otros lenguajes, aparece la instrucción **if**. Se escribe entre paréntesis y ejecuta el código si se cumple la condición. Además, la función **if** se escribe siempre en minúsculas y tiene llaves de apertura y de cierre. En caso contrario JavaScript devolverá un error en tiempo de ejecución.

El siguiente ejemplo muestra una estructura condicional con IF.

```
<script type="text/javascript">

    var nombre = prompt('Ingrese nombre:', '');

    if (nota >= 5){    // Si la nota es superior o igual a 5.

        document.write(nombre +' está aprobado con un '+ nota);

    }

</script>
```

Las estructuras condicionales compuestas son iguales que las simples solo que se les añade la opción '**else**' (sino). De esta forma si no cumple una condición automáticamente realiza el cálculo que haya dentro de '**else**'. En este tipo de condición se ejecuta una de las dos opciones '**if**' o '**else**' pero en ningún caso se ejecutarán las dos condiciones a la vez. '**else**' también dispone de sus llaves de apertura y cierre.

```
if (<condición>) { <Instrucción(es)> }  
else { <Instrucción(es)> }
```

La estructura **else if** también es válida en este tipo de estructuras y puede repetirse tantas veces como sea necesaria.

```
function calculoNota(){
```

```
    var nota;  
    var resultado;
```

Declaración de las variables.

```
    if (nota < 5) {  
        resultado = "Suspendido";  
    } else if (nota < 8) {  
        resultado = "Notable";  
    } else if (nota < 10) {  
        resultado = "Excelente";  
    } else {  
        resultado = "Excelente nota";  
    }
```

Condicionales para determinar el valor de la variable resultado.

```
// Escribe el valor en una caja de la web mediante innerHTML.  
document.getElementById("parrafo").innerHTML = resultado;}
```

Las estructuras condicionales anidadas se caracterizan por incluir instrucciones dentro de funciones.

Siguiendo el caso anterior podemos utilizar una instrucción ***if*** dentro de otra instrucción ***if*** quedando de la siguiente forma. Utilizando este sistema podemos dar salida a tantos resultados como nos convenga.

```
if (<condición>)  
{  
    <Instrucción(es)>  
}  
  
else  
{  
    if (<condición>) { <Instrucción(es)> }  
    else { <Instrucción(es)> }  
}
```



BUCLE WHILE // DO WHILE

El tipo de estructura **while** es un bucle de código fuente que se ejecuta siempre mientras se esté cumpliendo la condición que se haya especificado. Si en la primera pasada la condición no se cumple automáticamente salta al siguiente código sin tener en cuenta el bucle **while**.

```
while ( x <= 100){  
  
    document.write(x);  
  
    document.write('<br>');  
  
    x=x+1;  
  
}
```

El bucle se repite 'mientras' la variable x sea inferior o igual a 100. El resultado se muestra en el navegador. En este tipo de estructura puede darse el caso que no entre ninguna vez.

//El bucle carga un texto durante 10 veces empezando desde cero.

```
while (i < 10) {  
  
    texto += "El número es " + i;  
  
    i++; }  
  
// Carga el valor de texto en la caja con id 'parrafo' de HTML  
  
document.getElementById("parrafo").innerHTML = texto;
```

El tipo de estructura **do ... while** es un bucle de código que funciona exactamente igual que **while** pero la diferencia radica en que en esta segunda forma **como mínimo pasa una vez por el bucle** hasta que llega a la condición para ser evaluado el criterio.

```
do {  
  
    valor = prompt('Ingrese valor:',''); // Pide valor por teclado  
  
    valor = Number(valor); // Convierte a entero el num introducido  
  
    suma = suma + valor; // Suma el valor al total de la suma.  
  
    x = x+1; // Aumenta la variable 'contadora' x en una unidad.  
  
} while (x <= 10)
```

BUCLE FOR

Este tipo de bucle realiza un número de pasadas determinadas por el código. En este caso sabemos el número de veces que se pasará por dicho código. El bucle **for** es utilizado frecuentemente para recorrer arrays y dispone de tres “parámetros”.

1. El valor inicial del bucle. Por regla general suele ser cero este valor. Se puede omitir este valor y adoptará el valor que almacene en memoria. Es recomendable introducirlo dentro del bucle **for** y iniciararlo con el valor deseado para evitar sustos inesperados.
2. El valor condicional del bucle. Es un criterio o condición que determinará .
3. El incremento o decremento del bucle. Es una variable del tipo contador que al llegar a un valor determinado hace no vuelva a entrar en el bucle.

La inclusión y ejecución de la sentencia **break**; en cualquier parte del bucle provoca la salida inmediatamente del bucle. Esta sentencia puede ser útil para “hacer saltar” del bucle durante la ejecución del código cuando se ha encontrado un registro en concreto y de esta forma no será necesario tener que recorrer el resto del elemento.

Ejemplo 1:

```
// El bucle concatena un texto durante 5 veces empezando desde cero.  
  
// Valor inicial:0, Mientras que 'i' sea < 5, incrementando de 1 en 1.  
  
for (i = 0; i < 5; i++) { texto += "El número es " + i + "<br>";}  
  
// Carga el valor de texto en la caja con id 'parrafo' de HTML  
  
document.getElementById("parrafo").innerHTML = texto;
```

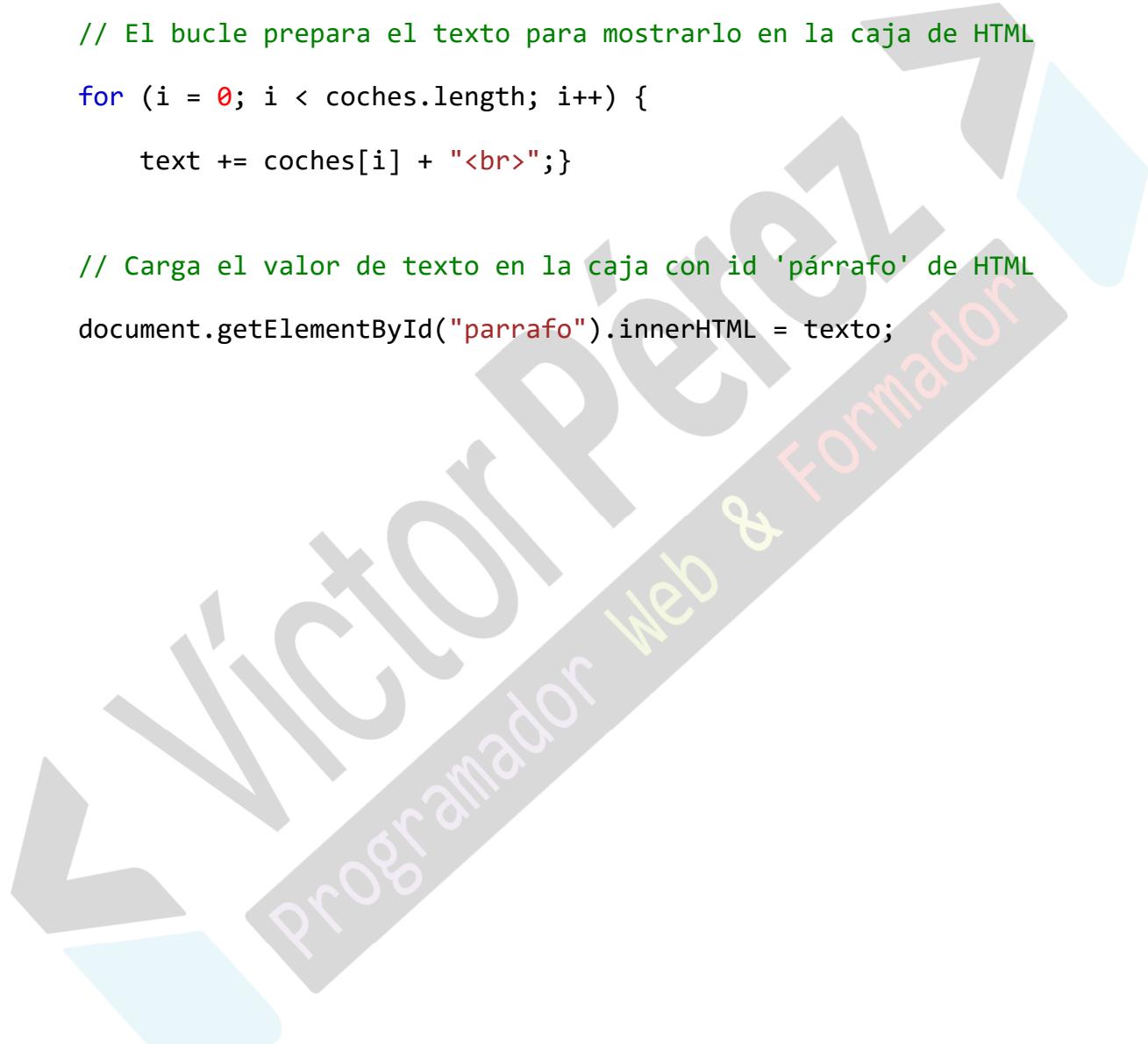
El número es 0
El número es 1
El número es 2
El número es 3
El número es 4



Resultado: Escribe en la caja el resultado que se ha ido cargando en cada pasada del bucle **for**.

Ejemplo 2:

```
// Declaración de la variable con sus valores. Esto es un array.  
  
var coches = ["BMW", "Volvo", "Saab", "Ford", "Audi"];  
  
// Con la propiedad 'length' sabemos que el tamaño del array es 5.  
  
// El bucle prepara el texto para mostrarlo en la caja de HTML  
  
for (i = 0; i < coches.length; i++) {  
  
    text += coches[i] + "<br>";}  
  
// Carga el valor de texto en la caja con id 'párrafo' de HTML  
  
document.getElementById("parrafo").innerHTML = texto;
```



ESTRUCTURAS SWITCH

Este tipo de estructuras se utilizan cuando se disponen de muchas posibles soluciones y como alternativa a la función **if**. Se utiliza cuando sabemos el valor exacto que podemos recibir, es decir, la comparación da como resultado verdadero o falso.

- **switch**: inicia este tipo de estructura. Se le debe especificar un valor mediante el cual realizará la comparación con cada uno de los casos.
- **case**: se utiliza para determinar el código a ejecutar si existe coincidencia para cada uno de los diferentes posibles resultados.
- **default**: determina el código que se ejecutará si ninguno de los casos posibles se ha podido ejecutar por falta de coincidencia.
- **break**: permite salir de la sentencia switch en el caso que alguno de los códigos haya sido ejecutado. Con esta palabra se libera al navegador de tener que comparar con todos los casos.

```
switch (valor) {  
    case 1: document.write("uno");  
    break; // Hace saltar del switch  
    case 2: document.write("dos");  
    break;  
    case 3: document.write("tres");  
    break;  
    case 4: // Se pueden poner 2 casos sobre el mismo código  
    case 5: document.write("cuatro o cinco");  
    break;  
    default:document.write("Introduce un numero entre 1 y 4.");  
}
```

En el siguiente ejemplo los diferentes casos son cadenas de texto.

```
switch (color) {  
  
    case "blanco": document.write("color blanco");  
  
    break;  
  
    case "negro": document.write("color negro");  
  
    break;  
  
    case "rojo": document.write("color rojo");  
  
    break;  
  
    default:document.write("Color no es blanco ni negro ni rojo");  
  
}
```

Instrucciones de transferencia de control

En algunas ocasiones **los bucles se deben interrumpir**. JavaScript ofrece utilizar dos palabras reservadas para detener la ejecución de bucles y condicionales. Aparentemente realizan la misma acción (interrumpir) pero lo ejecutan con alguna diferencia.

- **continue:** Esta instrucción interrumpe el ciclo actual y avanza hasta el siguiente ciclo dentro del mismo bucle sin ejecutar el resto del código del ciclo. En este caso el bucle principal se sigue ejecutando, pero concretamente en ese ciclo las líneas de código que hay a continuación de 'continue' se pasan por alto hasta el siguiente ciclo.
- **break:** Esta instrucción interrumpe el bucle. Todas las instrucciones restantes y los ciclos pendientes se ignoran saliendo del bucle para continuar con el resto del código.

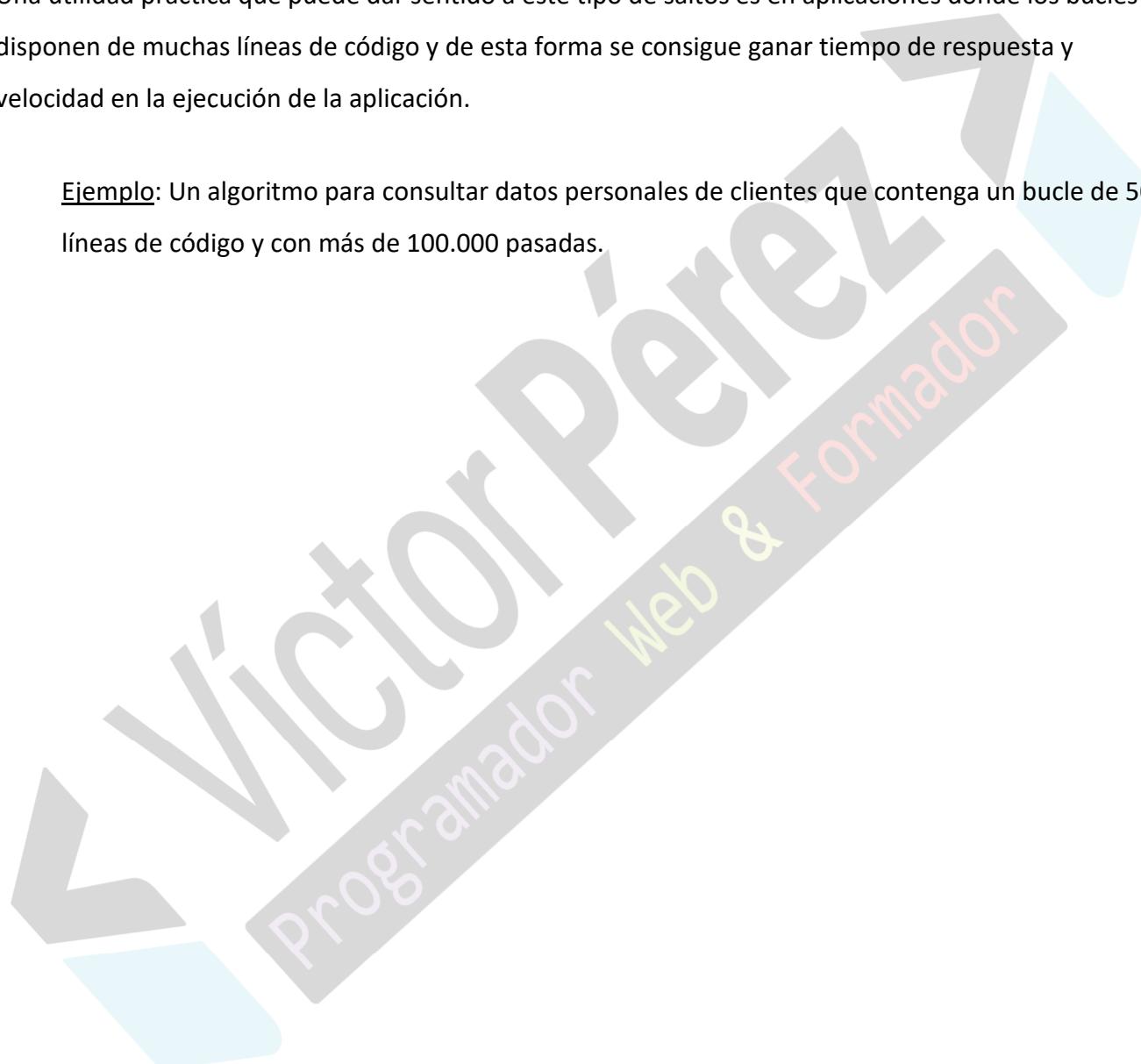
```
for (i = 0; i < coches.length; i++){  
  
    var marca = coches[i];  
  
    if (marca == "Seat") {  
  
        continue;  
    }  
  
    text += coches[i] + "<br>";  
  
}
```

El bucle FOR recorre un array. Si en alguna posición del array el valor es "Seat", salta de ciclo y pasa a la siguiente posición del array sin ejecutar el resto del código. Si fuese una instrucción **break** en vez de **continue**, directamente saltaría del bucle FOR.

Este tipo de saltos dentro de código en POO (programación orientada objetos) no son recomendables ni del todo bien vistos ya que rompen la lógica de la programación. Es decir, el código se tiene que ejecutar de forma secuencial de principio a fin sin utilizar atajos o puertas traseras para evitar ejecutar determinadas zonas de código.

Una utilidad práctica que puede dar sentido a este tipo de saltos es en aplicaciones donde los bucles disponen de muchas líneas de código y de esta forma se consigue ganar tiempo de respuesta y velocidad en la ejecución de la aplicación.

Ejemplo: Un algoritmo para consultar datos personales de clientes que contenga un bucle de 500 líneas de código y con más de 100.000 pasadas.



FUNCIONES EN JAVASCRIPT

Características de las funciones en JavaScript

- Una función es un **bloque de código encapsulado** que está diseñado para realizar una tarea específica y retornar o no un resultado después de ejecutarse.
 - Técnicamente **hablamos de función cuando no forman parte de un objeto** y en el caso de que formen parte de un objeto se llaman métodos. En ambos casos el código hace lo mismo.
 - Las funciones como en todos los lenguajes de programación **se ejecutan cuando se invocan**, es decir cuando se las llama mediante alguno de los sistemas posibles y no por defecto cuando se realiza la carga del código.
 - Las funciones permiten **segmentar el programa en varias secciones** permitiendo reutilizar el código en otras páginas.
 - Las funciones **evitan tener que copiar el mismo código de programación** en diferentes zonas del código. Mediante una única función se puede invocar invoca tantas veces como sea necesario.
 - Las funciones se pueden **agrupar en archivos externos** y enlazados a la página web, exactamente de la misma forma que se realiza con los archivos en CSS.
 - Utilizar funciones permite la **reutilización de código** de forma sencilla.
- Las funciones en JavaScript tienen la siguiente sintaxis:
1. Palabra reservada '**function**' para hacer referencia que es una función.
 2. **Nombre** de la función. El usuario determina como se llamará la función. El nombre a ser posible debería ser representativo sobre el tipo acción que realiza.
 3. **Parámetros** si los tuviera. No todas las funciones utilizan parámetros. Los parámetros se comportan como variables locales del código siempre que estén bien declaradas. Las variables son "los ingredientes" que necesita la función para poderse ejecutar de forma óptima.

```
function nombre_función (param1, param2, param3, etc...) {  
  
    // Código que se ejecutará.  
  
}
```

- Las funciones se pueden invocar (llamar) de 3 formas:

1. Cuando se produce un **evento**, por ejemplo, un clic con el mouse ("onclick").
2. Cuando se llama **desde el código** HTML o del mismo JavaScript que es lo más habitual.
3. De forma **automática** (se auto invoca).

Las funciones pueden **devolver un resultado** después de su ejecución, aunque NO es obligatorio. Esto lo determina el usuario en el momento de la programación de la misma mediante la palabra reservada **return** seguido del cálculo o variable a devolver.

Por convención se recomienda que las funciones devuelvan algún tipo de resultado, ya sea un cálculo o en su defecto un valor booleano que determine si se ha ejecutado correctamente la función.

El resultado de las funciones se puede aprovechar para introducirlo en un cálculo directamente y no es necesario almacenarlo previamente en una variable.

```
/* La siguiente función convierte grados Fahrenheit a Celsius. Se invoca
desde el código JavaScript y el resultado que devuelve es aprovechado para
mostrarlo en la pantalla del navegador mediante 'innerHTML' */

<script>

    document.getElementById("caja1").innerHTML =
        "La temperatura es" + aCelsius(77) + " Celsius";

    function aCelsius(fahrenheit) {
        // Convierte grados Fahrenheit a Celsius
        return (5/9) * (fahrenheit-32);
    }

</script>
```

Si se invoca una función sin los paréntesis **devuelve el código en formato texto** que contiene la función. En ocasiones puede ser interesante conocer el código que se ha escrito y verlo por el navegador.

Las funciones deben tener un **nombre representativo** sobre el cálculo o acción que realizan, es decir, si deseamos crear una función para calcular el volumen de un cilindro, la función podría llamarse por ejemplo **volumenCilindro()**.

El nombre de las funciones es **sensible a las mayúsculas y minúsculas (case sensitive)**. JavaScript es un lenguaje de programación muy estricto con la sintaxis. Por tanto, a la hora de llamar a las funciones debe contener exactamente el mismo nombre de parámetros que en su declaración.

Las **funciones son objetos** en Javascript de tal forma que podríamos declarar propiedades y métodos dentro de ellas, hacer instancias y acceder a su contenido. A continuación se muestran 2 ejemplos, el primero una función declarada como objeto y el segundo con propiedades y métodos.

El primer ejemplo muestra como una función devuelve un resultado a través de la palabra reservada 'return' para ser almacenada posteriormente en una variable.

```
// Ejemplo 1: Almacenamos el resultado de una función en una variable

const Celsius = function (grados) {
    return (5/9) * (grados-32);

}

console.log(Celsius(85));
```

Este segundo ejemplo solo se debe visualizar y intentar comprender si previamente se han trabajado con objetos y sus características.

// Ejemplo 2: Objeto con propiedades y métodos.

```
function EjemploObjeto(ape) {
```

```
    this.nombre = "Homer";
```

```
    this.apellido = ape;
```

```
    this.muestraInfo = function (){
```

```
        console.log(this.nombre + " " + this.apellido);
```

```
}
```

```
    console.log("Objeto inicializado");
```

```
}
```

// Declaración del objeto.

```
let PrimerObjeto = new EjemploObjeto(Simpson);
```

```
let SegundoObjeto = new EjemploObjeto(Simpson);
```

// Asignación del valor a la propiedad de cada objeto creado.

```
PrimerObjeto.titulo = "Lisa";
```

```
SegundoObjeto.titulo = "Bart";
```

// Muestra valor de la propiedad por la consola.

```
console.log(PrimerObjeto.titulo);
```

```
console.log(SegundoObjeto.titulo);
```

// Invoca método del objeto que muestra las propiedades.

```
PrimerObjeto.muestraInfo();
```

FUNCIONES DESDE ARCHIVO EXTERNO (*.JS)

Las funciones se pueden agrupar en archivos externos de Javascript (*.js) y ser invocadas desde cualquier documento HTML siempre que disponga del enlace al archivo correctamente introducido.

El enlace se realiza dentro de las etiquetas **<head>** de las páginas y dispone de etiqueta de apertura y de cierre. Si no se cierra la etiqueta **<script>** no enlazará correctamente con el archivo.

```
<script type="text/javascript" src="archivo_funciones.js"> </script>
```

Las funciones para poder ser invocadas desde código HTML es necesario que:

1. Desde el **<head>** se invoca al archivo con extensión .js que contiene las funciones. Este archivo se enlaza a través del atributo **src**.
2. Desde el **<body>** se invoca a la función que se encuentra dentro del archivo enlazado.

El siguiente ejemplo muestra un código HTML que contiene un enlace hacia el archivo 'funciones.js' externo. Las funciones se invocan desde el mismo HTML con la etiqueta **<script>** aunque se podrían invocar desde el mismo código HTML mediante eventos u otro sistema.

```
<html>
  <head>
    <title> Problema </title>
    <script type="text/javascript" src="funciones.js"> </script>
  </head>
  <body>
    <script type="text/javascript">
      document.write('La fecha de hoy es:' +retornarFecha());
      document.write('La hora es:' +retornarHora());
    </script>
  </body>
</html>
```

Estas funciones que se encuentran a continuación están ubicadas en el archivo externo 'funciones.js' que se llama desde el interior del script. Es posible también llamar a las funciones desde el mismo código HTML dentro de las etiquetas mediante los eventos.

```
function retornarFecha() {  
    var fecha = new Date();  
    // Carga en la variable 'cadena' la fecha actual  
    var cadena = fecha.getDate()+' / '+ (fecha.getMonth()+1) +' / '  
                +fecha.getFullYear();  
    // Devuelve el valor de cadena  
    return cadena;  
}  
  
function retornarHora() {  
    var fecha = new Date();  
    // Carga en la variable 'cadena' la fecha actual  
    var cadena = fecha.getHours()+':'+fecha.getMinutes()+' : '  
                +fecha.getSeconds();  
    // Devuelve el valor de cadena  
    return cadena;  
}
```

FUNCIONES ANÓNIMAS // CALLBACKS

Funciones anónimas

Las funciones anónimas se caracterizan por NO tener nombre de función. Este tipo de funciones se utilizan cuando **el valor se carga directamente sobre una variable**. La finalidad es no tener que crear una función específica y tener que invocarla para su ejecución. Los parámetros de la función son opcionales y se cargan desde la variable. Este tipo de funciones son extremadamente útiles para definir complejos patrones de programación.

// La variable x se carga con la función anónima que se le asigna, pero solo se carga cuando es invocada desde el método 'innerHTML'.

```
<script>

    var calcular = function (a, b) {
        return a * b // Devuelve la multiplicación
    };

    document.getElementById("caja1").innerHTML = calcular(4, 3);

</script>
```

La propiedad **arguments.length** permite averiguar cuantos parámetros se le ha pasado a una función.

```
function miFuncion(a, b) {
    return arguments.length; // Devuelve 2 porque son los argumentos
}
```

El método **toString()** aplicado a una función devuelve la función como una cadena de texto.

```
function miFuncion(a, b) {
    return a * b;           // function miFuncion(a, b) {return a * b;}
}

var txt = miFuncion.toString();
```

Callbacks

Traducido del inglés un callback es "una llamada de vuelta".

Las funciones pueden recibir cualquier tipo de dato como parámetro (numero, texto, array, etc..). Los callbacks **son funciones que reciben como parámetros otras funciones**, por tanto, primero se ejecutan las funciones externas y luego las funciones internas (las pasadas como parámetro).

Los callbacks y las funciones anónimas están muy directamente relacionadas y permite realizar acciones de forma asíncrona puesto que se ejecutan en diferentes momentos.

Un uso habitual de los callbacks es ejecutar una función dentro de otra cuando la primera ha realizado una tarea y dispone de un valor que será utilizado por la función que se ha pasado como parámetro.

```
// Paso 1. Llama a la función calcular() y le pasa 2 números (parámetro  
1 y 2) y 2 funciones (parámetros 3 y 4)  
  
calcular(2, 3, function (resultado) { console.log('Suma', resultado)},  
function (resultado) { console.log('Resta', resultado)  
})  
  
// Paso 2. Declaración de la función que recibe por parámetro otras 2  
funciones. En este caso las dos funciones pasadas son 'sumarCB' y  
'restarCB'.  
  
function calcular( datoA, datoB, sumarCB, restarCB ) {  
    var suma = datoA + datoB  
    var restar = datoA - datoB;  
  
    sumarCB( suma );  
    restarCB( restar );  
}
```

FUNCIONES ARROW, FAT ARROW, LAMBDA

Las funciones **arrow**, **fat arrow** o **lambda** son otra **forma de definir una función anónima** en Javascript de forma más reducida sin tener que escribir demasiadas líneas de código. A continuación, se muestran diferentes formas o variantes para declarar una función del tipo arrow.

// Definición valores 1^a línea:

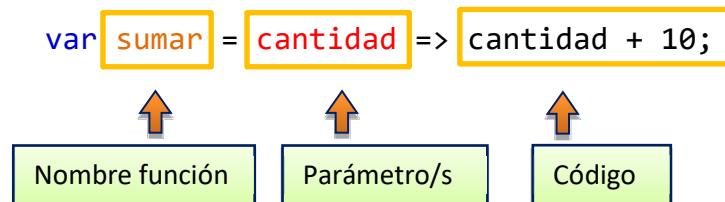
- La variable '**saludar**' es el nombre de la variable que almacenará el valor de la arrow function tras ser ejecutada, es decir, invocamos a 'saludar' y ejecuta la función
- La variable '**nombre**' es el parámetro de la función arrow. Como solo tiene un parámetro se puede poner sin paréntesis. Si hubiese más parámetros irían entre paréntesis.
- El símbolo **=>** determina que es una función tipo arrow. Sustituye a 'function' si fuese una función anónima.
- Si la función tiene solo una línea de código es posible dejarla sin las llaves de apertura y cierre de función. Además, si la función devolviera algún valor mediante '**return**' este también se podría omitir.
- Avanzado: Las funciones anónimas generan un contexto que afecta a la variable '**this**' dentro de una función que se encuentra en un objeto cuando se invocan, por tanto, devuelven '**undefined**'. Las arrow functions no generan este contexto por lo que devuelven el objeto al que pertenecen cuando son invocadas. Las arrow no tienen un this propio.

// La 2^a línea llama a la función 'saludar' y le pasa el parámetro 'nombre'. El valor devuelto es la cadena de texto.

```
var saludar = nombre => `Hola ${nombre}`;
```

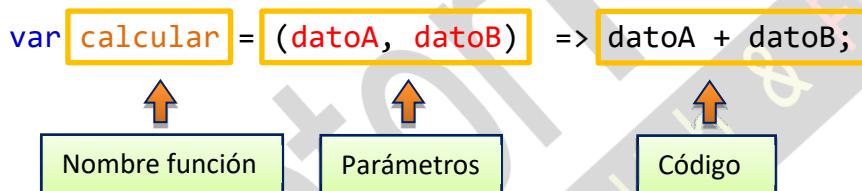
```
console.log(saludar("Susana"));
```

// Esta forma llama a la función pasándole un valor (50) y el resultado devuelto es la suma del valor pasado más 10.



```
console.log(sumar(50));
```

// En este caso se pasa más de un parámetro y por tanto la función recibe más de un argumento para realizar el cálculo de la suma. En este caso como recibe más de un parámetro es obligatorio introducirlos entre paréntesis.



```
console.log(calcular(10, 15));
```

// Se invoca a la función enviándole dos parámetros y como hay más de una línea de código es necesario encapsularlo entre llaves. Además, internamente hay una variable declarada que se necesita para el cálculo.

```
var generar = (datoA, datoB) => {
    var datoC = 5;
    return datoA + datoB + datoC;
}
console.log(generar(10, 15));
```

// Esta función no recibe parámetros y se especifica con los paréntesis (). A continuación, el resto de las operaciones es como cualquier otra función. La secuencia de ejecución es llamar a 'validar' y esta ejecuta el código que haya a la derecha del símbolo igual.

```
var validar = () => {  
    return 'Validación correcta';  
}  
  
console.log(validar());
```



FUNCIONES CON PARÁMETROS EN JS (rest y spread)

Las funciones con parámetros reciben una copia del valor de las variables declaradas entre los paréntesis (atributos, argumentos o parámetros), es decir, los parámetros que necesita para ejecutarse.

```
function <nombre de función>(argumento1, argumento2, ..., argumento n){  
    < código de la función>  
}  
  
// Ejemplo de función que debe recibir 3 parámetros.  
  
function suma3edades ( argumento1, argumento2, argumento3 ) {  
    < código de la función>  
}  
  
suma3edades ( edad1, edad2, edad3 ); // Llamada a la función
```

En este caso, la función que llama debe enviar el mismo número de atributos que la función que los espera, es decir, si llamamos a la función **suma3edades()** y le enviamos el valor de las tres edades, la función **suma3edades()** declarada en la parte superior deberá recibir esos tres valores para poderse ejecutar correctamente.

Hay que tener en cuenta también el tipo de valor que se le pasa. Si la función espera un conjunto de números para hacer una suma, no se le puede pasar textos, arrays o cualquier otro tipo de dato porque daría error de ejecución porque el código no está preparado. En este caso la función devolverá un error o un valor incorrecto como resultado.

El nombre de los parámetros puede variar de la llamada respecto a la ejecución de la función. Solo es necesario que sean el mismo número de parámetros y del mismo tipo para que se ejecuten correctamente.

```
// Llamada a la función pasándole dos parámetros (lado1 y lado2)
```

Por ejemplo, llamamos a la función **areaRectangulo(lado1 , lado2)**;

```
// Ejecución de la función y los valores adquieren otro nombre.
```

Se ejecuta la función **areaRectangulo(arista1 , arista2)**;

Un concepto teórico que se debe tener presente cuando se llama a una función es que se le pasan atributos, y cuando la función los recibe se convierten en parámetros de la función.

En algunos casos se pueden llamar a las funciones sin necesidad de enviarles los parámetros, aunque estén declaradas. En estos casos la función llamada debe tener en su declaración la **inicialización de las variables** por si no fuesen enviadas en su llamada.

```
// Ejemplo de función que en su declaración inicializa los valores de las variables por si estas no fuesen enviadas (pasadas) en la llamada.
```

```
function suma3edades( argumento1=10, argumento2=20, argumento3=30) {  
    < código de la función>  
}
```

```
// Llamada a la función sin parámetros que provocará que los tres argumentos tengan asignado el valor especificado en su declaración.
```

```
suma3edades ();
```

Las funciones con parámetros (y sin parámetros también) pueden devolver un valor que puede ser asignado a una variable del hilo principal del programa. De esta forma podemos obtener resultados que utilizaremos en posteriores cálculos.

Para devolver un valor mediante este sistema se deberá utilizar la palabra reservada '**return**' con el valor que deseamos devolver, por ejemplo, **return valor1**. El retorno del valor se realiza en la última línea de la función como norma general ya que cualquier código que haya a continuación de esta palabra reservada no se ejecutará. La palabra reservada '**return**' hace saltar (salir) de la función.

```
// Ejemplo de función devuelve el resultado calculado.
```

```
function suma3edades(argumento1=10, argumento2=20, argumento3=30) {  
    var suma = argumento1 + argumento2 + argumento3;  
    return suma;  
}
```

Otra particularidad de las funciones con parámetros radica en que estas pueden recibir un **número indeterminado de argumentos**. Para este caso no es posible declarar un número fijo de variables en la función y podemos quedarnos cortos o pasarnos con los parámetros.

Por tanto, la solución al problema es declarar los parámetros de las funciones con el operador tipo '**rest**' siguiendo una sintaxis específica. Este operador está disponible desde ES6 ("**rest operator**"). De esta forma podemos enviar tantos parámetros como queramos en la llamada de la función porque todos **serán recibidos y almacenados en una variable del tipo array**.

Posteriormente, el array deberá ser recorrido para poder extraer los diferentes valores que hayan sido almacenados en cada una de las posiciones. Si la función recibe 5 parámetros el array tendrá el mismo número de posiciones ocupadas como valores haya recibido la función.

Para declarar una función con 'N' argumentos se escribirá el nombre de la variable (que será un array) **precedido de tres puntos suspensivos**. Para acceder a los diferentes valores almacenados se realizará utilizando los métodos y técnicas para trabajar con arrays.

```
// Ejemplo de función que recibe un número indeterminado de parámetros.

function sumaEdades (...argumentos) {

    var suma = 0;

    for (let i = 0; i < argumentos.length ; i++) {
        suma += argumentos[i];
    }

    return suma;
}
```

El paso inverso al operador '**rest**' es la transferencia de valores desde la llamada a la función con '**spread**' (esparcir). Este sistema envía 'N' valores en la llamada de la función **en forma de array** y posteriormente la función que recibe los valores los distribuye (esparce) por las diferentes variables declaradas en la función.

```
// Invoca a la función pasándole un número indeterminado de valores en
// forma de array.

sumaEdades (...edades);
```

Si el número de valores recibidos por la función en el array con el sistema '*'spread'*' es menor a los valores declarados en la función provocará que los "no rellenados" tengan el valor '*'undefined'*'. Y si el número de valores recibidos por la función desde el array fuesen mayor que los declarados en la función algunos valores del array no serían asignados y por tanto perdidos.

```
// La función espera tres valores. Si se utiliza el sistema 'spread'  
deberán coincidir el número de valores pasados con los recibidos.
```

```
function sumaEdades (edad1, edad2, edad3) { ... }
```

Como es difícil saber el número de atributos que se envían (con '*'spread'*') y saber también el número de parámetros que se reciben en la función (con '*'rest'*') se intentan **combinar estos dos sistemas** para evitar tener problemas en tiempo de ejecución. Se envían '*n*' parámetros y se reciben '*n*' parámetros.

```
// Invoca a la función con el operador 'spread' y recibe los datos  
mediante el operador 'rest'.
```

```
sumaEdades (...edades);
```

```
function sumaEdades (...edades) { ... }
```

TRY / CATCH / FINALLY

La estructura **try{ } ; catch(infoExcepcion){ }** y **finally{ }** nos permite programar las acciones a realizar si se produce algún error durante la ejecución de un código. Está formado por 3 partes, el try{} el catch(){} y el finally{}:

- **try{} :** Contiene el bloque de código del que querremos controlar los posibles errores que se produzcan al ejecutar el programa. Si el código no se ejecuta correctamente pasa a 'catch' y sino va directamente a 'finally'.
- **catch(variable){} :** Contendrá las acciones a realizar cuando se produzca un error. Recibe un parámetro que hace referencia al error que se ha producido.
- **finally{} :** Parte opcional que contendrá las acciones que siempre querremos realizar aunque se haya producido o no un error.

El siguiente ejemplo mostrará dos alertas con los mensajes:

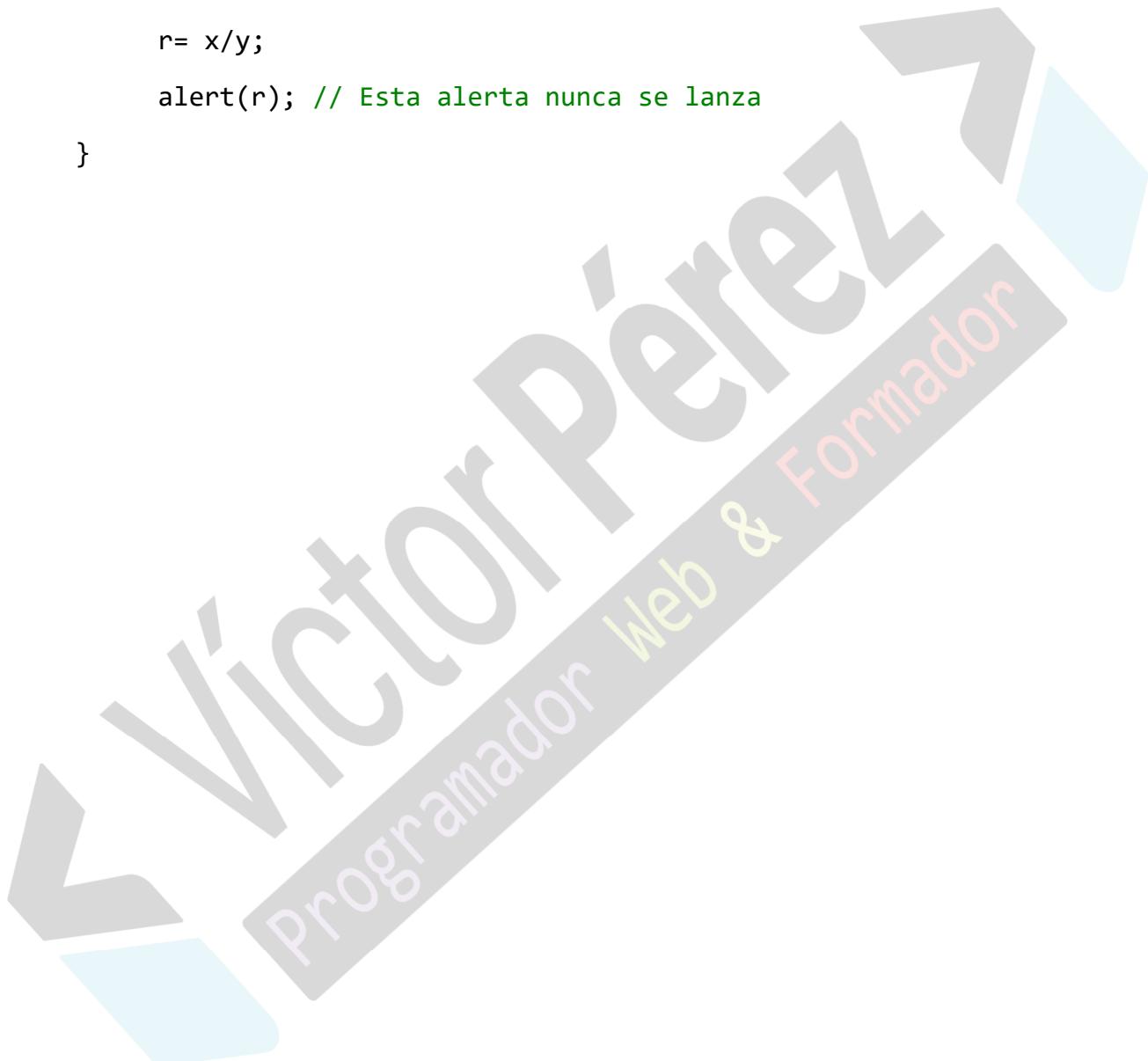
'Error: ReferenceError: y is not defined'

'Cuidado con no declarar las variables'

```
let x=20, r;  
  
try {  
    r= x/y;  
    alert(r); // Esta alerta nunca se lanza. Variable 'y' no existe.  
}  
  
catch (err) {  
    alert("Error: " + err);  
}  
  
finally { // Se ejecuta siempre despues de try..catch  
    alert("Cuidado con no declarar las variables");  
}
```

La sentencia '**throw**' permite lanzar un mensaje desde dentro del 'try' por si alguna sentencia no se cumple.

```
let x=20, r;  
  
try {  
  
    if(x==20){ throw "El numero introducido es 20"; }  
  
    r= x/y;  
  
    alert(r); // Esta alerta nunca se lanza  
}
```



STRINGS (CADENAS DE TEXTO)

Las cadenas de texto almacenan un conjunto de caracteres. Una cadena de texto es cualquier texto que se encuentra entre comillas simples o dobles.

```
var coche = "Volvo XC60"; // Comillas dobles
```

```
var coche = 'Volvo XC60'; // Comillas simples
```

Todos los objetos que almacenan texto se consideran elementos del objeto 'String' por lo que pueden utilizar todas sus propiedades y métodos disponibles. Una propiedad muy utilizada es '*length*' que devuelve el tamaño de una cadena de texto.

```
var txt = "ABCDEFGHIJKLMNPQRSTUVWXYZ"; // Texto guardado en variable
```

```
var tamaño = txt.length; // 'Tamaño' almacena el tamaño de 'txt'
```

Si las comillas simples o dobles se quieren utilizar como caracteres y no como elementos de apertura o cierre es preciso utilizar el carácter contra barra (\) antes de las comillas.

```
var y = "El coche de marca \"Audi\" es blanco."
```

Las cadenas de texto se pueden comportar como objetos y se pueden declarar así. No se recomienda declarar Strings como objeto porque ralentiza la velocidad de ejecución y puede provocar resultados inesperados. ES5 acepta esta última definición en el estándar.

```
var x = "Pep";
```

```
var y = new String("Pep"); // Ralentiza ejecución del programa.
```

Además, las cadenas de caracteres o strings se almacenan como array de tal forma que se podría acceder a los caracteres del string como si fuese un Array.

```
var x = "Pep";
```

```
var y = x[1]; // Devuelve la letra 'e' como resultado.
```

Métodos en Strings (Cadenas De Texto)

Las cadenas de texto al considerarse objetos de la clase String pueden acceder a las propiedades y métodos que esta dispone. La clase String solo dispone de la propiedad '**length**' que determina el tamaño de la cadena de texto.

- **length**: Es una propiedad que devuelve el tamaño de la cadena de texto. Los strings comienzan por la **posición cero** como si fuesen un array.

```
var str = "Audi Q5";  
  
var pos = str.length; // Devuelve el número 7.
```

El número de métodos se especifican a continuación:

- **String()**: Es el método más representativo de la clase y convierte una variable a string.

```
var edad = 35;  
  
var edadtxt = String(edad); // Convierte edad a texto.
```

- **indexOf()**: Devuelve la posición de la primera ocurrencia de un texto especificado en una cadena de texto. La posición del primer carácter es cero. Si no encuentra el texto devuelve -1. El método acepta un segundo parámetro que indica la posición de partida.

```
var str = "El coche Audi es blanco y la marca Audi tiene aros";  
  
var pos = str.indexOf("Audi") // 'pos' vale 9.
```

- **lastIndexOf()**: Devuelve la posición del elemento buscado empezando por la derecha. La posición del primer carácter de la cadena de texto es cero. Si no encuentra el texto devuelve -1. El método acepta un segundo parámetro que indica la posición de partida.

```
var str = "El coche Audi es blanco y la marca Audi tiene aros";  
  
var pos = str.lastIndexOf("Audi") // 'pos' vale 35.
```

- **search()**: Busca un valor especificado en una cadena y devuelve la posición inicial. Trabaja igual que el método 'indexOf()' pero a diferencia de este el método search() acepta muchos más parámetros de configuración.

```
var str = "El coche Audi es blanco";
var pos = str.search("Audi") // 'pos' vale 9.
```

- **slice()**: Extrae una parte de una cadena de texto y devuelve la parte extraída en una nueva cadena. El método tiene dos parámetros (posición de inicio y posición final). Si el valor es negativo se cuenta desde el final de la cadena, pero a la hora de extraer lo realiza igualmente de izquierda a derecha. Empieza a contabilizar desde la posición cero como si fuese un array.

```
var str = "Audi, Mercedes, BMW";
var res = str.slice(6, 13); //Extrae 'Mercedes' y guarda en 'res'
var res = str.slice(6); // Desde posición 6 hasta el final
var res = str.slice(-13); // Resultado 'Mercedes, BMW'
var res = str.slice(-13, -6); // Resultado 'Mercedes'
```

- **substring()**: Funciona exactamente igual que el método 'slice()' pero la diferencia es que este método no acepta los valores negativos como parámetros. En este caso el segundo parámetro determina cuantos caracteres se deben extraer desde la posición facilitada en parámetro uno.

```
var str = "Audi, Mercedes, BMW";
var res = str.substring(6, 8); // Extrae 'Mercedes'
```

- **substr()**: Realiza una acción casi idéntica a slice(), extraer un texto, pero la diferencia radica en los parámetros que se le pasan. El primer parámetro determina la posición de inicio (incluida), y la segunda el número de caracteres hacia la derecha (y no la posición) que debe extraer.

```
var str = "Audi, Mercedes, BMW";
var res = str.substr(6, 8); //Extrae 'Mercedes' y guarda en 'res'
```

- **replace()**: Realiza la sustitución del primer texto por el segundo. El primer parámetro es el texto que se desea buscar, mientras que el segundo parámetro es el texto por el que se sustituirá. La función devuelve una cadena de texto nueva. El patrón /g sustituye todas las coincidencias en el texto, sino solo realizará la sustitución la primera instancia de la palabra buscada.

El segundo parámetro de este método también acepta una función que nos permitirá realizar operaciones más complejas.

```
var str = "Audi, Mercedes, BMW";
```

```
var res = str.replace(/Mercedes/g, "Ferrari");
```

- **replaceAll()**: Reemplaza todas las repeticiones del texto buscado por el sustituido en una cadena de texto.

```
var str = "Audi, Mercedes, BMW, Audi, Mercedes, BMW ";
```

```
var res = str.replaceAll(/Mercedes/g, "Ferrari");
```

Resultado: "Audi, Ferrari, BMW, Audi, Ferrari, BMW"

- **toUpperCase()**: Convierte toda una cadena de texto a mayúsculas. Para trabajar con textos que no conocemos el formato exactamente es recomendable convertirlo todo a mayúsculas o minúsculas.

```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.toUpperCase(); //Convierte el texto en mayúsculas
```

- **toLowerCase()**: Convierte toda una cadena de texto a minúsculas. Esta función y la anterior pueden ser muy útiles cuando el usuario puede introducir valores en mayúsculas o minúsculas indistintamente.

```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.toLowerCase(); // Convierte el texto en minúsculas
```

- **concat()**: Enlaza dos o más cadenas de texto. Puede ser utilizado para sustituir el carácter más (+) que también está aceptado para concatenar textos. El resultado es una nueva cadena que se debe almacenar en una variable.

```
var text1 = "Hello";
var text2 = "World";
var text3 = text1.concat(" ", text2, "JEJEJE"); //Enlaza 3 textos
```

- **charAt()**: Devuelve el carácter de un índice (posición) específico. Las posiciones en un string comienzan por cero.

```
var txt = "Audi, Mercedes, BMW";
var res = txt.charAt(6); //Devuelve la letra "M" según la posición
```

- **charCodeAt()**: Devuelve el valor Unicode del carácter situado en un índice específico. El valor Unicode se puede localizar en las tablas ASCII

```
var txt = "Audi, Mercedes, BMW";
var res = txt.charCodeAt(1); // Devuelve el num Unicode 65
```

- **fromCharCodeAt()**: Devuelve el carácter de un valor Unicode pasado como parámetro

```
var res = txt.fromCharCodeAt(65); // Devuelve la letra A
```

Es posible acceder a los elementos de un string como si fuese una matriz aunque algunos navegadores no aceptan esta codificación (cada vez menos). El string se trata como si fuese un objeto con lo que ralentiza el programa. Se debe evitar trabajar con cadenas de texto tratadas como matrices ya que los procedimientos no son eficientes que deberían ser durante su ejecución.

- **split()**: Este método permite **convertir una cadena de texto en una matriz**. Hay que especificar qué carácter es el elemento separador para cada posición de la matriz (Array). El valor devuelto se almacena en una variable como array.

```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.split(","); // Convierte un string a Array
```

- **repeat()**: Este método **repite la cadena tantas veces como se le especifique** en el parámetro. El valor puede ser almacenado en otra variable. Este método se utiliza para simular texto en la web y poder maquetar correctamente la página con contenido cuando estas se ajustan dinámicamente.

```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.repeat(2); // Repite dos veces el texto
```

Resultado: "Audi, Mercedes, BMW, Audi, Mercedes, BMW"

- **localeCompare()**: El método comprueba **si dos valores son iguales o no**. Con valor cero las dos cadenas son idénticas, con valor -1 o 1 determina que una es mayor que la otra. Este método hace diferenciación entre mayúsculas y minúsculas.

```
var str1 = "ab";
```

```
var str2 = "cd";
```

```
var n = str1.localeCompare(str2); // Diferencia Mayusc y Minusc
```

- **startsWith()**: (ES2015) Comprueba **si una cadena comienza por el texto especificado**. El resultado devuelve True o False. El valor se puede guardar en una variable. El segundo parámetro es la posición de inicio de la búsqueda.

```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.startsWith("Audi", 0); // Devuelve True
```

- **endsWith()**: (ES2015) Comprueba **si una cadena termina por el texto especificado**. El resultado devuelve True o False. El valor se puede guardar en una variable.

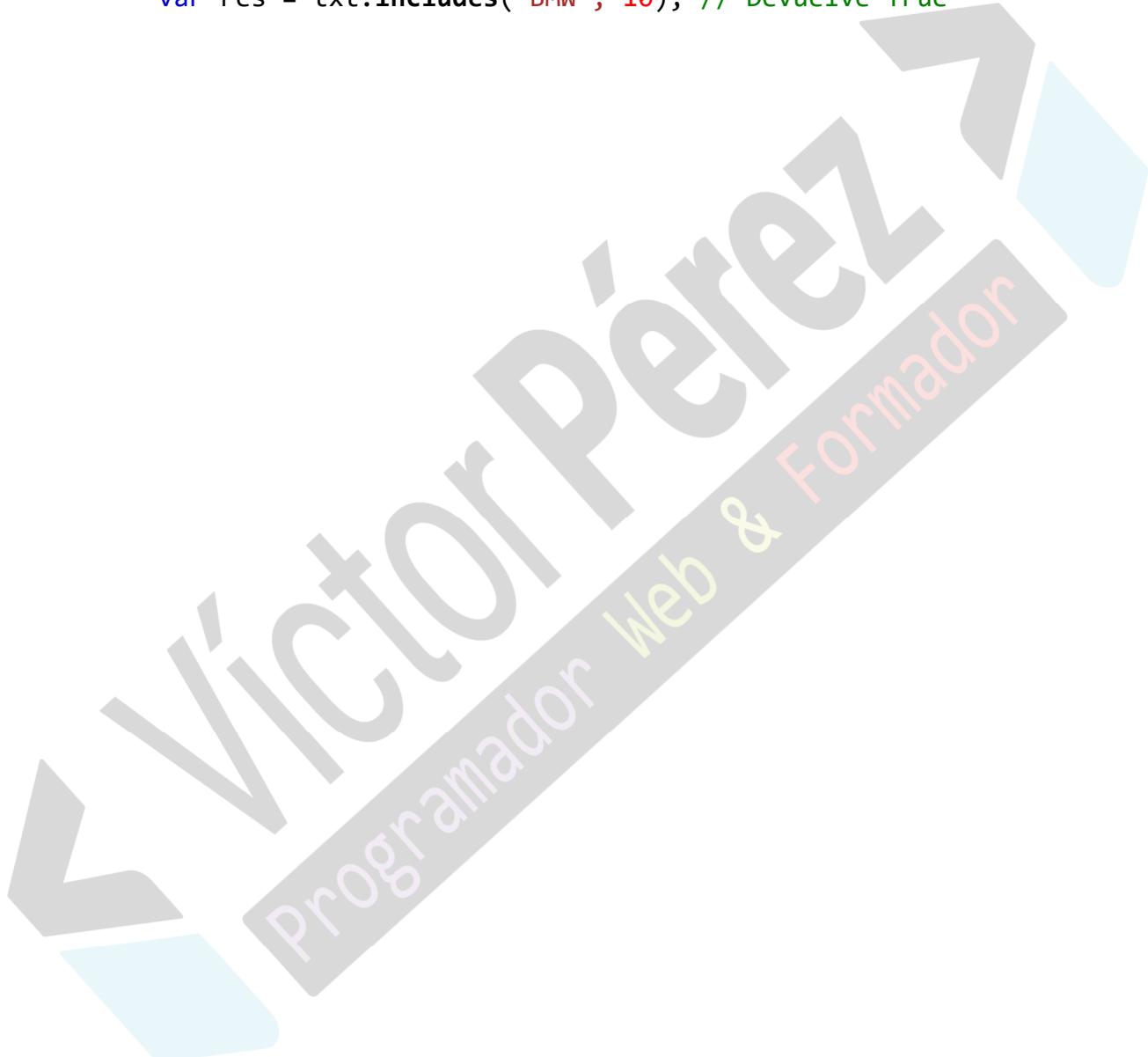
```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.endsWith("Audi"); // Devuelve False
```

- **includes():** (ES2015) Comprueba si una cadena contiene el texto especificado. El resultado devuelve True o False. El valor se puede almacenar en una variable. El segundo parámetro es la posición de inicio de la búsqueda.

```
var txt = "Audi, Mercedes, BMW";
```

```
var res = txt.includes("BMW", 10); // Devuelve True
```



- **match()**: Comprueba si una cadena de texto se encuentra en un texto y devuelve como resultado el texto buscado. Con el **patrón /g** devuelve todas las coincidencias que haya encontrado y las almacena dentro de un array. Y con el **patrón /i** (in case sensitive) no hace distinción entre mayúscula y minúsculas.

```
var str = "Manzana, banana, anciana";
```

```
var res = str.match(/ana/gi);
```

Resultado como array: "ana, ana, ana"

- **toString()**: Devuelve el valor del String. El valor se puede almacenar en una variable.

```
var str = "Hola mundo!";
```

```
var res = str.toString();
```

- **trim()**: Devuelve la cadena especificada **sin espacios a ambos lados** (derecha e izquierda).

```
var str = "    Hola mundo!    ";
```

```
var res = str.trim();
```

Resultado 'res': "Hola mundo!"

- **trimStart()**: (ES2017) Devuelve el texto **sin espacios a la izquierda**.

```
var str = "    Hola mundo!    ";
```

```
var res = str.trim(); // Devuelve "Hola mundo!    "
```

- **trimEnd()**: (ES2017) Devuelve el texto **sin espacios a la derecha**.

```
var str = "    Hola mundo!    ";
```

```
var res = str.trim(); // Devuelve "    Hola mundo!"
```

- **toLocaleLowerCase()** y **toLocaleUpperCase()**: Estas dos funciones son un poco especiales respecto al resto de funciones de texto en JavaScript. Convierten el texto en minúsculas y mayúsculas respectivamente, pero respetando la ubicación y configuración regional del host. La configuración regional **se basa en la configuración del idioma del navegador**.

Normalmente devolverá el mismo resultado que la función **toLowerCase()** salvo en zonas donde haya conflictos con varios idiomas (Bélgica, Austria, Canadá, Nueva Zelanda) que entonces la función adaptará el resultado a la zona geográfica regional.



Plantillas (backticks) y literales

Las **plantillas** son una forma de trabajar concatenando textos, pero escribiendo la sintaxis de una forma que en el futuro pueda hacer más fácil y accesible el mantenimiento del código.

Para escribir el texto con esta sintaxis el texto resultante deberá estar dentro de los acentos abiertos (*en inglés 'backticks'*) y escribir el nombre de las variables que almacenan los valores de texto con el símbolo dollar \$ y comprendido entre las dos llaves. De esta forma es más sencillo realizar cualquier modificación de código porque dispondremos de una plantilla más versátil.

```
var lenguaje = 'JavaScript';  
  
var lenguaje2 = 'HTML';  
  
var mensaje = `Me gusta ${lenguaje} y su integración con ${lenguaje2}`;  
  
console.log(mensaje);
```

Los **literales** son un concepto muy parecido que nos permite mostrar en la página el texto escrito de la misma forma, respetando los saltos de línea y la misma estructura como haya sido creada. También se realiza dentro de los acentos abiertos (*en inglés 'backticks'*) y siguiendo la misma lógica que en el caso anterior.

```
var mensajeMultilinea = `Hola mundo, estoy aprendiendo  
${lenguaje} y me gusta como se integra con ${lenguaje2}`;  
  
console.log(mensajeMultilinea);
```

NOTA: La concatenación de textos también se puede realizar utilizando el operador más (+) pero este sistema es más eficiente a la hora de realizar el mantenimiento del código y los algoritmos.

La clase String dispone de una serie de métodos para modificar determinadas propiedades de HTML.

Estos **NO son métodos estandarizados** y pueden no funcionar en todos los navegadores. Es recomendable en la medida de lo posible no utilizarlos.

Todos estos métodos pertenecen a ECMAScript 1. A continuación, se muestran estos métodos:

```
txt.anchor("ancla"); // Crea un ancla del tipo <a name="."> ... </a>  
  
var result = str.big(); // Pone letra grande en HTML es <big>  
  
var result = str.blink(); // Texto que parpadea, descatalogado.  
  
var result = str.bold(); // Texto en negrita equivalente a <b>  
  
var result = str.fixed(); // Texto como teletipo <tt>  
  
var result = str.fontcolor("green"); // Color de letra  
  
var result = str fontsize(7); // Tamaño de letra, solo los 7 tipos HTML  
  
var result = str.italics(); // Texto en cursiva equivalente <i>  
  
var result = str.link("https://www.google.com"); // Crea enlace <a>...</a>  
  
var result = str.small(); // Texto en letra pequeña <small>  
  
var result = str.strike(); // Muestra el texto tachado <strike>  
  
var result = str.sub(); // Texto como subíndice en HTML <sub>  
  
var result = str.sup(); // Texto como superíndice en HTML <sup>
```

NUMBERS

JavaScript tiene solo un tipo de número. Los números pueden contener o no decimales. Los decimales **se introducen con punto y no con coma**. Los números en JavaScript difieren de otros lenguajes en no disponer de un tipo de variable para cada tipo de número (entero, decimal, entero largo, etc...).

En Javascript todos los números están en formato binario de doble precisión de 64 bits según la IEEE 754 y comprenden los números desde -(253)-1 y 253-1

Los números en JavaScript se almacenan en variables de 64 bits (coma flotante de doble precisión) donde los bits 0 a 51 almacenan parte entera, los bits del 52 al 62 almacena parte decimal, y el bit 63 almacena el signo del valor.

```
var x = 34.00;      // Número con decimales (punto).

var x = 123e5;     // Añade ceros a la derecha 12300000.

var y = 123e-5;    // Añade ceros a la izquierda 0.00123.

var x = 0xFF;      // Valor pasado como hexadecimal.
```

Con el método **toString()** es posible cambiar la base de representación del número.

```
var numero = 128;      // Inicialización en base decimal

numero.toString(16);   // Devuelve 80 en hexadecimal

numero.toString(8);    // Devuelve 200 en octal

numero.toString(2);    // Devuelve 10000000 en binario
```

El valor **Infinity** será devuelto por cualquier método que su valor exceda del máximo permitido. También devuelve este valor cuando se realizan divisiones entre cero.

```
var x = 2 / 0;        // x devolverá como resultado 'Infinity'.
```

La palabra reservada **NaN** indica que un valor no es un número, por tanto, si se intenta realizar operaciones entre textos, o números y textos el valor resultante será **NaN**.

```
var x = 100 / "Audi"; // El resultado de la división es NaN.
```

Sin embargo, la excepción se produce si se realiza un cálculo entre un número y un texto que representa un número que entonces si realiza el cálculo.

```
var x = 100 / "10"; // El resultado de la división es 10.
```

La función **isNaN()** comprueba si un valor es un número o no. El resultado devuelto es true o false. Si es una cadena de caracteres que representa un número también devuelve true. Este último aspecto es muy importante tenerlo presente ya que un número en formato texto puede trabajar como un número en formato numérico.

Los números se pueden declarar como objetos pero no se recomienda porque ralentizan la función.

```
var x = 500; // Número declarado correctamente
```

```
var y = new Number(500); // Declarado como objeto, no recomendable.
```

Las variables tipo **Number()** además de valores decimales en Javascript pueden almacenar valores binarios, octales y hexadecimales utilizando la siguiente notación:

- **Binarios:** empezando el número con **0b** seguido de los ceros y unos que lo formen. Por ejemplo: *let n = 0b101;* para representar el 5.
- **Octales:** empezando el número con un **0** seguido de los números entre 0 y 7 que lo formen. Por ejemplo: *let n = 017;* para representar el 15.
- **Hexadecimal:** empezando el número con un **0x** seguido de los números entre 0 y 9 y las letras entre A y F que lo formen. Por ejemplo: *let n = 0xA;* para representar el 10.

Propiedades de los números: Solo se puede acceder a estas propiedades desde el objeto **Number**.

Constante	Valor en Javascript	Descripción
Number.POSITIVE_INFINITY	Infinity	Infinito positivo: +∞
Number.NEGATIVE_INFINITY	-Infinity	Infinito negativo: -∞
Number.MAX_VALUE	1.7976931348623157e+308	Valor más grande
Number.MIN_VALUE	5e-324	Valor más pequeño
Number.MAX_SAFE_INTEGER (ES2015)	9007199254740991	Valor seguro más grande
Number.MIN_SAFE_INTEGER (ES2015)	-9007199254740991	Valor seguro más pequeño
Number.EPSILON (ES2015)	2⁻⁵²	Número muy pequeño: ε
Number.NaN	NaN	Not A Number

La diferencia entre **Number.MAX_VALUE** y **Number.MAX_SAFE_INTEGER** es que, el primero es el **valor máximo** que es posible representar en Javascript. Por otro lado, el segundo es el **valor máximo** para realizar cálculos con seguridad en Javascript.

Si necesitamos realizar operaciones de muy alta precisión existen librerías específicas como '[decimal.js](#)' o '[bigNumbers.js](#)' que facilitan esta tarea.

Métodos en Numbers (Números)

Las variables tipo 'number' pueden acceder a una serie de métodos. Se pueden declarar como objetos con 'new' pero es recomendable no hacerlo porque ralentiza el desarrollo de la función.

- **toString()**: Devuelve el número como una cadena de texto. NO confundir con la opción `toString()` que ofrece para cambiar la base de un número (decimal, binario, etc..).

```
var x = 123;  
  
x.toString(); // Número declarado y convertido a texto
```

- **toExponential()**: Convierte un número en forma exponencial. Método poco utilizado.

```
var x = 9.656;  
  
x.toExponential(2); // Devuelve con 2 dígitos, 9.66e+0  
x.toExponential(4); // devuelve con 4 dígitos, 9.6560e+0  
x.toExponential(6); // Devuelve con 6 dígitos, 9.656000e+0
```

- **toFixed()**: Devuelve una cadena con el número de decimales especificado en el método.

```
var x = 9.656;  
  
x.toFixed(0); // Redondea y devuelve 10  
x.toFixed(2); // Devuelve con 2 dígitos, 9.66  
x.toFixed(4); // Devuelve con 4 dígitos, 9.6560  
x.toFixed(6); // Devuelve con 6 dígitos, 9.656000
```

- **toPrecision()**: Devuelve la parte entera y la parte decimal del tamaño especificado, siempre empezando desde la izquierda.

```
var num = 13.3714;  
  
var n = num.toPrecision(3); // Devolverá 13,3  
var n = num.toPrecision(5); // Devolverá 13,371
```

- **valueOf()**: Devuelve el valor que contenga la variable. Es una función común a muchos objetos.

Se utiliza para saber el contenido de la variable.

```
var num = 15;
```

```
var n = num.valueOf(); // Devuelve el valor 15
```

Existen unos métodos globales, es decir, utilizables por todas las clases de JavaScript, que afectan a las variables tipo *Number* y que convierten las variables a números.

- **Number()**: Convierte las variables de Javascript a números siempre que representen un número.

Si el valor convertido no representase un valor numérico devolvería como resultado NaN. Es recomendable realizar el control de este tipo de conversiones antes de continuar con la ejecución del código. El método String() convierte números a texto en formato tipo texto.

```
x = Number(false); // Devuelve 0, 'true' seria 1.
```

```
x = Number("10"); // Devuelve 10, al ser número lo convierte
```

```
x = Number("10 20"); // Devuelve NaN, no reconoce un numero
```

```
// En una fecha devuelve el valor en milisegundos
```

```
x = Number(11/09/1976); // Devuelve 0.0006185335132703555
```

- **parseInt()**: Analiza una cadena y devuelve un número entero. Si el número contiene parte decimal esta no se convierte. Si el número no se puede convertir devuelve NaN.

```
X = parseInt("10"); // Devuelve 10
```

```
X = parseInt("10.33"); // Devuelve 10
```

```
X = parseInt("10 20 30"); // Devuelve 10
```

```
X = parseInt("años 10"); // Devuelve NaN (no es un número)
```

- **parseFloat()**: Analiza una cadena y devuelve un número entero o decimal si este tuviera parte decimal. Si hay varios números separados por espacios en blanco devuelve el primer número. Si el número no se puede convertir devuelve NaN.

```
X = parseFloat("10");           // Devuelve 10  
  
X = parseFloat("10.33");         // Devuelve 10.33  
  
X = parseFloat("10 20 30");     // Devuelve 10  
  
X = parseFloat("años 10");       // Devuelve NaN (no es número)
```



ARRAYS (MATRICES)

Un Array en JavaScript se utiliza para almacenar varios valores en una única variable. Esta variable se puede considerar “especial” porque almacena muchos valores. Los valores del Array pueden ser de diferente tipo, numérico, texto, etc. Para acceder a los valores de la matriz se realiza mediante el número de índice.

A continuación, se muestran diferentes formas de crear Arrays. La primera forma es la más óptima porque no ralentiza la ejecución, es una asignación directa en cada posición del array de tamaño fijo. Al crear objetos con **new** provocará que la rutina vaya más lenta.

```
var cars = ["Saab", "Volvo", "BMW"]; // Método literal de array.

var cars = new Array("Saab", "Volvo", "BMW"); // Crea array iniciándolo

var cars = new Array(); // Crea array sin elementos, método 'lento'.

var cars = new Array(5); // Crea array de 6 elementos.

var cars = []; // Crea array sin elementos, es lo más recomendable
```

El primer elemento de un Array se encuentra en la posición cero [0]. Esto hay que tenerlo presente a la hora de hacer los cálculos.

```
var cars = ["Saab", "Volvo", "BMW"];

document.getElementById("parrafo").innerHTML = cars[0]; // Result Saab
```

El operador **typeof** devuelve el valor primitivo (que no tiene ni propiedades ni métodos) del objeto, es decir, determina qué tipo de valor almacena una variable. A veces es necesario conocer el tipo de elemento para saber como tratarlo. Los valores devueltos pueden ser 'number', 'string', 'boolean'. El valor devuelto en el caso de los arrays será 'object'.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];

typeof fruits; // Devuelve 'object' porque es un array
```

Desde **ECMAScript 5** la clase array tiene el método **isArray()** que devuelve 'true' o 'false' en función de si es un array o no el elemento o variable evaluada.

```
Array.isArray(fruits); // Devuelve Verdadero o falso si es un array
```

Para crear la clase Array es importante tener en cuenta que se escribe la primera letra en mayúscula. Los arrays comienzan siempre en la posición 0. Un error frecuente es no acordarse que la primera posición del array es cero y no un uno.

El objeto Array y sus Propiedades (2):

- La propiedad ***length*** de la clase array es de lectura-escritura, es decir, permite conocer el tamaño de un Array o asignarle un tamaño. Si asignamos un tamaño que es menor se eliminarán valores, y si es mayor habrá espacios vacíos con el valor '*undefined*'.

La propiedad tiene la particularidad que devuelve el número de elementos del array empezando por 1 y no tomando como referencia el valor cero del array. Dicho de otra forma, informa del numero de elementos que contiene el array.

```
// Declaración de un array de tamaño.  
  
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
  
// Devuelve 4 que el número de elementos.  
var n = str.length;  
  
// Asignamos un nuevo tamaño al array  
str.length = 3;
```

- La propiedad ***prototype*** permite **añadir nuevos métodos y propiedades al objeto array**. Para ello es necesario declarar previamente la propiedad o el método sobre el objeto. Es una práctica poco frecuente añadir nuevas propiedades en los objetos de Javascript, pero es posible.

El siguiente ejemplo muestra como añadir una propiedad al objeto, como asignarle un valor, y como recuperar el valor para mostrarlo por pantalla.

Ejemplo 1: Creación de una propiedad nueva para el objeto Array

```
var dominio_comercial = new Array();
// Aquí declaramos una variable de tipo array.

dominio_comercial.prototype.dominio = ".org";
// Declaramos una propiedad nueva al objeto Array y le
asignamos un valor.

document.write(dominio_comercial.dominio);
// Muestra el valor de la nueva propiedad.
// Ya no es necesario escribir 'prototype'.
```

La diferencia entre los valores **null** y **undefined** es que **undefined** indica que la variable fue declarada pero ningún valor le fue asignado, mientras que **null** indica que existe un valor asignado a la variable, pero es nulo.

Arrays multidimensionales

Los **arrays multidimensionales** son un tipo de variables que en su interior almacenan arrays, es decir arrays dentro de arrays. El siguiente ejemplo muestra un array con dos posiciones y cada una de estas posiciones almacena un array en su interior con 3 y 4 números respectivamente.

Los arrays son objetos que se pueden convertir a texto, concretamente JSON, para que toda la información pueda ser enviada a través de Internet. Hay que recordar que los objetos no se pueden enviar en su formato original y deben ser convertidos a un formato exportable para su envío.

```
var categorias = [ "Terror", "Accion", "Comedia" ];

var peliculas = [ "Scream", "Rambo", "Gladiator", "Benhur", "Lucy" ];

var cine = [ categorias, peliculas ];

console.log( cine[1][3] ); // Resultado: Benhur.
```

Array Multidimensional 'Cine'	0 (categorías)	1 (películas)
0	Terror	Scream
1	Acción	Rambo
2	Comedia	Gladiator
3		Benhur
4		Lucy

Arrays asociativos:

Los **arrays asociativos** son aquellos arrays que almacenan sus valores en posiciones **identificadas por un texto**. Si el valor de la '*clave*' es un número no haría falta escribirlo, podemos decir que en esta situación estamos trabajando con un array normal a través de la indexación a partir de su numero de posición.

En estos dos ejemplos el primer array asociativo es para almacenar un único registro de datos, mientras que el segundo array almacena un conjunto de registros con todos sus valores. Esta notación o sintaxis es denominada JSON.

Para leer un array asociativo se puede realizar de forma mas cómoda con la sentencia **for...in** o **foreach**

```
let miArrayAssoc = {"clave1":"valor 1" , "clave2": 300 , ..etc..}

var peliculas = [ {titulo: "Superman", year: 2015, pais: "EUA"},  

                 { titulo: "Batman", year: 2016, pais: "Canada"},  

                 { titulo: "Superlopez", year: 2018, pais: "Spain"} ];

// Recorrer un array asociativo
for ( let index in peliculas) {  

    console.log(peliculas[index].titulo)  

}
```

Métodos En Arrays (Matrices)

El objeto Array dispone de una serie métodos para trabajar con la matriz.

- **toString():** Convierte una matriz a una cadena de texto. Este método no es exclusivo de la clase array y se utiliza en otros objetos. Por ejemplo, al convertir números a texto es posible determinar la base del valor de salida (binario, octal, o hexadecimal).

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];  
  
document.getElementById("demo").innerHTML = car.toString();  
  
// Resultado 'Audi, BMW, Mercedes, Nissan'
```

- **join():** Permite concatenar los elementos de un array **a partir de un carácter** especificado. Si no se especifica el carácter separador lo hace automáticamente con una coma.

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];  
  
document.getElementById("demo").innerHTML = car.join(" - ");  
  
// Resultado 'Audi - BMW - Mercedes - Nissan'
```

- **pop():** Elimina el último elemento del array y lo devuelve como resultado. El método modifica el tamaño del array. Si el array estuviese vacío devolvería el valor de '*undefined*'.

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];  
  
document.getElementById("demo").innerHTML = car.pop();  
  
// Resultado 'Audi, BMW, Mercedes'
```

- **push():** Añade al final del array un elemento. Se pueden añadir varios elementos a la vez, pero siempre separados por comas. El método devuelve como resultado la nueva longitud del array

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];  
  
document.getElementById("demo").innerHTML = car.push("Honda");  
  
// Resultado 'Audi, BMW, Mercedes, Nissan, Honda'
```

- **shift():** Elimina el primer elemento del array y modifica el tamaño del mismo. Devuelve como el elemento que ha sido eliminado.

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];

document.getElementById("demo").innerHTML = car.shift();

// Resultado 'BMW, Mercedes, Nissan'
```

- **unshift():** Añade al principio del array un elemento nuevo. Este método modifica el tamaño del array y devuelve como resultado la nueva longitud.

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];

document.getElementById("demo").innerHTML = car.unshift("Honda");

// Resultado 'Honda, Audi, BMW, Mercedes, Nissan'
```

- **delete():** Elimina el elemento con el índice seleccionado. La matriz convierte ese espacio en 'undefined' pero NO lo elimina de la matriz, solo afecta al valor.

```
var car = ["Audi", "BMW", "Mercedes", "Nissan"];

delete[2].car;

// Resultado 'Audi, BMW, undefined, Nissan'
```

- **splice():** (Empalmar) Permite realizar las acciones de borrar, insertar, y borrar-insertar (a la vez). Cuando se insertan valores el array desplaza el resto de elementos hacia la derecha.

- El primer parámetro determina la posición donde el elemento será añadido.
- El segundo parámetro determina cuantos valores se eliminarán hacia la derecha.
- El tercer parámetro determina el valor o los valores que serán añadidos.

```
var car = ["Audi", "BMW", "Nissan"];

car.splice(1,0,"honda"); // Inserta 'Audi, Honda, BMW, Nissan'

car.splice(1,1,"honda"); // Elim-Insert 'Audi, Honda, Nissan'

car.splice(0,2); // Elimina los dos primeros valores del array
```

- **slice()**: Crea un nuevo array con un subconjunto de elementos pertenecientes a otro array. Los parámetros pasados indican el índice inicial y final del subconjunto, aunque hay otras formas de extraer subconjuntos de un array. El índice inicial puede ser un número negativo con lo que la selección de elementos comenzaría desde el final. El array inicial se mantiene intacto.

```
var numeros = new Array(1,2,3,4,5,6,7,8,9,10);

var nums1 = numeros.slice(1,5); // Devuelve Array 2,3,4,5

var nums2 = numeros.slice(-4); // Devuelve Array 7,8,9,10

var nums3 = numeros.slice(4); // Devuelve Array 5,6,7,8,9,10
```

- **concat()**: Permite enlazar o concatenar arrays. El resultado que devuelve es la suma de los dos arrays y que deberán guardarse en otra variable para trabajar de forma más cómoda.

```
var marcas1 = ["Audi", "BMW", "Nissan"];

var marcas2 = ["Honda", "Toyota", "Mercedes"];

var marcasT = marcas1.concat(marcas2); // Enlaza los textos

document.getElementById("parrafo").innerHTML = marcasT;
```

- **indexOf()**: Este método devuelve la posición del valor buscado en el array realizando la búsqueda de izquierda a derecha y empezando por la posición 0. Si no encuentra el valor devuelve -1. El método **search()** realiza la misma acción con el mismo resultado.

```
var marcas1 = ["Audi", "BMW", "Nissan"];

marcas1.indexOf("BMW") // Devuelve el valor 1.
```

- **lastIndexOf()**: Devuelve la posición del elemento del array especificado realizando la búsqueda de derecha a izquierda. Si hubiese mas de una coincidencia en el array devolvería el último o mayor posición. El resultado devuelto es el valor del índice igual que en el caso anterior. Si no encuentra el valor devuelve -1.

```
var marcas1 = ["Audi", "BMW", "Nissan"];

marcas1.lastIndexOf("Audi") // Devuelve el valor 2.
```

- **sort():** Este método **ordena los elementos de un array de forma alfabética**, ascendente A-Z.

Ordena el array y devuelve el valor sobre el mismo array.

```
var marcas = ["Nissan", "BMW", "Audi"];
marcas.sort();      // Devuelve 'Audi, BMW, Nissan'
document.getElementById("demo").innerHTML = marcas;
```

- **reverse():** Ordena de forma inversa los elementos del array, descendente Z-A. En este caso mas que una ordenación inversa lo que realiza es "girar" el array a partir de la posición actual, es decir, si no está ordenado lo ordena descendentemente (Z-A) pero si ya estuviese ordenado lo devolvería de forma ascendente.

```
var marcas = ["BMW", "Nissan", "Audi"];
marcas.reverse();    // Devuelve 'Nissan, BMW, Audi'
document.getElementById("demo").innerHTML = marcas;
```

- **every():** Comprueba **si todos los elementos de una matriz pasan una determinada condición**, es decir, todos los valores han de pasar la prueba para que devuelva **true**. La particularidad es que se le pasa como parámetro una función que es la que ejecuta la condición. Es el equivalente la función Y donde se deben cumplir todas las condiciones.

```
var menu = [
  { nombre: 'Ceviche', precio: 20, pais: 'Perú' },
  { nombre: 'Tacos', precio: 10, pais: 'México' },
  { nombre: 'Pasta', precio: 50, pais: 'Italia' },
  { nombre: 'Queso', precio: 15, pais: 'México' }
]
```

```
var resultado = menu.every( plato => plato.precio <= 10);
// RESULTADO: False
```

- **some()**: Esta función comprueba que **al menos uno de los valores pasados cumple la condición**, y si es así, devuelve 'true'. Para que el resultado final de la función sea 'false' ningún valor supera la condición. Es el equivalente a la función O.

Ejemplo 1: Disponemos de un array de objetos con tres atributos y sus valores. El método **some()** utiliza en este caso una arrow function como parámetro que contiene para cada pasada el registro ('plato') y a continuación la condición que evalúa ('plato.precio'). El resultado del método devuelve 'True' porque al menos un valor de todos cumple la condición establecida.

```
var menu = [
    { nombre: 'Ceviche', precio: 20, pais: 'Perú' },
    { nombre: 'Tacos', precio: 10, pais: 'México' },
    { nombre: 'Pasta', precio: 50, pais: 'Italia' },
    { nombre: 'Queso', precio: 15, pais: 'México' } ]

var resultado = menu.some( plato => plato.precio <= 10);

// RESULTADO: True. El parámetro de la arrow function es el nombre que le dará el usuario a cada registro en cada una de las pasadas.
```

- **filter()**: Este método **crea una matriz con todos los valores que superan la condición**. Si es una matriz de objetos devuelve todos los objetos devueltos por el campo especificado.

Ejemplo 1: Disponemos de un array con 4 valores. El método 'miFuncion()' escribe dentro del documento HTML con la etiqueta 'innerHTML' el resultado del filtro. El método **filter()** llama a una función de forma recurrente tantas veces como elementos dispone el array. Dentro de esta función se comprueba si es valor el mayor o igual que 18. El resultado final es un array que contiene todos los valores que han superado el criterio.

```
// Devuelve el resultado 32, 33, 40 dentro de un array.

var edades = [32, 33, 16, 40];

function checkAdult(edad) {
    return edad >= 18;
}
```

```
function miFuncion() {  
    document.getElementById("parrafo").innerHTML =  
        edades.filter(checkAdult);  
}
```

- **map()**: Crea una nueva matriz con los resultados obtenidos al llamar a una función que ejecuta una operación sobre cada elemento de la matriz. El parámetro pasado es una función que ejecuta un cálculo.

```
// Resultado es un array nuevo con los valores 2, 3, 4, 5 que  
corresponden a la raíz cuadrada de cada número.
```

```
var numbers = [4, 9, 16, 25];  
  
function myFunction() {  
    document.getElementById("demo").innerHTML =  
        numbers.map(Math.sqrt);  
}
```

- **split()**: Este método permite **convertir una cadena de texto en una matriz**. Hay que especificar que carácter es el elemento separador para cada posición de la matriz (Array).

```
var txt = "Audi, Mercedes, BMW";  
  
var res = txt.split(",");  
  
// Convierte string a Array
```

- **Array.of()**: Convierte el contenido que se le pasa en la función **como un array formal**. En el siguiente ejemplo convierte los tres valores que son tres valores en formato texto en un array con 3 elementos.

```
var platillos = Array.of("carne", "tacos", "pasta");
```

- **Array.from()**: Convierte el contenido de un array o un string en un array formal, es decir, la conversión habrá dejado el objeto de tal forma que nos permitirá acceder a los métodos y propiedades de la clase array.

En el siguiente ejemplo disponemos de un código HTML con una caja y tres párrafos en su interior. A continuación, hay un código Javascript que realiza la misma acción y que carga los valores de dos formas diferentes con sus pequeñas diferencias.

- La primera opción 'Op1' carga los valores en un array, pero los métodos a los que puede acceder serán los de la clase string porque no se ha convertido 100% a un elemento del tipo array.
- La segunda opción 'Op2' además de cargar los datos en un array convierte el elemento a un array real permitiéndonos de esta forma acceder a todos los métodos disponibles del objeto array.

----- Código HTML -----

```
<body>  
  <div class="platos">  
    <p>Carne</p>  
    <p>Tacos</p>  
    <p>Pasta</p>  
  </div>  
  <script src="js/app.js"></script>  
</body>
```

----- Código Javascript -----

```
// Guarda todos los elementos <p> que se encuentran en <div> dentro de la variable 'plats'. En este caso los métodos accesibles desde la variable serán los de la clase string.
```

```
Op1: var plats = document.querySelectorAll('.platos p');
```

// En este caso también guarda todos los elementos <p> que se encuentran dentro de la etiqueta <div>, pero en este caso al hacer la conversión con el método **Array.from()** permite posteriormente acceder a los métodos del objeto array.

Op2: `var plats = Array.from(document.querySelectorAll('.platos p'));`

- **find()**: Este método (ES6) permite **realizar la búsqueda de un elemento dentro de un array**. El mismo método recorre todos los registros que haya dentro del array. El método devuelve el valor encontrado en caso que haya coincidencia con algún elemento del array, o '*undefined*' si ningún valor coincide con el valor buscado. Si hubiese más de un valor coincidente solo devolvería el primer valor encontrado y no una colección de valores en forma de array.

Ejemplo 1: Disponemos de un array declarado con 4 valores en su interior. El método **find()** en este caso recibe una arrow function donde se le pasa un parámetro y a continuación el criterio que tiene que evaluar. La variable '*pElegido*' almacena el resultado de la función.

// Declaración del array con 4 elementos.

```
var platos = ["carne", "tacos", "pasta", "tostadas"];
```

// Recorre array hasta encontrar el primer elemento coincidente.

```
var pElegido = platos.find( plato => plato == 'Tacos');
```



El parámetro de la arrow function es el nombre que le dará el usuario a cada registro en cada una de las pasadas.

El código de la arrow function realiza la comparación de elemento del array con el valor buscado.

Ejemplo 2: Tenemos un array con una colección de 4 objetos en su interior. La arrow function recibe como parámetro cada uno de los registros para cada una de las pasadas. El cálculo de la arrow function compara si el nombre de un registro coincide con la palabra 'Tacos'. En este caso si se produce coincidencia el resultado que devuelve es todo el registro de datos (nombre, precio y país).

```
var menu = [
    {nombre: 'Ceviche', precio: 20, pais: 'Perú'},
    {nombre: 'Tacos', precio: 10, pais: 'México'},
    {nombre: 'Pasta', precio: 50, pais: 'Italia'}
];

var pElegido = menu.find( plato => plato.nombre == 'Tacos');
// RESULTADO: "nombre:'Tacos', precio:10, pais:'México'".
```

- **findIndex()**: Este método (ES6) realiza la búsqueda de un elemento dentro de un array. En este caso el resultado de la operación devuelve la posición (índice) del array una vez ha localizado el valor. Si no hay coincidencia devuelve -1.

```
var platos = ["carne", "tacos", "pasta", "tostadas"];

// Devuelve el número 1 que corresponde a la 2º posición del array
var pElegido = platos.findIndex( plato => plato == 'tacos');
```

El objeto **boolean()** determina si un valor o una comparación es verdadero o falso.

```
Boolean(10 > 9); // Devuelve True, evalúa la condición.

Boolean(5 == "5"); // Devuelve True.

Boolean(5 === "5"); // Devuelve False. Valor y tipo no iguales.

Boolean("BMW" == "Audi")); // Devuelve False
```

La **copia del contenido de un array a otro array** no se puede realizar con asignación directa como se realizaría con una variable. En este caso como los valores siempre son referencias a memoria si hacemos una asignación directa entre dos arrays se podrá modificar desde el segundo array el contenido primer array.

```
var platosA = ["carne", "tacos", "pasta"];  
  
platosB = platosA; // Asignación directa del primer al segundo array  
  
platosB[0] = "pollo"; // Cambio de valor en el segundo array.  
  
console.log(platosA); // Resultado: ["pollo", "tacos", "pasta"]
```

La forma clásica de solucionar este problema es copiar los valores de un array a otro array, pero esto nos implicaría generar una sentencia for dentro de nuestro código. Este problema se puede solucionar fácilmente siempre que utilicemos el método '**assign()**' del objeto Object de Javascript. De esta forma tendremos la misma variable con los mismos valores pero en zonas de memoria diferente.

```
Object.assign(platosB, platosA); // PlatoB es = platoA.
```

Los arrays multidimensionales (objetos dentro de arrays o objetos dentro de objetos) no permiten la utilización de este método ya que solo son capaces de leer hasta el primer nivel de datos. Ante esta situación una buena forma para poder copiar elementos de este tipo es pasarlo a objeto JSON (que básicamente es un texto), y copiarlo a una variable para posteriormente devolverlo a objeto.

Desestructuración de arrays

La **desestructuración de arrays** consiste en realizar una asignación a la inversa desde un array hacia unas variables utilizando una sintaxis específica. En el siguiente ejemplo se realiza una asignación a una serie de variables a partir del contenido del array.

```
// 1ª Línea: Declaración del array con sus 4 elementos.  
// 2ª a 5ª Línea: Declaración de las 4 variables sin contenido.  
// 6ª Línea: Asignación a cada variable desde los elementos del array.
```

```
var platos = ["carne", "tacos", "pasta", "tostadas"];
```

```
var plato1 = null;  
var plato2 = null;  
var plato3 = null;  
var plato4 = null;
```

Si en la asignación de las variables con los datos del array utilizamos la palabra reservada **var** no será necesario declarar anteriormente todas las variables.

```
var [plato1, plato2, plato3, plato4] = platos;
```

Sentencia for...in

Otra forma de interactuar con arrays es mediante la sentencia **for ... in**. Este sistema es muy parecido al **for** clásico, pero es más abreviado y permite optimizar mejor el código. La gran ventaja de utilizar este tipo de sentencia es que **permite leer los datos de un array asociativo**, donde el índice pasa de ser un número a un texto.

```
var plato["primero"] = "carne";
```

El siguiente código tiene una variable declarada del tipo array con 5 elementos. La sentencia **for ... in** recorre el array tantas veces como elementos tiene ya que la variable '*index*' almacena la posición del array para cada uno de los elementos que dispone.

En el siguiente ejemplo la última línea ('*console.log*') muestra el valor del array a partir de la posición especificada.

```
var platos = ["carne", "tacos", "pasta", "pescado", "fruta"];

for ( let index in platos) {
    console.log(platos[index])
}
```

Sentencia for...of

Este caso es similar al bucle **for... in** aunque se diferencia de este porque al iterar con los elementos del array no devuelve el índice de la posición sino que devuelve el valor de la posición donde está iterando.

```
<script>
  let numeros = [1, 3, 5, 7, 9];

  for (let i in numeros) {
    console.log(i); // log -> 0, 1, 2, 3, 4
  }
</script>
```

For in

```
<script>
  let numeros = [11, 33, 55, 77, 99];

  for (let i of numeros) {
    console.log(i); // log -> 11, 33, 55, 77, 99
  }
</script>
```

For of

Sentencia Foreach

La sentencia **foreach** está pensada únicamente para recorrer arrays u objetos de forma más cómoda.

Puede tener hasta tres parámetros y los nombres de estos no son relevantes, pero si la posición que ocupan cuando se están informando a la función. No es obligatorio pasar todos los parámetros. A diferencia del 'for' la sentencia 'foreach' no puede recorrer una cadena de texto. Además 'foreach' recorre todo el array sin tener que especificar el tamaño del array. Cada parámetro realiza:

1. *elemento*: Hace referencia al valor del array en esa pasada de bucle según la posición.
2. *Índice*: Determina la posición dentro del array, es un número.
3. *arraycompleto*: Contiene todo el array por si se necesita.

Es frecuente cuando se invoca esta sentencia pasarle una función anónima como parámetro con los tres argumentos que acepta para poder trabajar interiormente con ellos en la función.

El siguiente código muestra diferentes formas de recorrer arrays y como estas formas han ido evolucionando hasta la utilización del método **forEach**. Este método recorre el array pasándole una función con los tres parámetros que acepta que son (valor, índice, array).

```
var lenguajes = ["JS", "CSS", "PHP"];
```

----- Método clásico para recorrer arrays -----

```
for (var i=0; i<lenguajes.length; i++) {  
    console.log(lenguajes[i]); // Resultado: Cada pasada es un valor  
}
```

----- Método forEach para recorrer arrays con función anónima -----

```
lenguajes.forEach( function( elemento, indice, arraycompleto ) {  
    console.log(elemento); // Resultado: JS (1ª pasada)  
    console.log(indice); // Resultado: 0  
    console.log(arraycompleto); // Resultado: JS, CSS, PHP  
}
```

----- Método forEach para recorrer arrays con arrow function -----

```
lenguajes.forEach(( elemento, indice, arraycompleto ) => {  
  
    document.write(elemento);      // Resultado: JS (1ª pasada)  
  
}); // No es obligatorio pasar los 3 parámetros
```



Diferencias entre for, forEach, for...in, for...of

Un bucle **for** que recorra la longitud del array accederá a todos los valores del array (también los *undefined*) y no accederá a las propiedades del array.

Un bucle **forEach** recorrerá todas las claves numéricas del array que no contengan un valor '*undefined*' y no accederá a las propiedades del array.

Un bucle **for/in** recorrerá todas las claves numéricas del array que no contengan un valor '*undefined*' y también todas las propiedades del array.

Ejemplo:

Un bucle **for** imprime los valores por consola:

- 0: rama
- 1: *undefined*
- 2: fruta

Un bucle **forEach** imprime los valores por consola sin tener en cuenta los valores '*undefined*':

- 0: rama
- 2: fruta

Un bucle **for/in** imprime los valores por consola incluyendo los valores de **arrays asociativos**:

- 0: rama
- 2: fruta
- Hoja: perenne
- Edad: 50

INTRODUCCION A EVENTOS

Los eventos son las diferentes acciones que pueden sucederle a un objeto o elemento. Por ejemplo, un evento se produce al hacer clic sobre un elemento (*onclick*) o al pasar por encima de una zona (*onmouseover*). Existen multitud de eventos en JavaScript y que son accesibles de las páginas HTML.

JavaScript permite ejecutar código cuando se detectan los eventos. Desde HTML se puede, mediante un evento, llamar a las funciones de JavaScript.

```
// Evento 'onclick' que lanza función date y escribe en <p> la hora actual
<button onclick="document.getElementById('demo').innerHTML = Date()">
    ¿Qué hora es?
</button>
```

```
//Se cambia el estilo del mismo elemento al salir después de pasar sobre él.
<span onmouseout="this.style.color='red'"> Salir elemento </span>
```

```
//Llama a la función 'displayDate()' al hacer clic sobre el elemento.
<button onclick="displayDate()"> Que hora es? </button>
```

Algunos de los eventos más utilizados son:

Event	Description
onchange	Un elemento HTML ha cambiado
onclick	El usuario <u>hace clic</u> sobre un elemento HTML
onmouseover	<u>Pasar sobre el elemento</u> con el ratón en HTML
onmouseout	<u>Salir de un elemento</u> con el ratón en HTML
onkeydown	El usuario <u>pulsa una Tecla</u> .
onload	El navegador <u>ha terminado de cargar la pagina</u>

DOM en EVENTOS

Desde el DOM es posible ejecutar métodos a partir de una serie de eventos. Los eventos son acciones que se producen en la pantalla cuando **el usuario está interactuando con ella**. Los eventos se ejecutan siempre en combinación con las funciones de Javascript.

Tipos de eventos

Cuando capturamos un evento recibimos por parámetro información sobre ese evento. Cada tipo de evento contiene unas propiedades con distinta información sobre el evento. Todos los eventos heredan de "Event" y por lo tanto tienen como mínimo sus mismas propiedades y métodos. A continuación se muestran los tipos principales de eventos en JS y sus propiedades más destacadas:

Tipo evento	Descripción	Propiedades destacadas
Event	Es el evento más general. Todos los elementos son tipo event y heredan sus propiedades.	<ul style="list-style-type: none"> • target: retorna el elemento que ha lanzado el evento. • currentTarget: Retorna el elemento al que se ha vinculado la función • type: retorna el tipo de evento.
UiEvent	Gestionan la información derivada de cambios en la interfaz del usuario, como puede ser el redimensionar la pantalla.	<ul style="list-style-type: none"> • view: retorna una referencia al objeto window que ha lanzado el evento.
TouchEvent	Gestionan la información cuando el usuario toca la ventana.	<ul style="list-style-type: none"> • touches: retorna cuantos dedos están tocando la pantalla.
FocusEvent	Contiene información sobre los cambios de foco entre los distintos elementos de la web.	<ul style="list-style-type: none"> • relatedTarget: retorna el elemento que ha perdido o adquirido el foco.
InputEvent	Contiene información sobre los cambios o interacciones con los inputs del HTML.	<ul style="list-style-type: none"> • data: retorna los caracteres insertados. • inputType: Información sobre el tipo de operación realizada (insertar texto, borradot, etc...)

KeyboardEvent	Contiene información sobre la interacción del usuario con el teclado.	<ul style="list-style-type: none"> •altKey, shiftKey, ctrlKey: retorna si alguna de las 3 teclas estaba pulsada cuando saltó el evento. •charCode: retorna el carácter pulsado •keyCode, code, key: retornan un código identificativo de la tecla pulsada. •repeat: retorna si la tecla se está manteniendo pulsada.
StorageEvent	Gestiona los cambios en el windows storage	<ul style="list-style-type: none"> •key: retorna la clave del ítem que se ha modificado. •newValue, oldValue: retorna el antiguo valor o nuevo del ítem modificado.
MouseEvent	Gestionan las acciones del raton y sus movimientos.	<ul style="list-style-type: none"> •altKey, shiftKey, ctrlKey: retorna si alguna de las 3 teclas estaba pulsada cuando saltó el evento. •Button, buttons: retorna un numero identificativo de que botón o botones han sido pulsados (0-Izquierdo, 1-Medio, 2-Derecho). •clientX, clientY: Retorna la posicion X e Y del raton respecto al navegador. •offsetX, offsetY: retorna la posición X e Y del elemento que ha lanzado el evento. •pageX, pageY: retorna la corrdenada X e Y del ratón respecto al documento. •movementX, movementY: retorna el desplazamiento X e Y del ratón.
DragEvent	Gestiona las acciones de arrastrar contenido por la web.	<ul style="list-style-type: none"> •dataTransfer: contienen la información del elemento que se está moviendo.

ClipboardEvent	Gestiona la información cuando el usuario corta/copia/pega información.	•clipboardData: contiene la información copiada.
PopStateEvent	Gestiona la información sobre los cambios en el historial.	•state: retorna información con una copia del historial.
ProgressEvent	Gestiona la información del progreso de carga de recursos externos.	•loaded: retorna la cantidad de contenido cargado. •total: reotrna la cantidad de contenido total que se debe cargar.
AnimationEvent	Contiene información del estado de las animaciones CSS	•animationName: retorna el nombre de la animación. •elapsedTime: retorna los segundos que lleva ejecutándose la animación.
TransitionEvent	Gestiona información cuando se produce cambio en el estado de las transiciones CSS	•animationName:reotrna nombre de la transición
ChangeEvent	Contiene los cambios que se produzcan en el hash de la url	•newURL: reotrna la nueva URL
WheelEvent	Gestiona la información cuando el usuario mueve la rueda del mouse.	•deltaX, deltaY, deltaZ: retorna la cantidad de scroll en los ejes X, Y, Z.

Eventos generales

Existen un tipo de eventos que no son específicos a ningún elemento en concreto sino que pueden afectar a varios elementos o son de dominio más genérico.

Tipo evento	Descripción
load	Se lanza cuando el evento es lanzado completamente, por ejemplo, el caso más habitual es realizar alguna acción tras cargar el 'document'. Ej: <code>window.addEventListener('load', miFunción, false)</code>
contextmenu	Se lanza cuando el clic derecho del ratón provoca que se lance un menú contextual.
copy/cut	Se lanza cuando se copia o corta el contenido. Genera por defecto un evento del tipo ClipboardEvent.
scroll	Se lanza cuando se produce un scroll sobre el elemento vinculado. Genera un evento del tipo UIEvent.
resize	Se relaciona directamente con el objeto window y se lanza cuando la ventana es redimensionada. Genera un evento del tipo UIEvent. Ej: <code>window.addEventListener('resize', miFunción, false)</code>

Eventos de ratón

Eventos	Descripción	DOM
onclick	El evento se ejecuta cuando el usuario hace <u>clic sobre el elemento.</u>	2
Ex: <code><button onclick="myFunction()"> Pulsar </button></code>		
oncontextmenu	El evento se produce cuando el usuario hace clic con el <u>botón secundario</u> en un elemento para abrir un menú contextual	3
Ex: <code><div oncontextmenu="myFunction()" contextmenu="mymenu"></code>		
ondblclick	El evento se produce cuando el usuario <u>hace doble clic</u> en un elemento	2
Ex: <code><p id="myP" ondblclick="DobleClic()"> Doble click </p></code> <code><script></code> <code>function DobleClic() {</code> <code>document.getElementById("myP").innerHTML = "Click-Click";</code> <code>}</code> <code></script></code>		
onmousedown	El evento se produce cuando el usuario <u>presiona un botón del mouse</u> sobre un elemento.	2
Ex: <code><p id="myP" onmousedown="mouseDown()"> Texto </p></code> <code><script></code> <code>function mouseDown() {</code>		

```

document.getElementById("myP").style.color = "red";

document.getElementById("myP").style.fontSize = "48px";

}

</script>

```

onmouseenter	El suceso se produce cuando <u>el puntero se mueve sobre un elemento</u> (cuando entra). Similar a onmouseover pero en este caso solo se ejecuta una vez al entrar.	2
---------------------	---	---

Ex: // Modifica el tamaño de la imagen al entrar en ella. La función recibe la imagen como parámetro. La forma de realizar esta acción es con la palabra reservada 'this'.

```



<script>

  function bigImg(x) {

    x.style.height = "64px";

    x.style.width = "64px"; }

</script>

```

onmouseleave	El evento se produce cuando <u>el puntero se mueve fuera de un elemento</u> , es decir, cuando sale del elemento. Similar a onmouseout pero no es igual porque este evento no afecta a los elementos hijos.	2
---------------------	---	---

Ex: // Modifica el tamaño de la imagen al salir de ella. La función recibe la imagen como parámetro. La forma de realizar esta acción es con la palabra reservada 'this'.

```

```

```

<script>

function normalImg(x) {

    x.style.height = "64px";

    x.style.width = "64px";

}

</script>

```

onmousemove	El evento ocurre cuando el puntero <u>se mueve mientras está sobre un elemento.</u>	2
--------------------	---	---

Ex: `<div onmousemove="myFunction()">Mover cursor sobre elemento</div>`

onmouseover	El evento se produce cuando <u>el ratón “entra” en el elemento o en cualquiera de sus elementos hijos.</u>	2
--------------------	--	---

Ex: ``

```

<script>

function bigImg(x) {

    x.style.height = "64px";

    x.style.width = "64px"; }

</script>

```

onmouseout	El evento se produce cuando <u>el ratón “sale” del elemento seleccionado</u> o cualquiera de sus elementos hijos. Es similar a <u>onmouseleave</u> pero aquí sí afecta a los elementos hijos.	2
-------------------	---	---

Ex: ``

onmouseup	El evento se produce cuando <u>se suelta (se libera) el botón izquierdo</u> del ratón cuando está pulsado sobre un elemento.	2
<p>Ex: <code><p onmouseup="mouseUp()"> Clica el texto </p></code></p> <pre> <script> function mouseUp(x) { document.getElementById("myP").style.color = "red";} </script> </pre>		
onwheel	El evento se desencadena cada vez que <u>se hace girar hacia abajo la rueda</u> del ratón.	3
<p>Ex: <code>// Escucha hasta que el evento 'wheel' salta y ejecuta la función.</code></p> <pre> document.getElementById("myDIV").addEventListener("wheel", Tamaño); function Tamaño() { this.style.fontSize = "35px"; } </pre>		

Eventos de teclado

Evento	Descripción	DOM
onkeydown	El evento se produce cuando se pulsa una tecla, es decir, en el <u>momento que la tecla desciende</u> . Se desencadena con todas las teclas del teclado.	2
Ex: <code><input type="text" onkeydown="teclapulsada()"></code> <pre> <script> function teclapulsada(x) { alert("Has pulsado una tecla"); } </script></pre>		
onkeypress	El evento se produce cuando se pulsa una tecla y <u>se mantiene pulsada</u> . Es muy similar a la opción anterior. Solo se desencadena con las teclas comunes como letras y números.	2
Ex: <code><input type="text" onkeypress="teclapulsada()"></code>		
onkeyup	El evento se produce cuando <u>se deja de pulsar una tecla</u> , es decir, en el momento que la tecla sube.	2
Ex: <code><input type="text" onkeyup="soltartecla()"></code> <pre> <script> function soltartecla() { var x = document.getElementById("idcampo"); x.value = x.value.toUpperCase(); } </script></pre>		

Los eventos de teclado tienen la particularidad que al ser ejecutados generan de forma automática sus propios objetos del tipo **KeyboardEvent** que incluye una serie de propiedades.

- key: Devuelve una cadena de caracteres que identifica la/s tecla/s que han lanzado el evento.
- ctrlKey: Valor booleano que dice si se ha pulsado tecla Ctrl.
- altKey: Valor booleano que dice si se ha pulsado tecla Alt.
- shiftKey: Valor booleano que dice si se ha pulsado tecla Shift
- metaKey: Valor booleano que dice si se ha pulsado tecla meta (Windows o Command).
- repeat: Valor booleano que dice si el usuario pulsa la tecla continuamente.

El siguiente ejemplo captura la propiedad 'key' del evento que ha lanzado la función (keydown). El evento NO es pasado como parámetro a la función, sino que se genera automáticamente al ejecutarse el evento. No todos los eventos tienen la característica de generar propiedades de forma automática cuando se ejecutan, pero los eventos de teclado si.

```
// Al cargar la página se prepara el "escuchador" de eventos.
```

```
Window.addEventListener("keydown", verTecla);
```

```
// La función se lanza al pulsar la tecla, y se accede a la propiedad  
'key' que se ha generado al lanzar el evento.
```

```
<script>
```

```
    function verTecla(evento) {
```

```
        var x = evento.key;
```

```
        document.getElementById("result").innerHTML = x;
```

```
}
```

```
</script>
```

NOTA: Si un usuario introduce texto desde un sistema táctil como una tableta, los eventos para teclas no podrán ser ejecutados. Para ello, ya hay unos eventos propios y específicos para este tipo de dispositivos.

Eventos de formulario

Event	Descripción	DOM
onblur	El evento ocurre cuando <u>pierde el foco</u> . Afecta todos los los elementos <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style>, y <title>.	2
	Ex: <code><input type="text" id="fname" onblur="myFunction()"></code>	
onfocus	El evento se ejecuta cuando <u>coge el foco</u> . Afecta todos los los elementos <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style>, y <title>.	2
	Ex: <code><input type="text" id="fname" onfocus="miFunction()"></code> <code></script> // Pinta de color amarillo el interior del elemento 'type'</code> <code>function myFunction(x) {</code> <code> x.style.background = "yellow";}</code> <code></script></code>	
onchange	El evento se produce cuando el contenido de un elemento de formulario, la selección o <u>el estado marcado han cambiado</u> (para <input>, <keygen>, <select> y <textarea>)	2
	Ex: <code><select id="mySelect" onchange="myFunction()"></code> <code> <option value="Audi">Audi</code> <code> <option value="BMW">BMW</code> <code></select></code>	
onfocusin	El evento se produce cuando el elemento <u>está a punto de coger el foco</u> . Es muy similar a onfocus. Se produce justo al hacer clic. Afecta todos los los elementos <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style>, y <title>.	2
	Ex: <code><input type="text" id="fname" onfocusin="myFunction()"></code>	

onfocusout	El evento se produce cuando <u>el elemento está a punto de perder el foco</u> . Es muy similar a onblur. Se produce justo al hacer clic.	2
Ex: <code><input type="text" id="fname" onfocusout="myFunction()"></code>		
oninput	El evento ocurre cuando un elemento <u>obtiene entrada del usuario por teclado</u> , es decir que se escribe en el campo.	3
Ex: <code><input type="text" id="fname" oninput="myFunction()"></code>		
oninvalid	El evento se produce cuando <u>un elemento no es válido</u> . Por ejemplo un campo requerido no se introduce nada.	3
Ex: <code><input type="text" id="fname" oninvalid="myFunction()"></code>		
onreset	El evento se produce cuando <u>se resetea el formulario</u> .	2
Ex: <code><form onreset="myFunction()"></code>		
onsearch	El evento se produce cuando el usuario escribe algo en un campo de búsqueda (para <code><input = "search"></code>). El evento salta cuando se pulsa tecla Enter después de escribir el texto buscado.	3
Ex: <code><input type="search" id="fname" onsearch="myFunction()"></code>		
onselect	El evento se produce <u>después</u> de que el usuario seleccione algún texto (para <code><input></code> y <code><textarea></code>)	2
Ex: <code><input type="text" value="Texto" onselect="myFunction()"></code> <code></script></code> <code>function myFunction() {</code> <code> alert("Has seleccionado algun texto");</code> <code>}</code> <code></script></code>		

onsubmit	El evento ocurre cuando se envían los datos de un formulario	2
<p>Ex: <form onsubmit="myFunction()"></p> <p> Introduce el nombre: <input type="text"></p> <p> <input type="submit"></p> <p> </form></p> <p></script></p> <pre>function myFunction() { alert("Elementos de formulario enviados"); } </script></pre>		

Eventos de objetos

Evento	Descripción	DOM
onabort	<p>El evento ocurre cuando <u>la carga de un recurso ha sido abortada</u>. Solo está soportado por Microsoft Edge así que no es muy relevante este evento.</p>	2
	Ex: <code></code>	
onbeforeunload	<p>El evento <u>ocurre antes de que el documento esté a punto de descargarse</u>. Muestra un mensaje de información al usuario de confirmación por si quiere quedarse o salir de la página actual.</p>	2
	Ex: <code></code>	
onerror	<p>El evento se produce cuando se produce un error al cargar un archivo externo, por ejemplo, una imagen.</p>	2
	Ex: <code></code>	
onhashchange	<p>El evento se produce cuando se ha producido <u>algún tipo de cambio en un ancla</u> en la página. Cambiar un anclaje se realiza con el objeto Location (location.hash o location.href) y etiqueta <body>.</p>	3
	Ex: <code><body onhashchange="myFunction()"></code>	
onload	<p>El evento se produce cuando un objeto se ha cargado. Con mayor frecuencia se utiliza con la etiqueta <body> aunque se puede ejecutar con cualquier objeto.</p>	2
	Ex: <code><body onload="myFunction()"></code>	

onpageshow	Este evento es muy similar al evento onload. La diferencia radica que este evento <u>se ejecuta únicamente la primera vez que se carga la página</u> web. Es posible que la página esté almacenada en la memoria caché del equipo y esto haría que este evento no se ejecutase. Para actualizar una página desde el servidor es CTRL+F5.	3
Ex: <body onpageshow="myFunction()">		
onpagehide	El evento ocurre cuando <u>el usuario se aleja de una página web</u> . Alejarse de una página web es, por ejemplo, hacer clic en un enlace, actualizar la página, enviar un formulario, cerrar el navegador. Se utiliza en vez del evento onunload ya que este último hace que la página no se almacene en caché.	3
Ex: <body onpagehide="myFunction()">		
onunload	El evento se ejecuta generalmente cuando se cierra la ventana del navegador, un clic en un enlace, se actualiza la página, o envía los datos de un formulario. Se utiliza con la etiqueta <body>.	2
Ex: <body onunload="myFunction()">		
onscroll	El evento se produce cuando <u>se desplaza la barra de desplazamiento</u> de un elemento.	2
<pre> <script> //Captura objeto y se queda esperando hasta que se hace scroll document.getElementById("box").onscroll = function() {myFunction()}; function myFunction() { document.getElementById("box").innerHTML = "Hiciste scroll.";} </script> </pre>		

onresize	El evento se produce <u>cuando se cambia el tamaño de la vista de documento.</u>	2
Ex: <body onresize="myFunction()">		



Eventos de portapapeles

Evento	Descripción	DOM
oncopy	El evento ocurre cuando <u>el usuario copia el contenido de un elemento.</u>	
	<pre><script> // Captura objeto y se queda esperando hasta que se hace copy document.getElementById("myinput").oncopy = function() { myFunction(); function myFunction() { alert("Texto copiado"); } </script></pre>	
oncut	El evento ocurre cuando <u>el usuario corta el contenido de un elemento.</u>	
	<pre><script> document.getElementById("myinput").oncut = function() { myFunction(); function myFunction() { alert("Texto cortado"); } </script></pre>	
onpaste	El evento ocurre cuando <u>el usuario pega algún contenido en un elemento.</u>	
	<pre><script> document.getElementById("myinput").onpaste = function() { myFunction(); function myFunction() { alert("Texto pegado"); } </script></pre>	

Eventos multimedia

Evento	Descripción	DOM
onabort	El evento se produce cuando la carga de un elemento multimedia se aborta	3
oncanplay	El evento se produce cuando el navegador puede comenzar a reproducir el medio (cuando se ha almacenado lo suficiente como para comenzar).	3
oncanplaythrough	El evento se produce cuando el navegador puede reproducir a través del medio sin detenerse para el almacenamiento en búfer.	3
ondurationchange	El evento ocurre cuando se cambia la duración del medio. Es una forma de detector que el video ha sido descargado correctamente.	3
onemptied	El evento se produce cuando algo malo ocurre y el archivo multimedia de repente no está disponible (como desconexión inesperada)	3
onended	El evento se <u>produce cuando los medios de comunicación han llegado al final</u> (útil para mensajes como "gracias por escuchar").	3
onerror	El evento se produce cuando <u>se produce un error durante la carga</u> de un archivo multimedia.	3
onloadeddata	El evento se produce <u>cuando se cargan datos multimedia</u> .	3
onloadedmetadata	El evento ocurre <u>cuando se cargan metadatos</u> (como dimensiones y duración).	3
onloadstart	El evento ocurre <u>cuando el navegador comienza a buscar el medio</u> especificado.	3

onpause	El evento se produce <u>cuando el medio de comunicación es pausado</u> por el usuario.	3
onplay	El evento se produce <u>cuando el medio se ha iniciado</u> o ya no se detiene.	3
onplaying	El evento se produce <u>cuando el medio se está reproduciendo después de haber sido pausado</u> o detenido para el almacenamiento en búfer.	3
onprogress	El evento se produce <u>cuando el navegador está en el proceso de obtener los datos multimedia</u> (descarga de los medios).	3
onratechange	El evento ocurre cuando se cambia la velocidad de reproducción del medio.	3
onseeked	El evento ocurre <u>cuando el usuario termina de mover / saltar a una nueva posición</u> en el medio	3
onseeking	El evento ocurre <u>cuando el usuario comienza a mover / saltar a una nueva posición</u> en el medio.	3
onstalled	El evento se produce cuando el navegador está intentando obtener datos de medios, pero los datos no están disponibles.	3
onsuspend	El evento se produce cuando el navegador no está intencionadamente obteniendo datos multimedia.	3
ontimeupdate	El evento ocurre cuando la posición de reproducción ha cambiado (como cuando el usuario avanza rápidamente a un punto diferente en el medio).	3
onvolumechange	El evento ocurre <u>cuando el volumen del medio ha cambiado</u> (incluye ajustar el volumen a "silenciar")	3
onwaiting	El evento se produce <u>cuando el medio se ha detenido, pero se espera que se reanude</u> (como cuando el medio se detiene para almacenar más datos).	3

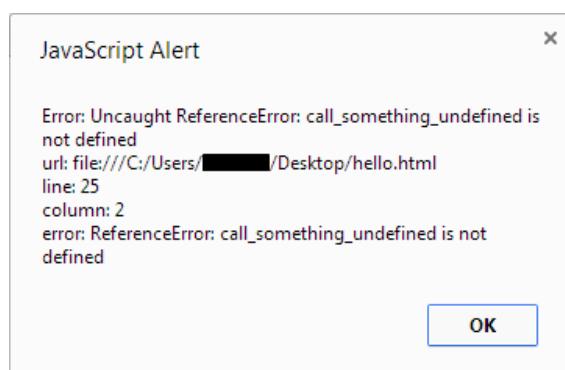
Evento Error

En ocasiones podemos encontrar errores que no los hemos generado nosotros programando nuestro código ya que dependen de factores externos, por ejemplo, recursos inaccesibles, librerías desactualizadas, etc...

El evento Error que está disponible en varias API y crea su propio objeto Event llamado ErrorEvent para transmitir información a la función. El objeto incluye las siguientes propiedades:

- error: Devuelve un objeto con información sobre el tipo error.
- message: Devuelve una cadena de caracteres que describe el error.
- lineno: Esta propiedad devuelve la línea en el documento donde ha ocurrido el error.
- colno: Devuelve la columna donde comienza la instrucción que ha producido el error.
- filename: La propiedad devuelve la URL (dirección servidor) del archivo que ha dado error.

```
function mostrarError(evento) {  
    console.log("Error: " + evento.error);  
    console.log("Error: " + evento.message);  
    console.log("Error: " + evento.lineno);  
    console.log("Error: " + evento.colno);  
    console.log("Error: " + evento.filename);  
}  
  
// Se pone a la escucha para ejecutarse si se produce un error.  
  
window.addEventListener('error', mostrarError);
```



DOM EventListener()

El método **addEventListener()** es un **detector de eventos** ("escuchador") o controlador de eventos. Los controladores de eventos facilitan la creación de código. Los eventos se pueden generar de **3 formas diferentes**:

1. Desde HTML: Es el método más rápido de preparación de eventos. Se realiza desde el mismo código HTML introduciendo el evento dentro de la etiqueta. No se recomienda este sistema porque es contrario al propósito principal de HTML5 que determina que HTML es para estructura del documento, CSS para el estilo o presentación, y Javascript para funcionalidad. Por tanto cada una de las zonas es mejor tenerlas perfectamente separadas.

```
// Evento 'onfocus' como atributo dentro de la etiqueta input.
```

```
<input type="text" id="fname" onfocus="miFunction()">
```

```
// Párrafo con evento doble-clic dentro de la etiqueta que lanza la función y escribe dentro del párrafo el texto '2-Click'.
```

```
<p id="myP" ondblclick="DobleClic()> Doble click </p>
```

```
<script>
```

```
function DobleClic() {
```

```
document.getElementById("myP").innerHTML = "2-Click"
```

```
}
```

```
</script>
```

2. Capturando el objeto: En este caso se hace referencia al objeto o elemento HTML con el método **getElementById()** y posteriormente se asigna el evento y se deja preparado para cuando se realice el evento o acción que ejecuta el código. Esta forma tiene algunos inconvenientes como por ejemplo **no poder asignar varios eventos a un mismo elemento**, o asignar el mismo evento varias veces.

// Captura el elemento con id 'p1' que al hacer clic lanza una función anónima que ejecuta un método.

```
<script>
```

```
document.getElementById("p1").onclick = function() {myFunction()};
```

```
function myFunction() {
```

```
    document.getElementById("p1").innerHTML = "Párrafo nuevo";
```

```
}
```

```
</script>
```

Estas dos primeras formas de programar eventos utilizan siempre el prefijo 'on' delante del evento, por ejemplo, 'onclick', 'onkeypress', 'onmouseup', etc... El tercer sistema explicado a continuación no utiliza el prefijo 'on' para declarar los eventos.

3. Controlador de eventos: Este sistema, similar al anterior, precisa del método **addEventListener()** que **deja en escucha al objeto** para un evento determinado. Permite aplicar varios eventos de forma fácil a un mismo elemento y el mismo evento varias veces a un mismo elemento.

```
// Captura botón 'input' formulario y ejecuta función 'validar()'.
```

```
document.getElementById("form1").addEventListener('submit', validar, false);
```

```
// Captura el mismo objeto para cargar dos eventos diferentes.
```

```
document.getElementById('text1').addEventListener('focus', tomafoco, false);
```

```
document.getElementById('text1').addEventListener('blur', fuerafoco, false);
```

```
// Cambia de color el objeto cuando toma el foco.
```

```
function tomafoco() {
```

```
    document.getElementById('text1').style.color = '#ff0000'; }
```

```
// Cambia de color cuando pierde el foco.
```

```
function fuerafoco() {
```

```
    document.getElementById('text1').style.color = '#000000'; }
```

El método **addEventListener()** dispone 3 parámetros que establecen la forma de ejecución del método.

1. El primer parámetro establece el tipo de evento que se desea configurar ('clic', 'mouseover', 'load', etc..). A diferencia de los otros métodos explicados anteriormente, el prefijo 'on' que se utilizaba para llamar a los eventos desaparece con este nuevo sistema.
2. El segundo parámetro determina la función que se ejecutará cuando se produzca el evento. Hay que tener presente que esta función no admite parámetros.
3. El tercer parámetro es un valor booleano que especifica si se debe utilizar burbujeo de eventos o captura de eventos. Este último parámetro hace referencia a la propagación del evento y normalmente tiene el valor '*false*'.

El método **removeEventListener()** elimina los controladores de eventos que han sido añadidos a un elementos con el método **addEventListener()**. Es poco frecuente eliminar un 'listener' pero en determinadas aplicaciones es necesario y este método nos da la opción.

Por ejemplo, un controlador de eventos que al pasar sobre un elemento active un banner publicitario. En este caso solo se desea que se muestre una vez y no siempre que pase sobre el elemento. Para ello se lanza el evento y a continuación se elimina el controlador para este evento de forma que no se volverá a ejecutar.

```
document.getElementById("box").removeEventListener("clic", miFuncion);
```

Propagación de evento

Cuando se produce un evento sobre un elemento en muchas ocasiones se produce también sobre sus padres o hijos. Por ejemplo, si clicamos sobre un DIV también estaremos clicando sobre el BODY que irremediablemente contiene el DIV. Y si el DIV y el BODY tuvieran ambos asociados un evento onclick, se ejecutarían ambos? Y en caso afirmativo, cuál se ejecutaría antes? Todo éste comportamiento es lo que entenderemos como “la propagación del evento”.

En JavaScript cuando se produce un evento éste evento **se propaga dos veces por todos los elementos** a los que afecte a través del árbol del DOM. Inicialmente, en la llamada “fase de captura” se propaga desde el nodo raíz “document” hasta el nodo más alejado al que afecte. Y posteriormente en la llamada “fase de bubbling” se propaga desde el nodo más alejado afectado por el evento hasta document.

Cuando nosotros vinculamos un evento con “addEventListener” podemos indicarle 3 parámetros, el 3º contendrá un valor “true” para ejecutar la función en la **“fase de captura”** y un valor “false” para ejecutar la función en la **“fase de bubbling”**.

Los dos siguientes ejemplos muestran como trabaja cada una de las fases:

Ejemplo 1: (Fase captura). En el siguiente ejemplo hemos añadido tres eventos “onclick” que van a ser lanzados cada vez que el usuario clique sobre el H1. Al ser añadidos con el parámetro “true” los hemos vinculado en la “fase de captura”. Por ello el orden de ejecución será: BODY, DIV y H1.

Resultado:

1. click Body capture!
2. click DIV capture!
3. click H1 capture!

Documento HTML

```
<html>
<head>
  <!-- Importante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <div id="elDiv">
    Propagacion
    <h1 id="elH1">Event</h1>
  </div>
</body>
</html>
```

Documento JavaScript dom.js

```
let body = document.body;
let div=document.getElementById("elDiv");
let h1=document.getElementById("elH1");
//vinculamos los eventos en la "fase de captura"
body.addEventListener("click",clickBody,true);
div.addEventListener("click",clickDiv,true);
h1.addEventListener("click",clickH1,true);

function clickBody(){
  console.log("click Body Capture!");
}
function clickDiv(){
  console.log("click Div Capture!");
}
function clickH1(){
  console.log("click H1 Capture!");
}
```

Ejemplo 2: (Fase Bubbling). En el ejemplo hemos añadido tres eventos “onclick” que van a ser lanzados cada vez que el usuario clique sobre el H1 pero ésta vez son añadidos con el parámetro “false” y por lo tanto los vincularemos en la “fase de bubbling”. Por ello el orden de ejecución será: H1, DIV, BODY.

Esta forma es la más habitual para ser utilizada ya que empieza a ejecutar las funciones sobre el elemento que se encuentra mas cercano al usuario y desde ahí va expandiéndose hasta llegar al elemento body. En este caso una vez finalizada la ejecución del código se debería interrumpir la propagación del evento con **'stopPropagation()'** para que no ejecute también funciones de sus elementos padre siempre y cuando compartieran el mismo evento.

Resultado:

4. click H1 capture!
5. click DIV capture!
6. click Body capture!

Documento HTML

```
<html>
<head>
  <!-- Importante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <div id="elDiv">
    Propagacion
    <h1 id="elH1">Event</h1>
  </div>
</body>
</html>
```

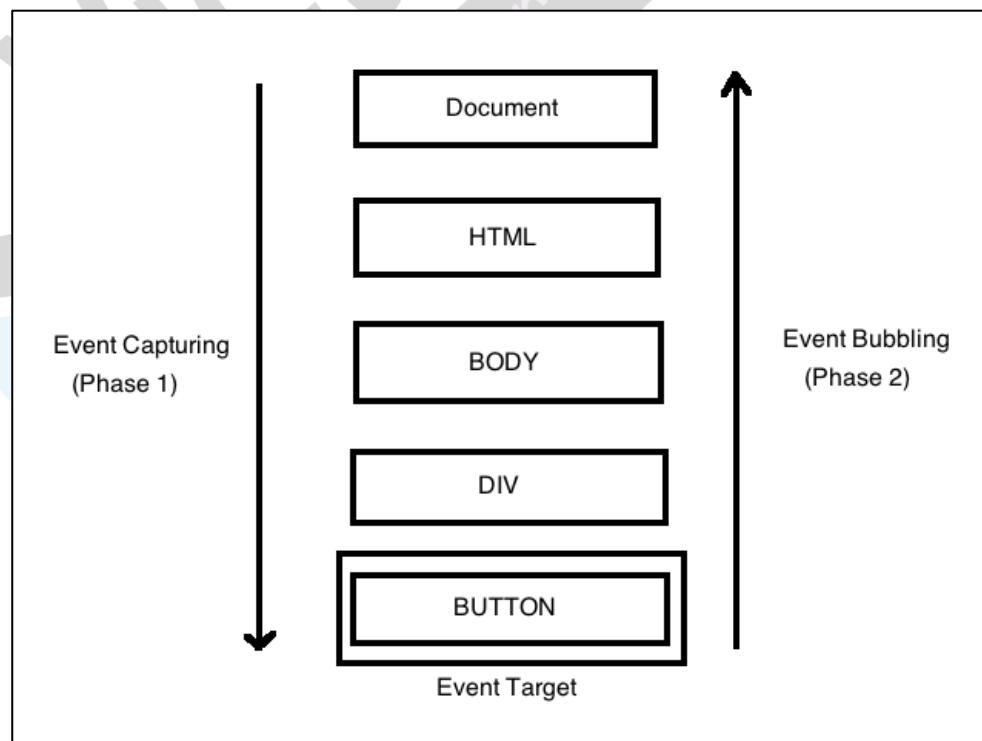
Documento JavaScript dom.js

```
let body = document.body;
let div=document.getElementById("elDiv");
let h1=document.getElementById("elH1");

//vinculamos los eventos en la "fase de captura"
body.addEventListener("click",clickBody, false);
div.addEventListener("click",clickDiv, false);
h1.addEventListener("click",clickH1, false);

function clickBody(){
  console.log("click Body Capture!");
}
function clickDiv(){
  console.log("click Div Capture!");
}
function clickH1(){
  console.log("click H1 Capture!");
}
```

Como normalmente se suele dar preferencia al elemento más alejado del “document”, lo más común es capturar los eventos en la “fase de bubbling” (*false*).



Detener eventos

Cuando se ejecuta una función lanzada por un evento, podemos utilizar el objeto "Event" que recibe como parámetro para detener la propagación del evento. De ésta forma evitaremos que se ejecuten el resto de funciones vinculadas con el evento que aún no se hayan ejecutado.

Para hacerlo se debe utilizar el método "**stopPropagation()**" implementado en el objeto "Event".

Ejemplo: En el siguiente ejemplo si clicamos encima del DIV o del H1 no se ejecutará la función "clickBody()" ya que hemos detenido la propagación del evento y las funciones se han vinculado en la fase de Bubbling.

Documento HTML

```
<html>
<head>
  <!-- Importante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <div id="elDiv">
    Propagacion
    <h1 id="elH1">Event</h1>
  </div>
</body>
</html>
```

Documento JavaScript dom.js

```
let body = document.body;
let div=document.getElementById("elDiv");
let h1=document.getElementById("elH1");

//vinculamos los eventos en la "fase de captura"
body.addEventListener("click",clickBody, false);
div.addEventListener("click",clickDiv, false);
h1.addEventListener("click",clickH1, false);

function clickBody(){ console.log("click Body Capture!");
}
function clickDiv(){
  //detenemos la propagación del evento
  evt.stopPropagation();
  console.log("click Div Capture!");
}
function clickH1(){ console.log("click H1 Capture!");}
```

En muchas ocasiones hay eventos que tienen acciones predeterminadas en el navegador. La más típica es el click derecho del ratón que abre un menú contextual. Desde JavaScript también podemos evitar las acciones por defecto utilizando el método "**preventDefault()**" implementado en el mismo "Event".

Ejemplo: En el siguiente ejemplo evitamos que se muestre el menú contextual.

Documento HTML

```
<html>
<head>
  <!-- Importante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <p>Evita el menú
    contextual</p>
</body>
</html>
```

Documento JavaScript dom.js

```
let body = document.body;
//contextmenu es el evento que se lanza al mostrar el
menu contextual
body.addEventListener("contextmenu",evitaMenu,true);

function evitaMenu(evt){
  //preventDefault evita las acciones por defecto del
navegador
  evt.preventDefault();
}
```

Ejemplo de utilización de addEventListener()

El siguiente código muestra como dos elementos de formulario de tipo texto se ponen a escuchar desde la función **inicio()** al cargar la página (load) para ir cambiando de color de fondo cuando se tenga o no el foco sobre el elemento.

```
<script type="text/javascript">

  //al cargar la ventana se lanza la función inicio.
  addEventListener('load',inicio,false);

  //se declaran las 4 funciones que se utilizaran

  function inicio()  {

    document.getElementById('text1').addEventListener('focus',tomarfoco1,false);
    document.getElementById('text2').addEventListener('focus',tomarfoco2,false);
    document.getElementById('text1').addEventListener('blur',perderfoco1,false);
    document.getElementById('text2').addEventListener('blur',perderfoco2,false);

  }

  function tomarfoco1()
  { document.getElementById('text1').style.color='#ff0000'; }
```

```
function tomarfoco2()  
  
{ document.getElementById('text2').style.color='#ff0000'; }  
  
  
function perderfoco1()  
  
{ document.getElementById('text1').style.color='#000000'; }  
  
  
function perderfoco2()  
  
{ document.getElementById('text2').style.color='#000000'; }  
  
</script>  
  
<body>  
  
// Dos cuadros de textos, con su id para poder ser invocados.  
  
<input type="text" id="text1" name="text1" size="30">  
  
<input type="text" id="text2" name="text2" size="30">  
  
</body>
```

MATH

El objeto '**Math**' permite realizar tareas matemáticas sobre los números. Esta clase es de tipo contenedor y tiene diversas propiedades y métodos.

Hasta ahora cada vez que queríamos hacer algo con una clase debíamos instanciar un objeto de esa clase y trabajar con el objeto y en el caso de la clase Math se trabaja directamente con la clase. Esto se permite por que las propiedades y métodos de la clase Math son lo que se llama propiedades y métodos de clase y para utilizarlos se opera a través de la clase en lugar de los objetos.

Dicho de otra forma, para trabajar con la clase Math no deberemos utilizar la instrucción new y utilizaremos el nombre de la clase para acceder a sus propiedades y métodos.

- Propiedades de Math: **E, LN10, LN2, LOG10E, LOG2E, PI, SQRT1_2** (raíz cuadrada de un medio), SQRT2 (raíz cuadrada de 2). De estas propiedades las más utilizadas serían PI y SQRT. El resto son propiedades menos utilizadas. Hay que tener presente que se escriben en mayúsculas. Al invocar a la propiedad no lleva paréntesis porque no es un método.

Propiedad	Descripción
E	Devuelve el número de Euler (aproximadamente 2.718)
LN2	Devuelve el logaritmo natural de 2 (aproximadamente 0,693)
LN10	Devuelve el logaritmo natural de 10 (aproximadamente 2.302)
LOG2E	Devuelve el logaritmo base-2 de E (aproximadamente 1.442)
LOG10E	Devuelve el logaritmo base-10 de E (aproximadamente 0.434)
PI	Devuelve el valor PI
SQRT1_2	Devuelve la raíz cuadrada de 1/2 (aproximadamente 0.707)
SQRT2	Devuelve la raíz cuadrada de 2 (aproximadamente 1.414)

```
document.write (Math.cos(2 * Math.PI)); // Calcula el coseno.  
  
document.write (Math.SQRT1_2); // Calcula raíz cuadrada de 1/2.  
  
document.write (Math.SQRT2); // Devuelve la raíz cuadrada 2.  
  
document.write (Math.cos(2 * Math.PI)); // Invoca a PI.
```

Métodos Objeto Math

- Metodos de la clase Math: abs(), acos(), asin(), atan(), atan2(), ceil(), cos(), exp(), floor(), log(), max(), min(), pow(), random(), round(), sin(), sqrt(), tan(). A continuación, se muestra un ejemplo de cada método.

- **abs():** Valor absoluto de un número.

```
var numabs = Math.abs(-45); // Convierte valor a positivo.
```

- **ceil():** Redondea al entero superior más próximo. Por ejemplo, ceil(3) vale 3, ceil(3.4) es 4. Redondea un valor hacia arriba, el siguiente entero.

```
var redexceso = Math.ceil(4.25); // Devolverá el valor 5.
```

- **exp():** Devuelve E elevado a la potencia dada. Devuelve el valor del número e (constante de Euler, aproximadamente 2,718) elevada al exponente dado por el argumento. Si el argumento fuera no entero será convertido a numérico siguiendo las reglas de las funciones parseInt() o parseFloat().

- **floor():** Redondea número decimal al valor entero inferior más próximo.

```
var redexceso = Math.floor(4.75); // Devolverá el valor 4.
```

- **log():** Devuelve el logaritmo de un número.

```
var logaritmo = Math.log(1000); // Devuelve el valor 6.90775527898213
```

- **max():** Devuelve el número mayor de un conjunto de valores pasados.

```
var mayor = Math.max( 12, 5, 7, 9 ); // Devuelve valor mayor, 12.
```

- **min():** Devuelve el valor más pequeño de un conjunto de valores especificados.

```
var menor = Math.min( 12, 5, 7, 9 ); // Devuelve el valor menor, 5.
```

- **pow():** Eleva el primer argumento a la potencia del segundo.

```
var potencia = Math.pow( 2, 4 ); // Potencia devolverá el valor 16.
```

- **random():** Devuelve un valor decimal comprendido entre 0 (inclusivo) y 1 (exclusivo).

```
var azar = Math.random() * 10; // Devuelve aleatorio * 10
```

- **round():** Devuelve un número al entero más próximo.

```
var entero1 = Math.round( 4.25 ); // Devuelve 4
```

```
var entero2 = Math.round( 4.65 ); // Devuelve 5
```

- **fround():** (ES2015). El método fround () devuelve la representación flotante más cercana (precisión simple de 32 bits) de un número.

```
var entero1 = Math.fround( 4.25 ); // Devuelve 4
```

```
var entero2 = Math.fround( 4.65 ); // Devuelve 5
```

- **trunc():** (ES2015). Devuelve un número al entero sin decimales y sin redondeo.

```
var entero1 = Math.trunc( 4.25 ); // Devuelve 4
```

```
var entero2 = Math.trunc( 3.99 ); // Devuelve 3
```

- **sqrt():** Calcula la raíz cuadrada de un número.

```
var raiz = Math.sqrt( 49 ); // La variable raíz contendrá el valor 7.
```

- **acos()**: Devuelve el arco-coseno de un número.

```
var arco = Math.acos( 1 ); // La variable arco contendrá el valor 0.
```

- **asin()**: Devuelve el arcoseno de un número.

```
var arco = Math.asin( 1 );
```

// La variable arco contendrá el arco cuyo seno es 1, o sea, 1.57 o lo que es lo mismo pi / 2 radianes.

- **cos()**: Devuelve el coseno de un número.

```
var coseno = Math.cos(Math.PI/2 );
```

// La variable coseno contendrá el valor 0, que es el coseno de pi/2 radianes (90º).

- **sin()**: Devuelve el seno de un número.

```
var seno = Math.sin(Math.PI/2 )
```

// La variable seno contendrá el valor 1, que es el seno de pi/2 radianes (90º).

- **atan()**: Devuelve el arco tangente de dos valores. El valor es un grado.

```
var tangente = Math.tan(Math.PI/4 )
```

// La variable tangente contendrá el valor 1, es la tangente de pi/4 radianes (45º).

Librería	Descripción
Math.js	Librería matemática de propósito general.
Fraction.js	Librería matemática para trabajar con fracciones.
Polynomial.js	Librería matemática para trabajar con polinomios.
Complex.js	Librería matemática para trabajar con números complejos.
Angles.js	Librería matemática para trabajar con ángulos.

DATE

El objeto 'Date' permite trabajar con fechas (año, mes, día, hora, minutos, segundos y milisegundos). Las fechas toman el valor en milisegundos calculado desde el 1 de Enero 1970.

a) Lun Abr 03 2017 17:25:35 GMT + 0200

b) 1491233135216

A continuación, se muestran diferentes formas de declarar e inicializar una variable de tipo Date.

```
var d = new Date(); // Declaración del objeto  
  
var d = new Date("October 13, 2014 11:13:00"); // Con cadena de texto  
  
var d = new Date("2015-03"); // Año y mes  
var d = new Date("2015"); // Año  
  
// El carácter 'T' separa la fecha y la hora y la Z la hora UTC  
var d = new Date("2015-03-25T12:00:00Z");  
  
//Resultado 'Thu Jun 24 1999 11:33:30 GMT+0200'  
  
var fecha = new Date(99, 5, 24, 11, 33, 30, 0);  
  
// Crea fecha corta formato MM/DD/AAAA  
var fecha = new Date("03/25/2015");  
  
// Crea fecha larga formato MMM DD AAAA  
var fecha = new Date("Mar 25 2015");  
  
//Crea objeto Date y lo muestra por pantalla, valor es la fecha actual.  
fecha = new Date();  
  
document.getElementById("párrafo").innerHTML = fecha;
```

Al establecer una fecha, sin especificar la zona horaria, JavaScript utilizará la zona horaria del navegador.

Algunos países disponen de diferentes zonas horarias dentro del mismo territorio. Para esos casos JavaScript ya tiene previsto unos métodos para conseguir la hora local.

Por lo tanto, cuando trabajemos con fechas, la diferencia entre ponerlo o no, radica en que si no se pone estarás usando tu franja horaria (que es lo más normal), y si lo pones pedirás esa fecha en la hora universal (UTC - Universal Time Coordinated), que es GTM+0 (La del meridiano 0, o meridiano de Greenwich).

Métodos Objeto Date

Los métodos de la clase Date pueden ser del tipo "Get" para obtener un valor del tipo fecha, o "Set" para modificar un valor del tipo fecha. Dicho de otra forma, uno lee y el otro escribe en las variables.

Método	Descripción
getDate()	Obtiene el día de una fecha como un número (1-31)
getDay()	Obtiene el día de la semana como un número (0-6)
getFullYear()	Obtiene el año de una fecha con 4 dígitos (yyyy)
getHours()	Obtiene la hora de una fecha como (0-23)
getMilliseconds()	Obtiene los milisegundos de una fecha como (0-999)
getMinutes()	Obtiene los minutos de una fecha como (0-59)
getMonth()	Obtiene el mes de una fecha como (0-11)
getSeconds()	Obtiene los segundos de una fecha como (0-59)
getTime()	Obtiene hora en milisegundos desde 1 Enero de 1970

```

var d = new Date();

document.getElementById("parrafo").innerHTML = d.getTime();
document.getElementById("parrafo").innerHTML = d.getFullYear();
document.getElementById("parrafo").innerHTML = d.getDay();

```

Método	Descripción
setDate()	Asigna el día del mes como un número (1-31)
setDay()	Asigna el día de la semana como un número (0 'Domingo'-6)
setFullYear()	Asigna el año con 4 dígitos (yyyy)
setHours()	Asigna la hora como (0-23)
setMilliseconds()	Asigna los milisegundos como (0-999)
setMinutes()	Asigna los minutos como (0-59)
setMonth()	Asigna el mes como (0-11)
setSeconds()	Asigna los segundos como (0-59)
setTime()	Asigna hora en milisegundos desde 1 Enero de 1970

```

var d = new Date();    d.setDate(15); // Asigna el dia del mes (1-31)

d.setFullYear(2020); // Asigna el año con 4 dígitos

d.setHours(15);      // Asigna la hora (0 a 23)

d.setMonth(4);       // Asigna el mes del año como numero (0-11)

d.setSeconds(35);    // Asigna los segundos (0-59)

d.setMinutes(17);    // Asigna los minutos (0-59)

d.setDay(5);         // Asigna el dia de la semana (0-6). Cero domingo.

d.setTime(1332403882588); // Añade el número de millis a fecha 1/01/1970

```

Los mismos métodos para obtener datos de fecha existen, pero respetando la zona UTC. En este caso las funciones son exactamente iguales y el valor que devuelven es el mismo.

Método	Descripción
getUTCDate()	Obtiene el día como un número (1-31)
getUTCDay()	Obtiene el día de la semana como un número (0-6)
getUTCFullYear()	Obtiene el año con 4 dígitos (yyyy)
getUTCHours()	Obtiene la hora como (0-23)
getUTCMilliseconds()	Obtiene los milisegundos como (0-999)
getUTCMinutes()	Obtiene los minutos como (0-59)
getUTCMonth()	Obtiene el mes como (0-11)
getUTCSeconds()	Obtiene los segundos como (0-59)
setUTCDate()	Asigna el día como un número (1-31)
setUTCDay()	Asigna el día de la semana como un número (0-6)
setUTCFullYear()	Asigna el año con 4 dígitos (yyyy)
setUTCHours()	Asigna la hora como (0-23)
setUTCMilliseconds()	Asigna los milisegundos como (0-999)
setUTCMinutes()	Asigna los minutos como (0-59)
setUTCMonth()	Asigna el mes como (0-11)
setUTCSeconds()	Asigna los segundos como (0-59)

El método **parse()** convierte una cadena de texto en milisegundos.

```
var milisec = Date.parse("March 21, 2012");  
  
document.getElementById("parrafo").innerHTML = milisec;
```

Existen dos métodos para convertir fechas en cadenas de texto y para obtener otro tipo de información:

- **toString():** Convierte una fecha en cadena de texto expresado en formato inglés americano.

```
var d = new Date();  
  
var n = d.toString();  
  
// Resultado Wed Jun 07 2017 15:30:56 GMT+0200
```

- **toDateString():** Convierte una fecha en cadena de texto expresado en formato corto inglés americano. Solo devuelve la parte de la fecha.

```
var d = new Date();  
  
var n = d.toDateString();  
  
// Resultado Wed Jun 07 2017
```

- **toTimeString():** Convierte una fecha en cadena de texto expresado en formato corto inglés americano. Solo devuelve la parte de la hora.

```
var d = new Date();  
  
var n = d.toTimeString();  
  
// Resultado 15:30:56 GMT+0200
```

- **toUTCString():** Convierte automáticamente una fecha en cadena de texto respetando la hora universal UTC. La hora en UTC es la misma que en GMT. Este método sustituye al que se considera obsoleto 'toGMTToString()'.

```
d = new Date();  
  
document.getElementById("parrafo").innerHTML = d.toUTCString();
```

- **toLocaleString():** Convierte un objeto Date a una cadena utilizando las convenciones de localización. Hay que tener presente que los valores la gran mayoría de veces los datos deben ser exportados y el formato más utilizado es texto.

```
var d = new Date();
var n = d.toLocaleString();
// Resultado 7/6/2018 15:20:35
```

- **now():** Devuelve un número que representa el total de milisegundos transcurridos hasta la fecha actual tomando como referencia el 1 de Enero de 1970. Dicho de otra forma, nos informa de la fecha y momento actual representado en milisegundos.

```
var n = Date.now();
// Resultado de 'n' podría ser 14968413567
```

- **toLocaleDateString() y toLocaleTimeString():** Devuelven la fecha o la hora en formato más corto, ya que el formato de fecha y hora por defecto muestra mucha información. Además, tiene en cuenta las convenciones de localización de zona horaria.

```
var n = d.toLocaleDateString();
// Resultado de 'n' podría ser 12/5/2017

var n = d.toLocaleTimeString();
// Resultado de 'n' podría ser 17:34:33
```

La librería **MomentJS** de Javascript esta específicamente diseñada para trabajar de forma más cómoda con las fechas en Javascript.

TIMERS (Temporizadores)

Dentro del objeto '**window**' hay una serie de "*métodos especiales*" que nos permiten jugar o controlar el tiempo dentro de las aplicaciones. Son métodos muy útiles que otorgan cierta autonomía a las aplicaciones ya que permite repetir secuencias de código cada cierto tiempo. Los 'timers' normalmente se ejecutan una vez ha sido cargada la página a través de evento '*onload*'. También se pueden ejecutar con cualquier otro evento.

Las funciones **setInterval()**, **setTimeout()**, **clearTimeout()**, **clearInterval()** del objeto '**window**' son globales y permiten especificar en qué momento se ejecuta un método y de qué forma se puede parar su ejecución.

- **setInterval()**: Este método global permite ejecutar de forma ininterrumpida una función o una expresión cada cierto tiempo especificado. El primer parámetro del método especifica qué función o código se desea ejecutar, mientras que el segundo parámetro especifica el tiempo (en milisegundos) que debe pasar entre cada una de las llamadas.

El valor devuelto por la función es un identificador o referencia del método que será utilizado para detener el ciclo.

Ejemplo 1: Ejecuta la alerta a través de una función anónima de callback cada 3 segundos.

```
setInterval(function(){  
    alert("Hello");  
}, 3000);
```

Ejemplo 2: En este caso el método ejecuta la función '*procesar()*' cada medio segundo y el identificador del método es guardado en la variable '*reloj*' por si fuese necesario detenerlo. Este tipo de procedimientos utilizan variables globales precisamente para que no se pierdan o desaparezcan si estuvieran dentro de una función al finalizar la ejecución del código.

```
reloj = setInterval(procesar,500);
```

- **setTimeout():** Este método realiza la misma acción que 'setInterval()' con la diferencia que se ejecuta exclusivamente una vez. En determinadas ocasiones en vez de realizar un 'setInterval()' que es un bucle infinito mientras el usuario no lo detenga es preferible llamar varias veces a 'setTimeout()' cuando sea necesario.
- **clearInterval():** Este método detiene la ejecución del método 'setInterval()'. El método para poder ser ejecutado con éxito necesita el identificador que devuelve el método 'setInterval()' cuando es invocado.

Ejemplo 1: Detiene la ejecución en forma de bucle de la función **setInterval()**. Es necesario pasarle el identificador o referencia creada con el método setInterval().

clearInterval(reloj);

- **clearTimeout():** Detiene la ejecución de 'setTimeout()'. La función también necesita del identificador que crea el método 'setTimeout()' para poder detenerlo. Este método es útil cuando se ejecuta un 'setTimeout()' que tiene un tiempo muy extenso.

Ejemplo de un código Javascript que lanza la función **inicio()** cuando la página se haya cargado.

```
<script type="text/javascript">

    // Pone en escucha y al cargar la página y lanza función 'inicio()'
    window.addEventListener('load', inicio, false);

    var reloj;          // Variable global

    function inicio() {

        reloj=setInterval(procesar,50); // Ejecuta de forma repetida
        document.getElementById('boton1').addEventListener('click',
            presionBoton,false);
    }
}
```

```
function procesar() {  
  
    var nro=parseInt(document.getElementById('cronometro').innerHTML);  
  
    if(nro++<51){  
  
        document.getElementById('cronometro').innerHTML=nro;  
  
    }  
  
    else{  
  
        document.getElementById('cronometro').innerHTML=0;  
  
    }  
  
}  
  
function presionBoton() {  
  
    if (document.getElementById('boton1').value='detener') {  
  
        clearInterval(reloj); // Detiene ejecución de setInterval  
  
        document.getElementById('boton1').value='continuar';  
  
    }  
  
    else {  
  
        reloj = setInterval(procesar,50); // Ejecuta de forma repetida  
  
        document.getElementById('boton1').value='detener';  
  
    }  
  
}  
  
</script>
```

THIS

La palabra clave '**this**' tiene un comportamiento diferente al de otros lenguajes de programación. Su valor hace **referencia al propietario de la función** que la está invocando o en su defecto, al objeto donde dicha función es un método. Por tanto, como puede devolver diferentes valores según se encuentre ubicado dentro del código se comentan los más frecuentes:

- En un método de un objeto: Hace referencia al objeto padre que almacena el método.

```
// Resultado devuelto "John Doe".  
  
var person = {  
  
    firstName : "John",  
  
    lastName  : "Doe",  
  
    id         : 5566,  
  
    myFunction : function() {  
  
        return this.firstName + " " + this.lastName; }  
  
};
```

- En una función: (modo normal) En este caso devuelve el objeto 'window'.

```
function myFunction() {  
  
    return this; // Devuelve [object window].  
  
}
```

- En una función: (modo estricto) En este caso devuelve el objeto '*undefined*' porque no permite enlazar en modo estricto con el elemento padre.

```
"use strict";  
  
function myFunction() {  
  
    return this; // Devuelve [undefined].  
  
}
```

- Solo en el código: Hace referencia siempre al objeto 'window'.

```
var x = this; // Devuelve [object window].
```

- En un evento: Hace referencia siempre al objeto que lo invoca.

```
// Devuelve el objeto <button>  
<button onclick="this.style.display='none'"> </button>
```



OBJETOS EN JAVASCRIPT (POO)

La programación orientada a objetos (POO) es un paradigma de la programación dentro de los muchos paradigmas que hay dentro del mundo de la programación. Los **paradigmas**, básicamente, son las diferentes formas o metodologías existentes para solucionar problemas dentro del mundo de la programación y POO es una de ellas.

JavaScript se basa en el paradigma de objetos, es decir, todos los elementos de una web son considerados como objetos por el lenguaje. Dentro de los objetos que podemos encontrar en una web podrán ser clasificados en uno de los dos grupos:

- **Predefinidos por el lenguaje:** Son los objetos que JavaScript ya aporta con el estándar del lenguaje como el objeto de la aplicación ('*window*'), la página web ('*document*'), la pantalla ('*screen*'), las cadenas de texto ('*string*'), las fechas ('*date*') o las colecciones de objetos ('*array*') entre otros.
- **Personalizados:** Son objetos que **diseñamos libremente** en base a unas necesidades particulares de desarrollo, determinando el nombre del objeto, sus propiedades y los métodos tendrá. Dentro de este tipo la forma más fácil de crear un objeto es mediante la utilización de objetos de notación literal (**objetos literales**).
 - **Objetos literales:** Este tipo de objeto está delimitado por llaves {} y en su interior se almacenan una serie de valores con el formato '*propiedad:valor*'.

```
var nombreObjeto = { valor:propiedad, valor:propiedad, };
```
 - **Palabra clave 'new':** Forma de instanciar un objeto que ha sido creado. Es la forma mas frecuente de crear objetos dentro del lenguaje.
 - **Constructor objetos:** (Explicado más adelante).

Los objetos, en esencia, son elementos que almacenan sus **propiedades** (características del objeto) y sus **métodos** (capacidades del objeto, acciones que puede realizar). Por ejemplo, el objeto '*coche*' tiene sus propiedades de objeto como (color, peso, tamaño, etc..) y sus funcionalidades o métodos como pueden ser (arrancar, parar, girar, frenar, acelerar, etc...). Puede haber dos objetos iguales del mismo tipo, pero con propiedades diferentes.



Propiedades (¿Cómo es?)	Métodos (¿Qué se puede hacer?)
Coche.nombre="Fiat"	Coche.arrancar()
Coche.peso=750kg	Coche.girar()
Coche.color=blanco	Coche.retroceder()
Coche.modelo=500	Coche.parar()
Coche.velocidad=110	Coche.acelerar()

Formas de declarar un objeto

- **Modo minificado:** La declaración del objeto 'coche' y sus propiedades en JavaScript utilizando el modo minificado quedaría de la siguiente forma:

```
var coche = { nombre:"Fiat", peso:750, modelo:"500", color:"blanco" };
```

- **Modo normal:** La declaración del objeto (4 propiedades y 1 método en este caso) utilizando un sistema que permite mayor legibilidad y un mantenimiento más sencillo quedaría:

```
var coche = { nombre:"Fiat",
    peso:750,
    modelo:"500",
    color:"blanco",
    pesaje: function() {
        return this.peso * 2; }
};
```

Instanciar un objeto

Para instanciar (declarar) una nueva variable con las propiedades y métodos del objeto que previamente ha sido creado se realiza con la palabra reservada '*new*'. La gran mayoría de objetos nativos en Javascript se declaran de esta forma.

```
var x = new coche(); // Se instancia variable 'x' del tipo objeto coche
```

Acceder propiedades de un objeto

Las propiedades de un objeto se pueden invocar de dos formas diferentes:

1. Forma 1 (Nomenclatura del punto): **coche.nombre** // Más utilizado
2. Forma 2: **coche["nombre"]** // Utilización mucho menos frecuente.

Invoker métodos de un objeto

Para invocar un método de un objeto se realiza mediante la **nomenclatura del punto**:

```
coche.pesaje(); // Se llama a la función 'pesaje()' del objeto coche
```

Añadir propiedad a un objeto

Es posible añadir propiedades que no se encuentran dentro del objeto creado inicialmente. A través de la nomenclatura del punto es posible personalizar con nuevas propiedades los objetos que ya tenemos declarados.

```
coche.combustible = "diesel"; // Nueva propiedad del objeto 'coche'
```

Eliminar propiedad a un objeto

Es posible eliminar propiedades de los objetos siempre utilizando la palabra reservada '**delete**'.

```
delete coche.combustible = "diesel"; // Elimina propiedad del objeto.
```

Añadir método a un objeto

Existe la posibilidad de añadir un método a un objeto después de su creación. En este caso se añade el método '*'peso()'*' y se construye mediante una función anónima. Para invocar al método se debe realizar respetando los paréntesis al realizar la llamada.

```
var coche = { nombre:"Fiat",
              peso:750,
              modelo:"500",
              color:"blanco",
            };
coche.peso = function() { // Añadimos función 'peso()' al objeto
  return this.peso * 2; }
coche.peso(); // Invoca a peso() que ha sido previamente creada.
```

Invocar método externo desde dentro del objeto

En ocasiones desde dentro de la función se pueden invocar funciones que están fuera de él. La función exterior es como cualquier función declarada en la aplicación. En este ejemplo la variable '*'this'*' hace referencia al objeto padre.

```
var coche = { nombre:"Fiat",
              peso:750,
              modelo:"500",
              color:"blanco",
              pesaje: calculoPesaje
            };
function() calculoPesaje() {
  return this.peso * 2; }
```

CREACION DE OBJETOS EN JAVASCRIPT

Los objetos en JavaScript se pueden construir o crear de diferentes formas.

1. **Método literal:** Es la forma más fácil de crear un objeto y es la forma que se ejecuta más rápido, por tanto, es la más recomendable si es posible. Es un sistema que desde la entrada de ES6 con las clases se ha dejado de lado por su poca flexibilidad. La declaración se puede realizar en una línea o en varias líneas siguiendo el mismo formato.

```
var persona = {  
    nombre: "John",  
    apellido: "Doe",  
    edad: 50,  
    colorojos: "azul"  
};  
  
// Escribe en el elemento 'caja1' los valores de las propiedades  
document.getElementById("caja1").innerHTML =  
    persona.nombre + " tiene " + persona.edad + " años.";
```

2. **Palabra clave 'new':** No es el método más recomendable porque "tarda más" en ejecutarse el código a nivel interno, pero es una forma aceptada desde el estándar ES5 para crear objetos y que se utiliza con muchísima frecuencia.

```
// Objeto declarado mediante palabra reservada 'new'.  
  
var persona = new Object();  
  
persona.nombre = "John";  
persona.apellido = "Doe";  
persona.edad = 50;  
persona.colorojos = "blue";
```

// Escribe en el elemento 'demo' los valores de las propiedades.

```
document.getElementById("demo").innerHTML =  
` La ${persona.nombre} tiene ${persona.edad} años.`;
```

La forma 1 y 2 realizan exactamente lo mismo a nivel de código así que no hay necesidad de utilizar la palabra clave **new Object()**. Es preferible por simplicidad, legibilidad y velocidad de ejecución del código la primera forma. Tanto una como otra son formas poco habituales de declarar nuevos objetos en Javascript.

3. **Constructor de objetos:** Este sistema a diferencia de los dos anteriores permite crear varios objetos del mismo tipo. Dicho de otra forma, con los métodos anteriores solo se crea un objeto mientras que con este sistema se pueden crear varios objetos del mismo tipo, pero con diferentes propiedades.

El siguiente código crea el objeto '**persona**' al cual se le pasan unos parámetros para construir el objeto mediante el constructor del objeto. Los valores pasados son lo que se utilizarán para asignarle un valor a cada propiedad del objeto.

```
function persona(nombre1, apellido1, edad1, colorojos1) {  
    this.nombre = nombre1;  
    this.apellido = apellido1;  
    this.edad = edad1;  
    this.colorojos = colorojos1;  
}
```

// Creación de dos objetos idénticos con diferentes propiedades.

```
var padre = new persona("John", "Doe", 50, "azul");  
var madre = new persona("Sally", "Rally", 48, "verde");
```

El siguiente código muestra la creación de un objeto con 3 propiedades que se pasan por parámetro y un método. El método concatena los valores de las propiedades y los muestra mediante un mensaje de alerta cuando se invoca.

```
function CuentaBancaria (nombre, apellidos, dinero) {  
  
    this.nomTitular = nombre;  
  
    this.apeTitular = apellidos;  
  
    this.saldo = dinero;  
  
  
    this.mostrarDatos = function () { // Método dentro de objeto  
        var msg = 'Datos de la cuenta son Nombre: ${this.nomTitular}  
Apellidos: ${this.apeTitular} Saldo: ${this.saldo};  
        alert(msg);  
    }  
}  
  
// Creación de dos objetos del tipo CuentaBancaria con diferentes  
propiedades de creación.  
  
var cliente1 = new CuentaBancaria("John", "Doe", 5000);  
var cliente2 = new CuentaBancaria("Barack", "Obama", 10000);
```

OBJETOS Y PROTOTIPOS EN JAVASCRIPT

La propiedad **prototype** de JavaScript permite **añadir nuevas propiedades y métodos a un objeto existente** desde su base (el prototipo). Este objeto que ya ha sido creado puede ser construido por el usuario o puede ser un objeto de los que vienen predefinidos en JavaScript (por ejemplo, String).

Solo es recomendable modificar los prototipos creados por uno mismo y no los establecidos en los objetos por el estándar en JavaScript. Los elementos añadidos deberán estar lo más próximos posible a los objetos que hacen referencia.

Cuando trabajamos con un objeto podemos añadir propiedades y métodos fácilmente. El problema radica en que añadiendo propiedades o métodos sin la propiedad '**prototype**' solo se añadirá la propiedad o el método a la instancia creada y no a todas las instancias que podamos generar del objeto. Mediante '**prototype**' agregamos los nuevos métodos y propiedades al objeto principal, el padre del cual se generan todas las instancias.

Ejemplo 1:

```
// Declaración de un objeto con 4 propiedades pasadas por parámetro.
```

```
function Persona (nombre1, apellido1, edad1, ojocolor1) {  
    this.nombre = nombre1;  
    this.apellido = apellido1;  
    this.edad = edad1;  
    this.ojocolor = ojocolor1;  
}
```

La palabra reservada **this** se utiliza para referenciar al objeto al que la instrucción pertenece, es decir, se llama a sí mismo.

```
// Creación de la nueva propiedad 'sexo' en el objeto 'persona'. En este caso se incluye en el objeto padre.
```

```
Persona.prototype.sexo = "mujer";
```

```
// Creación de nuevo método, que se añade al objeto padre. Afectaría a todas las instancias que se creasen del objeto.
```

```
Persona.prototype.nombrecompleto = function() {  
    return this.nombre + " " + this.apellido; };
```

Ejemplo 2: Creación de un objeto utilizando un sistema válido, pero algo antiguo según las tendencias actuales de programación en Javascript (ES6). La palabra reservada 'prototype' permite añadir en este caso los nuevos métodos. El código funcionaría perfectamente en todos los navegadores.

```
// Declaración del objeto y asignación de valor a las propiedades. El elemento 'this' en este caso hace referencia a la función 'Pantalla'

function Pantalla(marca, modelo, pulgadas) {

    this.marca = marca;
    this.modelo = modelo;
    this.pulgadas = pulgadas;
}

// Al declarar los métodos es necesario la palabra 'function'.

Pantalla.prototype.encendido = function () {
    -----codigo función-----
};

Pantalla.prototype.volumen = function () {
    -----codigo función-----
};

Pantalla.prototype.info = function () {
    -----codigo función-----
};

// Creación del objeto con el paso de parámetros para su creación.

const tvComedor = new Pantalla('Sony', 'Bravia', 55);

// Llamada a un método del objeto creado.

tvComedor.encendido();
```

Los **objetos y prototipos** en JS no han sido bien aceptados por la comunidad de programadores de otros lenguajes que intentaban desarrollar aplicaciones en este lenguaje porque no había demasiada similitud con los lenguajes más populares (C, Java, Phyton , etc..). Después de ejercer mucha presión JS se ha visto forzado a incluir **las clases** dentro de su estándar desde la versión ES6.

Trabajar con clases es la forma en que Javascript gestiona los diferentes objetos que podemos encontrar dentro de una página web.

Las clases a lo largo del tiempo en Javascript han sufrido diferentes cambios en su nomenclatura y podemos encontrarnos formas de declarar clases de una forma u otra en función entre diferentes código según las diferentes actualizaciones del estándar en esta área.



JSON (Javascript Object Notation)

Los objetos JSON son un **formato de texto ligero** para intercambio de datos en forma de objeto. Leer y escribir JSON es muy sencillo porque es considerado un array asociativo de elementos u objetos.

Permite estructurar un gran volumen de datos de forma simple, óptima y rápida.

Los elementos del JSON están distribuidos por { 'nombre propiedad' : 'valor' }. Si hubiera muchas propiedades, que es lo más habitual, se deberán separar por comas (,).

Los elementos JSON deben ser recorridos a través de la sentencia '**for...in**' ya que es la única que permite leer elementos que se encuentran en modo asociativo. Por tanto, ni la clásica sentencia '**for**' ni la más reciente '**foreach**' pueden leer este tipo de contenido.

Ejemplo 1: Elemento JSON con un único objeto. La clave se puede escribir con el nombre directamente con o sin dobles comillas. En cambio el valor es más estricto, si es un texto irá siempre entre dobles comillas y si es un numero o booleano sin ellas.

```
var pelicula = {  
    titulo: "Superman",  
    year: 2015,  
    pais: "EUA"  
};
```

- **Acceder elemento del JSON:** Se realiza mediante la nomenclatura del punto como en cualquier objeto de Javascript. También es posible acceder al elemento como si fuese la posición de un array.

```
var nombre_pelicula = pelicula.titulo; // Nomenclatura del punto  
var nombre_pelicula = pelicula["titulo"]; // Array asociativo
```

- **Realizar una asignación:** Se puede utilizar cualquiera de las dos opciones disponibles.

```
pelicula.titulo = "Harry Potter";  
pelicula["titulo"] = "Harry Potter";
```

- **Eliminar valor:** Es posible eliminar cualquier valor mediante la palabra '*delete*'. Cualquiera de los dos sistemas es válido.

```
delete pelicula.titulo;
```

```
delete pelicula["titulo"];
```

Ejemplo 2: Elemento JSON con varios objetos. En este caso se deberá recorrer el objeto JSON para poder acceder a cada una de las propiedades de los diferentes elementos.

```
var peliculas = [ {titulo: "Superman", year: 2015, pais: "EUA"},  
                 {titulo: "Batman", year: 2016, pais: "Canada"},  
                 {titulo: "Superlopez", year: 2018, pais: "Spain"},  
                 {titulo: "Hulk", year: 2017, pais: "EUA"}  
];  
  
// Recorre el JSON y en cada iteración se podrá recuperar los  
valores de las propiedades para cada "fila". La variable 'index'  
dentro de un bucle for ... in devuelve la posición.  
  
for ( let index in peliculas) {  
    console.log(peliculas[index].titulo)  
}
```

El **objeto JSON** de Javascript dispone de una serie de métodos que permiten convertir objetos JSON a otros formatos, por ejemplo, un String.

- **stringify():** Este método convierte un objeto JSON a cadena de texto. Esta acción es necesaria porque un objeto puro no se puede tratar como una variable ya que es un contenedor de elementos y devolvería un valor del tipo [object Object]. Además para poderlo enviar al servidor web este debe recibirla en un formato reconocible, y en este caso JSON es el sistema más utilizado junto con XML.

```
var peliculas = [ {titulo: "Superman", year: 2015, pais: "EUA"},  
                  {titulo: "Batman", year: 2016, pais: "Canada"},  
                  {titulo: "Superlopez", year: 2018, pais: "Spain"},  
                  {titulo: "Hulk", year: 2017, pais: "EUA"}  
];  
  
var texto_convertido = JSON.stringify(peliculas);
```

- **parse()**: Convierte un string (texto) que representa un JSON a objeto propiamente JSON preparado para ser utilizado. En ese caso hace la acción inversa. Cuando el objeto es convertido a JSON se puede nuevamente tratar como un objeto y de esta forma acceder a todos los elementos y propiedades del mismo.

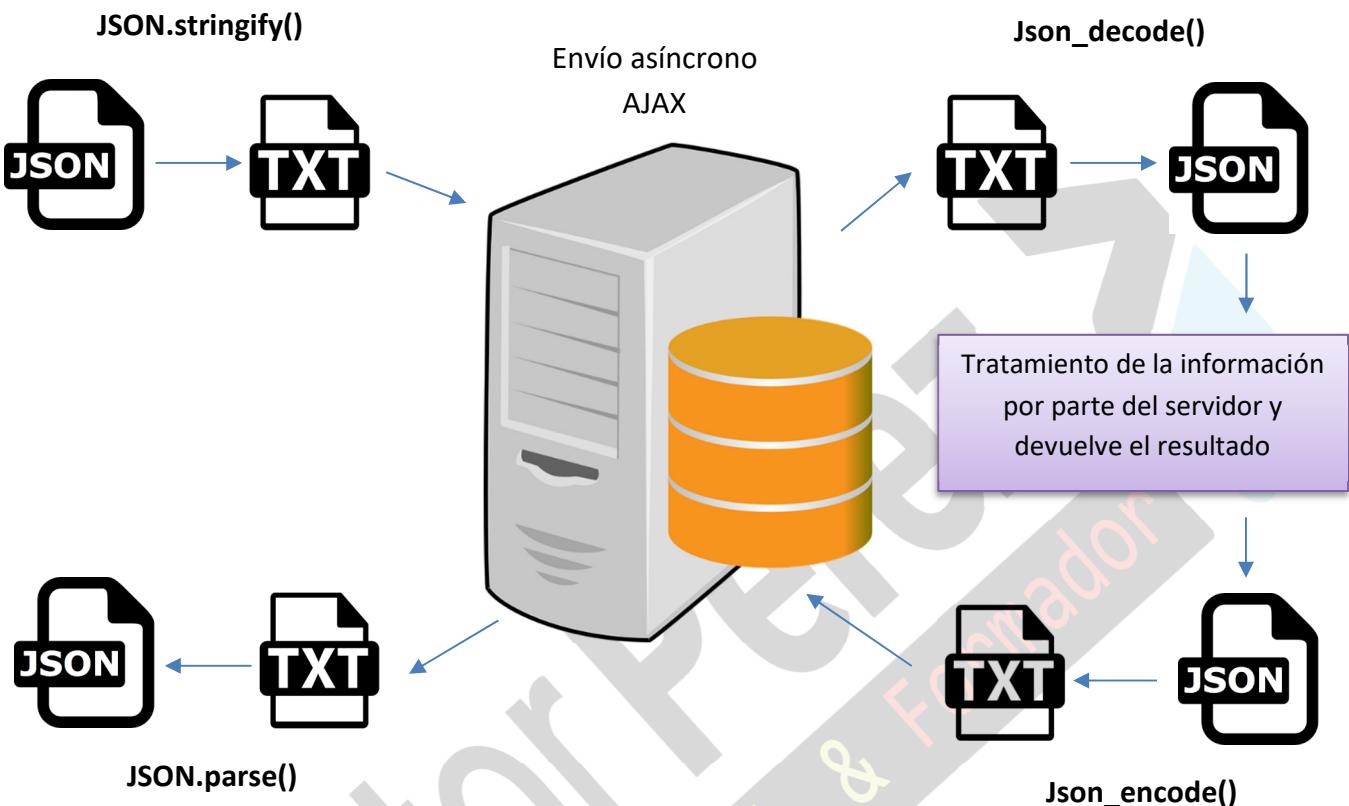
```
// Convierte el String recuperado a objeto JSON con el método  
parse() y lo almacena en una variable mediante localStorage.  
  
var peliculas = JSON.parse(localStorage.getItem("pelis"));
```

El objeto JSON convertido a texto se utiliza frecuentemente para compartir información entre plataformas que utilizan diferentes lenguajes de programación. De esta forma una información convertida a formato JSON en Javascript puede ser leída por una aplicación desarrollada en Python y viceversa. La ventaja de utilizar formatos universales (JSON o XML) facilitan la integración de nuestras aplicaciones con otros sistemas.

Un uso muy habitual es enviar datos como objeto JSON convertido a string al servidor. Este servidor deberá abrir estos datos, tratarlos y, como norma general, enviar una respuesta que nuevamente debe ser desempaquetada para poderla leer. A continuación se muestra este proceso por pasos.

Lado cliente (Javascript)

Lado servidor (PHP)



NOTA: Algunos servidores web aceptan el envío de archivos en formato del tipo JSON sin tener que realizar la conversión de los datos a string y viceversa porque son capaces de reconocer el formato JSON. Por ahora es recomendable realizar siempre esta conversión para no tener que depender de la interpretación que realice el servidor.

FUNDAMENTOS POO EN JAVASCRIPT

Los lenguajes basados en POO como JS o Java utilizan el paradigma (la forma de solucionar problemas) de la POO, aunque hay otros paradigmas muy reconocidos como, por ejemplo:

'programación imperativa': Es la más usada y la más antigua basada en órdenes directas.

'programación dinámica': Trocear problemas para una resolución más fácil de los algoritmos.

'programación dirigida a eventos': La ejecución de los programas van determinados por los sucesos que ocurran en el sistema.

'programación multiparadigma': Es el uso de dos o más paradigmas dentro de un programa.

La POO tiene las siguientes características que son comunes a otros lenguajes de programación:

- **Abstracción**: Significa **ver algo como un todo** sin saber cómo está formado internamente. Por ejemplo, una persona sabe utilizar un Smartphone, pero desconoce el funcionamiento de todos los componentes y programas internos. En programación podemos utilizar las propiedades y métodos de un objeto (o una clase) sin necesidad de conocer su implementación ni como está diseñada o programada internamente.

Para ello, es necesario que todos los objetos del proyecto se encuentren perfectamente documentados especificando que realizan, los parámetros que precisan y si devuelven algún tipo de valor. Realizar POO implica una considerable tarea de informar sobre cada uno de los procedimientos existentes en las diferentes clases.

- **Encapsulamiento**: Significa **ocultar la información** (encapsular). Con esta característica los objetos y clases utilizadas ocultan los datos que almacenan, aunque puedan ser utilizadas. Esto es muy beneficioso para la seguridad de nuestro website ya que de esta forma el código no será accesible para ser “estudiado” y atacado.
- **Herencia**: Un objeto o clase puede heredar las características de otra clase. Esta característica nos permite la reutilización de código para la creación de nuevos objetos con el consiguiente ahorro de tiempo y código. Javascript no soporta la herencia múltiple, es decir, heredar de varios padres a la vez.

- **Poliformismo**: Es la capacidad de un objeto para comportarse de diferentes formas de acuerdo al mensaje recibido. Es un concepto muy relacionado con la herencia. Por ejemplo, desde un Smartphone se puede llamar a Ana o a Pedro, es decir realizando la misma acción ('*marcar_numero()*') el resultado es diferente, por tanto, un objeto tiene múltiples formas de ejecutarse según los parámetros recibidos.
- **Modular**: Posibilidad de trocear o fragmentar el código en distintos ficheros de forma que permita la exportación de clases, funciones y variables. La modularización de nuestro código va directamente relacionado con el encapsulamiento del mismo al importar-exportar.



CLASES EN JAVASCRIPT

La utilización de clases para crear objetos es la evolución natural del lenguaje JS para asimilarse a otros lenguajes de programación (Java, C, Phyton, etc...). JS se aproxima cada vez más a los lenguajes de programación "de toda la vida" y uno de los pasos evolutivos ha sido la utilización de clases. Las clases pretenden hacer una analogía o metáfora de la vida real utilizando un lenguaje de programación. Las clases son "*plantillas*" para poder crear tantos objetos como se necesiten sobre la clase (instancia).

PASO 1:

Todas las clases en Javascript utilizan la palabra reservada '**class**' y obligatoriamente necesitan de su '**constructor**' de objetos, ambos con las llaves de apertura y cierre.

```
class Pantalla {  
    constructor() {  
        . . .  
    }  
  
    var tvSala = new Pantalla(); // Instancia vacía de la clase 'Pantalla'  
    var tvCine = new Pantalla();
```

La declaración de la clase no necesita el símbolo igual (=) como si se tenía que realizar en la declaración de un objeto.

PASO 2:

Se deberán **añadir las propiedades** a la clase '*Pantalla*' para que al crear la instancia la variable construida disponga de las propiedades y métodos necesarios. En este caso se le pasan los valores de las propiedades al constructor y es necesario hacer una asignación para que el objeto exista. La palabra '**constructor**' solo se puede utilizar una vez por clase.

```
class Pantalla {  
    constructor(marca, modelo, pulgadas) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.pulgadas = pulgadas;  
    }  
  
    var tvSala = new Pantalla( 'Sony', 'bravia', 55 );  
  
    var tvCine = new Pantalla(); // Si no hay parámetros para la creación  
    del objeto 'Pantalla' los valores de las propiedades serán 'undefined'.
```

PASO 3:

Se añaden los métodos (funciones) del objeto siempre dentro de las llaves principales. No será necesario ni la palabra 'function' ni 'prototype' para declarar los métodos de la clase como era necesario en versiones antiguas de Javascript. Los métodos llevan los paréntesis de abrir y cerrar función. Estos métodos se consideran nativos del objeto.

----- CLASE PANTALLA -----

```
class Pantalla {  
    constructor(marca, modelo, pulgadas) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.pulgadas = pulgadas;  
    }  
  
    encendido() {  
        console.log(`La pantalla ${this.marca} está OK`);  
    }  
  
    volumen() {  
        console.log(`El volumen se ha modificado`);  
    }  
  
    info() {  
        console.log(`La pantalla ${this.marca} de modelo  
${this.modelo} es de ${this.pulgadas} pulgadas`);  
    }  
}  
  
// Declaración de los dos objetos tipo 'Pantalla'  
  
var tvSala = new Pantalla( 'Sony', 'Bravia', 55 );  
  
var tvCine = new Pantalla(); // Sin valores de inicialización  
  
// Invocación de métodos sobre la instancia creada del objeto.  
tvSala.info(); // Return texto con valores de las propiedades  
tvCine.encendido(); // Return texto sin valores de las propiedades
```

PASO 4:

Para finalizar, es posible **crear nuevas propiedades** a través de las cuales podemos asignar nuevos valores al objeto o recuperar los valores almacenados. Las palabras reservadas '**set**' y '**get**' se utilizan para añadir o recuperar valores de las propiedades. La palabra '**set**' permite asignar valores a nuevas propiedades o a las que han sido creadas con la clase, y la palabra '**get**' captura valores de la instancia creada.

Cuando invocamos a los métodos mediante las palabras reservadas '**get**' y '**set**' no es necesario la introducción de los paréntesis.

```
class Pantalla {  
    constructor(marca, modelo, pulgadas) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.pulgadas = pulgadas;  
    }  
  
    encendido() { console.log(`La pantalla ${this.marca} está OK`); }  
  
    volumen() { console.log(`El volumen se ha modificado`); }  
  
    info() {  
        console.log(`La pantalla ${this.marca} de modelo  
        ${this.modelo} es de ${this.pulgadas} pulgadas`);  
    }  
  
    set peso(value) {this.kgs=value; } // Asigna valor nueva propiedad  
    get peso() { return this.kgs; } // Recupera valor propiedad.  
}  
  
// Declaración de los dos objetos tipo 'Pantalla'  
  
var tvSala = new Pantalla( 'Sony', 'bravia', 55 );  
  
var tvCine = new Pantalla();  
  
// Asignación y recuperación de valores sobre nuevas propiedades.  
  
tvSala.peso = "10 kg";      // Asigna valor a propiedad (set).  
var peso = tvSala.peso;      // Recupera valor de la propiedad (get).
```

CLASES - HERENCIA

La utilización de clases para crear objetos **permite crear objetos a partir de objetos "padre"** del cual heredan todas sus propiedades y métodos, es decir, habrá clases padre y clases hijo. En estos casos el hijo podrá acceder a las propiedades y métodos de la clase padre.

Los que heredan siempre son los hijos respecto a sus padres **y no a la inversa**, es decir, una clase hija podrá acceder a todos los métodos y propiedades de la clase padre, pero la clase padre no podrá acceder a los métodos y propiedades de la clase hija.

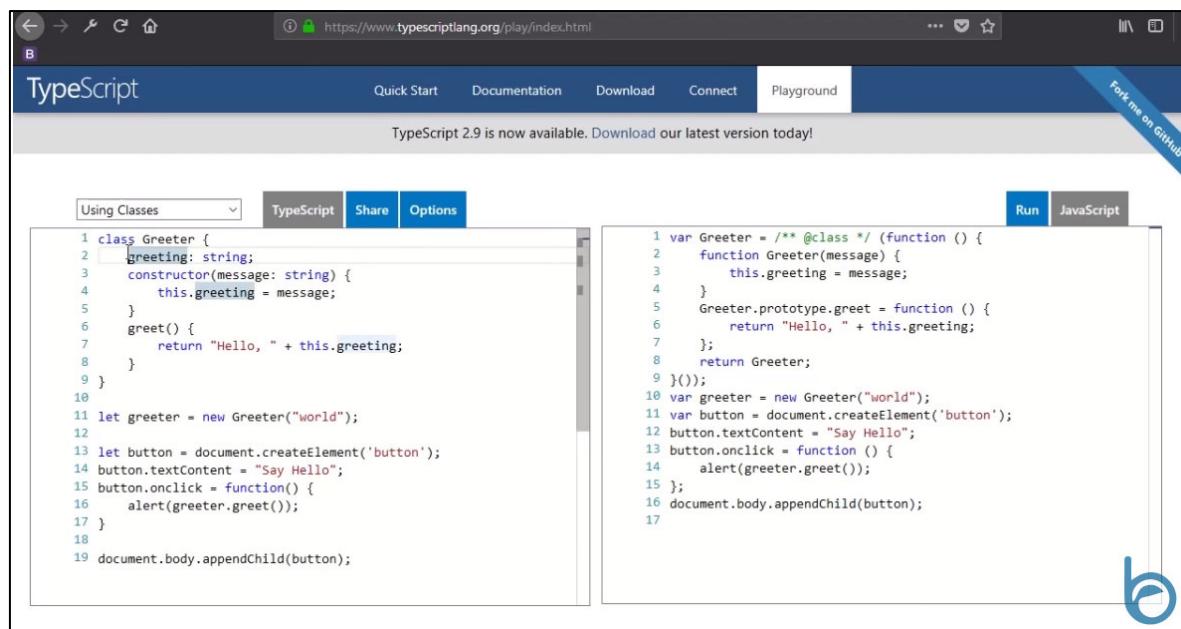
Siguiendo con la clase '*Pantalla*' del apartado anterior, es posible **crear clases más genéricas que puedan englobar a más objetos**. Por ejemplo, la clase '*Pantalla*' podría ser hija de una clase superior denominada '*Productos*' que contiene propiedades y métodos genéricos para todos los productos. En este caso se deberán configurar las dos clases para crear esta relación de herencia y modificar algunas partes del código.

Aspectos a tener en consideración al crear la herencia entre dos clases:

- Se recomienda que el nombre de las clases siempre comience en mayúsculas ('*Producto*').
- Las dos clases deben tener un '**constructor**' con sus argumentos si fuera necesario. El constructor es un elemento imprescindible cuando se trabaja con clases. Si la clase hija no inicializa con ningún valor se puede omitir el constructor ya que se inicializa con el constructor del padre. Es poco frecuente que la clase hija no construya algo para sí misma al ser creada.
- La clase hija '*Pantalla*' utiliza la palabra reservada '**extends**' para determinar que es hija de la clase '*Producto*'. De esta forma la clase '*Pantalla*' podrá tener acceso a todos los métodos y propiedades de la clase padre '*Producto*'.
- La palabra reservada '**super**' en la clase '*Pantalla*' es necesaria para poder **enviar el parámetro al constructor de la clase padre '*Producto*'** y poder inicializar correctamente el objeto. Cuando se trabaja con clases y herencias es necesario crear una serie de acciones en cascada de tal forma que se creará primero el elemento padre y luego el hijo.
- El parámetro '**numSerie**' del constructor en la clase '*Pantalla*' hay que añadirlo porque forma parte de la necesaria inicialización del objeto padre '*Producto*'. Mediante la palabra reservada '**super**' enviamos el parámetro a la clase padre para que se pueda inicializar.
- La palabra reservada '**static**' permite poder acceder al método de la clase **sin necesidad de realizar una instancia** de esta. Podemos decir que se convierte en una clase pública y accesible desde cualquier zona de la aplicación. Cuando se accede al elemento mediante '*get*' o '*set*' no es necesario introducir los paréntesis cuando se invoca. Desde la clase hija también se puede acceder al elemento sin necesidad de ser instanciada.

La utilización de clases es un concepto relativamente nuevo en JS por lo que puede haber problemas de compatibilidad con los diferentes navegadores. Para no tener problemas de compatibilidad es recomendable utilizar '**TypeScript**'.

TypeScript es como JS pero con muchas más funcionalidades y que se encuentra integrado en todos los navegadores web con la finalidad de traducir a lenguaje prototipado las nueva sintaxis de clase en JS.



The screenshot shows the TypeScript playground interface. On the left, there's a code editor with two tabs: 'Using Classes' (selected) and 'TypeScript'. The code in the editor is:

```

1 class Greeter {
2   greeting: string;
3   constructor(message: string) {
4     this.greeting = message;
5   }
6   greet() {
7     return "Hello, " + this.greeting;
8   }
9 }
10 let greeter = new Greeter("world");
11 let button = document.createElement('button');
12 button.textContent = "Say Hello";
13 button.onclick = function() {
14   alert(greeter.greet());
15 }
16 document.body.appendChild(button);

```

On the right, there's another code editor with tabs 'Run' (selected) and 'JavaScript'. The generated JavaScript code is:

```

1 var Greeter = /** @class */ (function () {
2   function Greeter(message) {
3     this.greeting = message;
4   }
5   Greeter.prototype.greet = function () {
6     return "Hello, " + this.greeting;
7   };
8   return Greeter;
9 })();
10 var greeter = new Greeter("world");
11 var button = document.createElement('button');
12 button.textContent = "Say Hello";
13 button.onclick = function () {
14   alert(greeter.greet());
15 };
16 document.body.appendChild(button);

```

CLASE PADRE

```

class Producto {
  constructor(numSerie) {
    this.numSerie = numSerie;
    this.tiempoGarantia = 100;
  }
  static infoTienda() {
    console.log(`Tienda creada el 2020`);
  }
  set garantia(value) {
    this.tiempoGarantia -= value; // Asigna valor propiedad
  }
  get garantia() {
    return this.kgs; // Recupera valor propiedad.
  }
}

```

Los parámetros '*marca*', '*modelo*' y '*pulgadas*' son propiedades de la clase hija, pero '*numSerie*' es un parámetro obligatorio del padre por tanto hay que enviárselo con la palabra '*super*'.

----- CLASE HIJA -----

```
class Pantalla extends Producto {  
    constructor(numSerie, marca, modelo, pulgadas) {  
        super(numSerie);  
        this.marca = marca;  
        this.modelo = modelo;  
        this.pulgadas = pulgadas;  
    }  
  
    encendido() {  
        this.garantia = 1; // Accede al método del padre  
        console.log(`La pantalla ${this.marca} está OK`);  
    }  
  
    volumen() {  
        console.log(`El volumen se ha modificado`);  
    }  
  
    info() { console.log(`La pantalla ${this.marca} de modelo  
        ${this.modelo} es de ${this.pulgadas} pulgadas`);  
    }  
  
    set peso(value) { this.kgs = value; } // Asigna valor propiedad.  
    get peso() { return this.kgs; } // Recupera valor propiedad.  
}  
  
// Se le pasan 4 parámetros porque la clase padre también es necesario  
inicializarla ya que su constructor precisa de un valor.  
  
var tvSala = new Pantalla( '123456', 'Sony', 'bravia', 55 );  
  
Producto.infoTienda; // Accede al elemento, no lleva paréntesis.  
Pantalla.infoTienda; // Acceso al mismo elemento desde la clase hija.
```

CLASES - MODULARIZACION

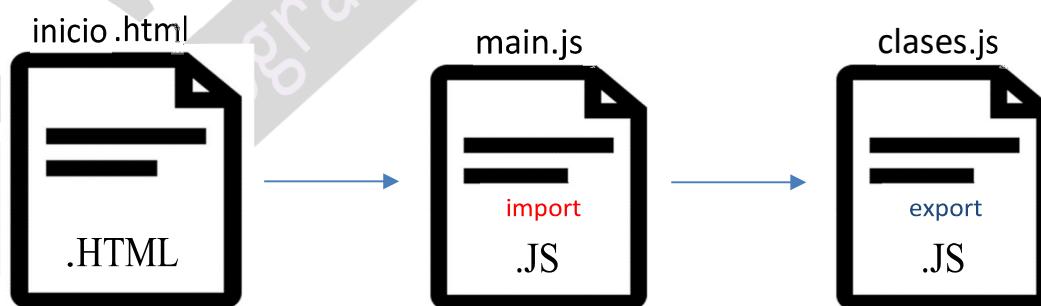
Desde ES6 es posible realizar la modularización del código JS, es decir, **reaprovechar fragmentos de código** como clases, funciones o variables para optimizar el trabajo. Por tanto, las clases seleccionadas se pueden utilizar en otros proyectos simplemente invocándolos.

Los módulos en Javascript son archivos que **nos permiten encapsular código y evitar colisiones de código** (variables y/o funciones repetidas). Los módulos nos permiten tener pequeños archivos que realizan tareas específicas liberándonos de tener grandes archivos de código con infinidad de funciones y clases, es decir, podemos tener el código mejor estructurado para su reutilización.

El "problema" (actualmente) de modularizar es la forma de implementar sintácticamente el código necesario en nuestros archivos. Hasta ahora hay tres patrones (sintaxis) diferentes para realizar esta tarea. Son librerías o sistemas que se han ido generando debido a esta necesidad. Todo y con eso el estándar ES6 ya lo lleva incorporado de forma nativa con sus palabras reservadas '**import**' y '**export**'. Los sistemas más utilizados son los siguientes:

1. CommonJS (Librería orientada del lado del servidor).
2. AMD (Librería orientada del lado cliente pero con carga asíncrona de datos).
3. Módulos ES6 (Es el estándar con su propia sintaxis).

En el siguiente ejemplo el archivo '**inicio.html**' accede mediante un enlace al módulo (archivo) de JS '**main.js**' y este a su vez importa el contenido desde el módulo '**clases.js**'. El contenido del módulo '**clases.js**' (variables, funciones o clases) debe estar configurado mediante la palabra reservada '**export**' para que pueda ser utilizado por el módulo '**main.js**'.



Es imprescindible que al enlazar el archivo .html ('**inicio.html**') con el primer archivo .js ('**main.js**') se especifique el atributo **type** con el valor '**module**'. De esta manera se informa que el archivo .js es un módulo que extraerá la información de otros archivos .js.

```
<script src="index.js" type="module" </script>
```

Este sistema facilita una de las principales características de la POO, el **encapsulamiento de los datos**, es decir, que **el código no sea accesible ni visible**.

Es posible importar desde cualquier módulo (archivo) una variable con su contenido almacenado, una función, o una clase con todos sus métodos y propiedades, siempre y cuando se utilice alguna de las tres posibles sintaxis de importación. En nuestro caso cualquier explicación se realizará tomando como referencia el estándar de Javascript (Módulos ES6).

Desde **un módulo se puede invocar a un archivo** para que importe cualquier elemento, y desde este archivo, a su vez, se puede invocar a otros archivos para que importen sus contenidos produciendo un efecto de cascada.

En este contexto de trabajo (cascada de archivos) es importante resaltar que los archivos que importen funciones, variables o clases no podrán contener nombres repetidos entre ellos ya que podría provocar problemas en tiempo de ejecución y que el intérprete del navegador no sepa como actuar y muestre un error.

Para hacerlo sencillo primero se explicará la importación de una o varias funciones y finalmente se realizará con una clase. El proceso es muy similar en ambos casos.

Importacion de funciones y variables

Archivo 'main.js': (IMPORT). Archivo que solicita las funciones y propiedades al módulo 'clases.js'. Cabe destacar que no hay ninguna diferencia sintáctica al llamar a los métodos o funciones por lo que podría dar a confusión. Se escribe el nombre de la función/es y variables entre llaves y además la ruta relativa donde se encuentra, siempre empezando por './'. En este caso se importan dos funciones y una variable constante.

```
import { saludo, despedida, edad } from './clases.js';

console.log(edad); // Se puede ver valor después de importar
```

Archivo 'clases.js': (EXPORT) Se especifica que función o funciones podrán ser exportadas. En este caso son dos funciones ('saludo', 'despedida') y una variable constante ('edad').

```
export function saludo(nombre) { // Exporta función
    console.log(`Hola mi nombre es ${nombre}`);
}

export function despedida(nombre) { // Exporta función
    console.log(`Adiós, me llamo ${nombre}`);
}

export const edad = 30; // Exporta propiedad
```

Siguiendo con el ejemplo expuesto hay algunas **sintaxis asociadas con la modularización** que facilitan el trabajo con las mismas y que hay que tener presente como por ejemplo:

- **Asignación de un alias:** Los valores importados se pueden renombrar con un alias para facilitar el trabajo si tuvieran nombres largos y complejos. La sintaxis es 'nombre as alias'.

```
import { saludo, despedida, edad as numero} from './clases.js';
```

- **Función por defecto:** La función que contiene el texto '**default**' carga la función en el módulo que lo importa por defecto. NO tiene porque tener el mismo nombre que en el archivo que exporta. Las funciones por defecto permiten utilizar el nombre a modo de alias.

```
// Función que se exportara por defecto
export default function saludo(nombre) { // Exporta función
    console.log(`Hola mi nombre es ${nombre}`);}
```

```
// Forma de invocar la función por defecto
import saludar from './clases.js';
```

- **Importar todos los datos de un módulo:** La importación se realiza sobre el módulo 'main.js'. Se utiliza el asterisco para importarlo todo y se le asigna un alias (Mensajes) además de la ruta. Para acceder a un método se efectúa mediante la nomenclatura del punto.

```
// Importar todos los elementos de un módulo
import * as Mensajes from './clases.js';
```

```
// Accede a los dos métodos pasándoles un parámetro.
console.log( Mensajes.saludo(nombre) );
console.log( Mensajes.despedida(nombre) );
```

```
// Accede a las propiedades
console.log( Mensajes.edad );
```

Importación de clases

Se puede realizar la misma tarea de exportación e importación sobre las clases que se hayan creado dentro de un módulo y la una acción es muy similar. La exportación se puede realizar de dos formas diferentes y la importación de una forma. La sintaxis se muestra a continuación:

1. **Exportar código:** Determinar que código será modularizado (archivo origen – 'clases.js'). Esta acción se puede realizar de dos formas siempre con la utilización de la palabra reservada 'export'.
 - a. (*Forma 1*). Añadiendo la palabra reservada 'export' delante de la clase. Puede realizarse también con clases heredadas. Con este sistema se debe escribir la palabra 'export' en cada elemento que se desea exportar

```
export class Producto { ... }  
export class Pantalla extends Producto { ... }
```
 - b. (*Forma 2*). Agrupando con llaves las clases exportadas en una nueva línea de código. Las clases con herencia se especificará el nombre de la clase padre y clase hija. Esta forma es más rápida ya que en una sola línea se especifica todos los elementos exportados de un determinado archivo.

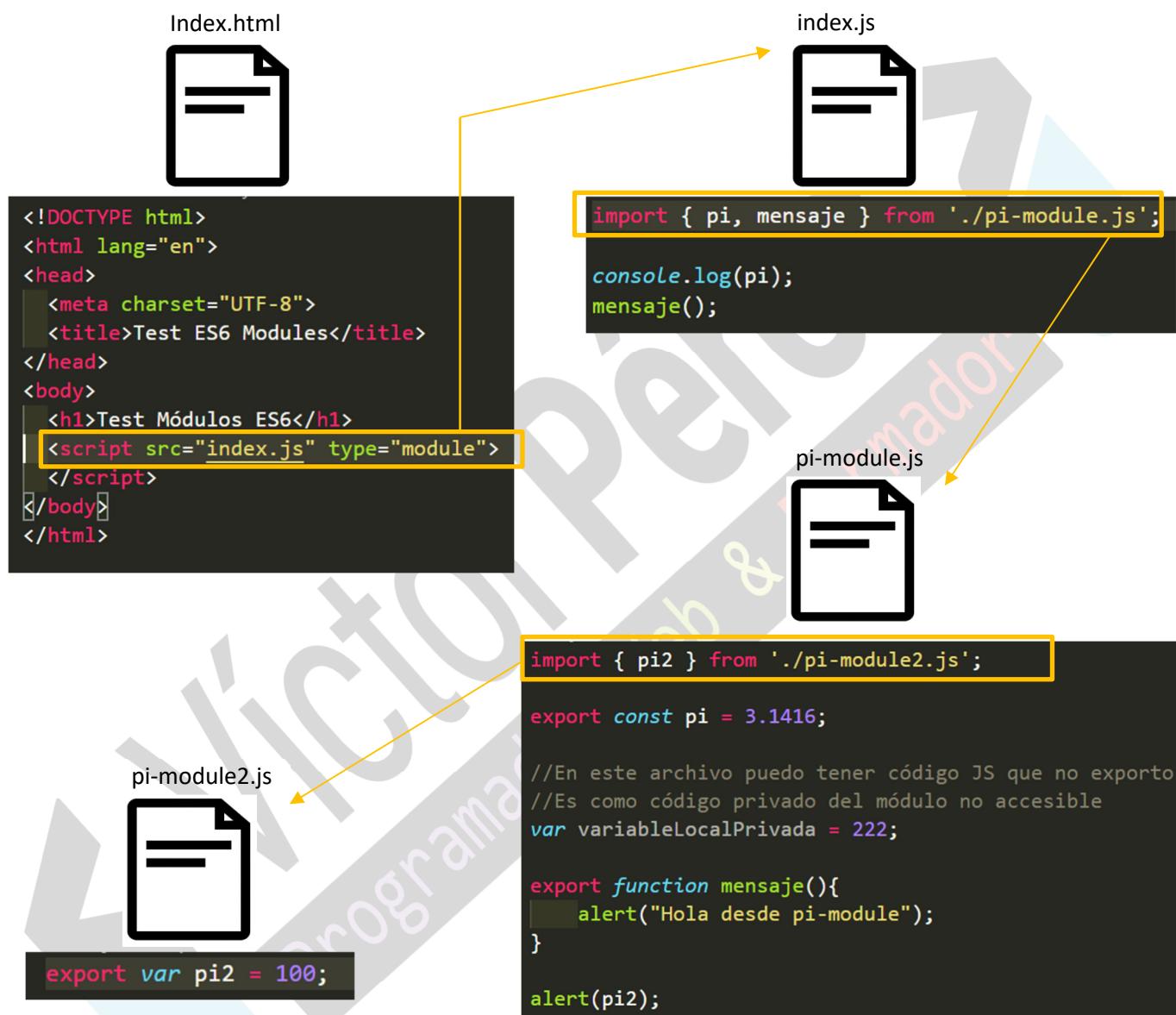
```
class Producto { ... }  
class Pantalla extends Producto { ... }  
  
// Entre llaves todas las clases que se exportarán.  
export { Producto, Pantalla }
```

2. **Importar código:** Especificar en el archivo de destino dónde se encuentra el código que se desea recuperar para su reutilización. Este código puede estar en un único archivo o en varios archivos en el servidor. La sintaxis en la importación sigue los mismos criterios que en los ejemplos anteriores.

```
// Importa las clases 'Producto' y 'Pantalla'  
import { Producto, Pantalla } from './archivo.js';
```

Ejemplo modularización a varios niveles

En este ejemplo se muestra de forma muy sencilla el código de varios archivos que solicitan (importan) o ofrecen (exportan) sus diferentes elementos. La ordenación de los archivos y como se exponen en el siguiente gráfico es la misma que si tuviésemos que programar cada uno de los diferentes documentos.



Explicación del código

- El archivo 'index.html' tiene un enlace al módulo 'index.js'. Se debe especificar que el archivo enlazado contiene la propiedad 'type' con el valor 'module'.
- El módulo 'index.js' importa una propiedad ('pi') y un método ('mensaje') del archivo 'pi-module.js'. La propiedad se mostrará por consola y el método se invocará del archivo donde se encuentra declarado ('pi-module') para ser ejecutado.

- El archivo '**pi-module.js**' importa del archivo '**pi-module2.js**' una variable ('*pi2*') que será invocada desde el archivo '**pi-module.js**'. El valor importado se mostrará a través de un alert().
- La variable ('*pi2*') del archivo '**pi-module2.js**' se caracteriza porque **NO SERÁ ACCESIBLE** desde el archivo '**index.js**'. Para poder utilizar la variable en este archivo será necesario copiar el valor importado ('*p2*') en una nueva variable dentro de '**pi-module.js**' y a su vez se exportará al archivo '**index.js**' que también lo importará.

La importación de archivos **no está implementada en todos los navegadores de forma nativa**. ¿Qué sucede?

En realidad, es que la página html importaría un fichero JavaScript, que a su vez importaría a otros ficheros JavaScript resolviendo todas sus dependencias e importando para ello más ficheros JavaScript. La resolución en cascada podría traducirse en miles de llamadas al servidor para traerse pequeños módulos JavaScript.

Y aquí es donde viene uno de los cambios que han revolucionado el mundo front, y es que antes para ejecutar JavaScript te valía un navegador, y ahora se requiere un **proceso de "compilación"**, que normalmente realiza un TaskRunner (herramienta que automatiza tareas) tipo Grunt, Gulp, Parcel o Webpack., que **empaquea todo nuestro código diseminado por módulos en un único fichero** (bundle) entendible por el navegador, que a veces ha sufrido procesos de minificación, de validación, transpilación a ES5 o varias cosas más.

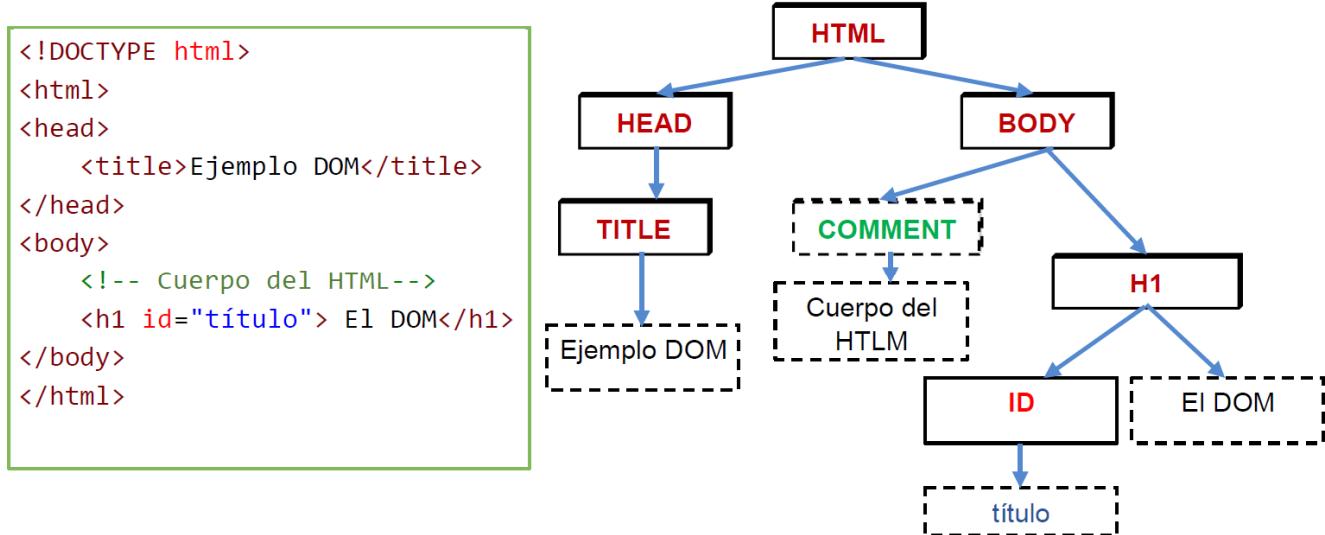
DOM (Document Object Model)

DOM significa **Document Object Model** y es la estructura jerárquica que genera el navegador de todos los elementos (nodos) de una página. Con el árbol DOM JavaScript puede acceder, modificar, añadir elementos o atributos, reaccionar a eventos, crear nuevos eventos, o eliminar todos los elementos HTML de una web de forma dinámica. El DOM (objeto Document) está dentro del BOM (objeto Window que es el de mayor nivel dentro de la estructura jerárquica en una web).

En una página web los nodos que la conforman pueden ser de 5 tipos:

1. **Document**: Es el nodo raíz que contiene todos los nodos en su interior. Ej: html.
2. **Nodos tipo Comment**: Representan los comentarios del HTML. Ej: <!-- Cuerpo HTML -->
3. **Nodos tipo Element**: Representan las marcas o etiquetas que conforman el HTML. Estos son los nodos con los que se suele trabajar principalmente. Ej: head, title, body, p, span, div, a, h1, etc..
4. **Nodo tipo Texto**: Representan los textos contenidos en las marcas o etiquetas. Ej: Hola, adiós.
5. **Nodo tipo Attribute**: Hace referencia a los atributos de las marcas o etiquetas. Ej: id, name, class

El siguiente ejemplo muestra desglosada una estructura jerárquica del DOM de una web.



Los nodos de tipo etiqueta pueden tener hermanos, hijos y padre. Los nodos de tipo texto no pueden tener hijos ni hermanos. Un nodo solo puede tener un único parente.

El DOM nos ofrece una API (conjunto de métodos y atributos) para poder interaccionar con los nodos. Si el DOM sufre algún cambio este se verá reflejado en la página. El DOM tiene su nodo raíz que hace referencia al elemento "document" (HTML). Desde este nodo principal se pueden acceder al resto de nodos y sus respectivos hijos.

Si accedemos a un nodo obtenemos un objeto que representa una referencia al elemento del DOM. Si modificamos las propiedades de este objeto se modificará el árbol DOM. El objeto "*document*" contiene un conjunto de propiedades que hacen referencia al "body", "head", todos los links, imágenes y forms.

- **document.body**: Retorna un objeto que tendrá una referencia al objeto BODY.
- **document.head**: Retorna un objeto que tendrá una referencia al objeto HEAD.
- **document.links**: Retorna un array que contendrá por cada uno de los links de la página web un objeto representativo.
- **document.links.length**: Devuelve un valor que representa el número de links que tiene la web.
- **document.images**: Retorna un array que contendrá por cada una de las imágenes de la página web un objeto representativo.
- **document.images.length**: Devuelve un valor que representa el número de imágenes que tiene la web.
- **document.forms**: Retorna un array que contendrá por cada uno de los formularios de la página web un objeto representativo.
- **document.title**: Devuelve el texto que se encuentre entre las etiquetas <title> de la página.
- **document.links.length**: Devuelve un valor con el número de links que contiene la página.

Esta forma de acceso al DOM es muy genérica y por regla general no soluciona los problemas cuando queremos acceder a un nodo específico. Para ello, el objeto "*document*" nos ofrece una serie de métodos que nos permiten acceder a zonas o elementos específicos facilitando el trabajo.

Los métodos del DOM determinan que acciones se pueden realizar sobre los elementos HTML. La forma más común para acceder a los elementos HTML es mediante el método **getElementById()** haciendo siempre referencia sobre el identificador de la etiqueta.

```
<p id="titulo"> Pagina personal </p>

<script>
    // Carga el elemento en una variable accediendo por su id 'titulo'
    var x = document.getElementById("titulo");
</script>
```

Las propiedades del DOM determinan los valores que se pueden modificar. Una propiedad es un valor que se puede **obtener o establecer** en un elemento HTML. Mediante la propiedad '**innerHTML**' se pueden **modificar de forma dinámica los valores de las etiquetas** y sus elementos.

```
<p id="titulo"> Página personal </p>
<script>
    // Cambia el valor del elemento id 'titulo' con innerHTML
    document.getElementById("titulo").innerHTML="Presentacion";
</script>
```

En HTML la distribución de los elementos del DOM es mediante objetos. Cualquier elemento es accesible y modificable. El objeto que representa la ventana del navegador es **Window**, mientras que el objeto que representa la página web es **Document**.

Para acceder a cualquier elemento del documento las formas más utilizadas son:

Metodo	Descripción
document.getElementById(<i>id</i>)	Busca el elemento <u>a partir del id</u>
document.getElementsByTagName(<i>name</i>)	Busca elemento <u>a partir del nombre de la etiqueta</u> .
document.getElementsByClassName(<i>name</i>)	Busca un elemento <u>a partir del nombre de la clase</u> .

Ejemplos:

```
// Captura el objeto que contiene el id 'boton'.
var boton = document.getElementById("box1");
boton.innerHTML = "Introduciendo texto";

// Captura todos los elementos de la web que son una etiqueta <p>. Todos los resultados se guardarán en una variable de tipo array. Posiblemente será necesario recorrer el array para acceder a cada uno de los elementos.
var todosLosParrafos = document.getElementsByTagName('p');

// Captura todos los elementos con la clase 'principal' y son almacenados en un array. El atributo 'textcontent' retorna el valor del elemento del array a partir de la posición establecida.
var paPorClase = document.getElementsByClassName('principal')[0].textContent;
```

Encontrar elementos por Id

Se realiza con el método **getElementById()**. Es el método más extendido para localizar objetos dentro del documento (web). Las características de este método son:

- El objeto debe tener un campo 'id' para poder hacer referencia a él y poder capturarlo.
- El resultado devuelto es un objeto y este se puede almacenar en una variable.
- El atributo 'id' de un elemento debería ser único en toda la página y ningún elemento más debería tener ese mismo nombre para identificarlo.
- Si hubiera dos elementos con el mismo 'id' el método **getElementById()** solo devuelve el primer elemento con el identificador especificado que encontrase.
- Si en la página no hay ningún elemento con el 'id' especificado el método devuelve 'null'.

```
// La etiqueta <p> tiene un identificador único en toda la página.  
<p id="titulo"> Pagina personal </p>  
<script>  
    // Almacena el objeto con identificador 'titulo' dentro de la  
    // variable 'x' para ser posteriormente tratada.  
    var x = document.getElementById("titulo");  
</script>
```

Encontrar elementos por nombre etiqueta

Las páginas web se construyen mediante etiquetas y se puede acceder a ellas mediante el método **getElementsByName()**. Las características de este método son:

- El resultado devuelto es un array con todos los objetos encontrados de la etiqueta especificada. Los valores en el array siempre se almacenan ordenados según se van localizando en la página, es decir, de arriba hacia abajo.
- El array de objetos generado comienza en la posición [0]. Se debe tener presente este dato a la hora de acceder a los elementos de la página.
- La referencia a la etiqueta html se realiza utilizando dobles comillas y no con los símbolos mayor (>) y menor (<). Este es un error muy habitual al hacer referencia a la etiqueta.
- Si no hubiera elementos con la etiqueta especificada el método retornaría 'NodeList'. Si se añadiese un nuevo elemento de las mismas características al DOM, la lista de nodos (el array) se actualizaría de forma automática.

Ejemplo 1: El siguiente código carga todas las etiquetas `<p>` en un array.

```
//Carga todas las etiquetas <p> en la variable x en forma de array
var x = document.getElementsByTagName("p");
```

Ejemplo 2: El siguiente código tiene dos cajas con dos párrafos en su interior cada una.

Mediante el método `getElementById()` se captura el elemento "caja1" en la variable 'x', y posteriormente con el método `getElementsByTagName()` se capturan en un array todos los elementos hijos, es decir, los párrafos que contiene la "caja1".

```
// El código HTML dispone de dos cajas con dos párrafos. Las
// etiquetas <p> son hijas de las etiquetas <div>
<div id="caja1">
    <p> Párrafo 1 </p>
    <p> Párrafo 2 </p>
</div>
<div id="caja2">
    <p> Párrafo 3 </p>
    <p> Párrafo 4 </p>
</div>
<script>
    // Carga el elemento (objeto) con identificador "caja1" y lo
    // almacena en la variable 'x'.
    var x = document.getElementById("caja1");

    // Carga todas las etiquetas hijas <p> del objeto <div> con
    // identificador "caja1" en la variable 'y'. En este momento la
    // variable 'y' se convierte en un array que contiene dos
    // elementos (las 2 etiquetas <p> hijas).
    var y = x.getElementsByTagName("p");

    // Escribe en la página el texto 'párrafo 2', que corresponde
    // al segundo elemento hijo de la caja con identificador "caja1"
    // y que está almacenada en la 2º posición del array.
    document.write('El segundo párrafo es:' + y[1]);
</script>
```

Encontrar elementos por nombre de clase

Este sistema permite localizar todos los elementos a partir del nombre de clase asignado a la etiqueta. El método es **getElementsByClassName()**. Las características de este método son:

- El elemento devuelto es un array con todos los objetos que tienen la clase especificada.
- El array comienza en la posición [0] y almacena los valores por orden según los va localizando al leer la página, es decir, de arriba hacia abajo.
- Si no hubiera elementos con la clase especificada el método retornaría '*NodeList*' sin elementos en su interior. Si se añadiese un nuevo elemento de las mismas características al DOM, la lista de nodos (el array) se actualizaría de forma automática.

```
<p class="p1"> Párrafo 1 con clase P1 </p>
<p class="p1"> Párrafo 2 con clase P1 </p>
<p class="p2"> Párrafo 3 con clase P2 </p>
<p class="p2"> Párrafo 4 con clase P2 </p>
<p class="p2"> Párrafo 4 con clase P2 </p>

<script>

    // Carga en la variable 'x' (que es un array) todas las
    // etiquetas que tienen asignada la clase "p2".
    var x = document.getElementsByClassName("p2");

    // Escribe en la página el texto 'parrafo 4' que corresponde
    // a la última posición del array.
    document.write(`El segundo valor p2 es: ' ${x[1]} `);
    document.write(`El tercer valor p2 es: ' ${x[2]} `);

</script>
```

Encontrar elementos por selectores CSS

El método **querySelectorAll()** permite acceder a todos los elementos HTML que coinciden con un **selector CSS** especificado dentro de un array. El método **querySelector()** funciona exactamente igual con la diferencia que **solo devolverá el primer elemento que cumpla con el criterio establecido**. En este segundo caso NO hay ningún array que almacene una colección de valores ya que solo retorna un valor. Si en la búsqueda no encuentra nada el valor devuelto será '*null*'. Si se modifica el DOM este **array no se actualiza**.

- El resultado se almacena en un array que comienza en la posición [0].
- Si no encontrase elementos con los criterios especificados devolvería un '*Nodelist*' vacío.
- La forma de llamar a los selectores es igual que al construir un archivo en CSS pero en este caso utilizando código JS. La sintaxis utilizada es a través de selectores CSS.
 - Si es una etiqueta html con el nombre. Ej: `querySelectorAll("div")`;
 - Si es un identificador con la almohadilla. Ej: `querySelectorAll("#caja")`;
 - Si es una clase con el punto. Ej: `querySelectorAll(".caja")`;
 - Si es una combinación de selectores. Ej: `querySelectorAll("div .caja")`;
 - Combinación selectores y pseudoclases: Ej: `querySelectorAll(".box1 > div:nth-child(2)")`;
- El método se puede invocar sobre el 'document' o sobre alguna de las etiquetas que contiene. El resultado obtenido será siempre sobre los elementos contenidos en este.

```
<p class="p1"> Párrafo 1 </p>
<p class="p1"> Párrafo 2 </p>
<p class="p2"> Párrafo 3 </p>
<p class="p2"> Párrafo 4 </p>

<script>
    /* Carga todos los elementos <p> del documento que tengan
     aplicados una clase o selector de estilo CSS 'p2' */
    var x = document.querySelectorAll(".p2");

    // Escribe en la página el texto 'parrafo 4' que corresponde
    // al segundo elemento del array que almacena las etiquetas <p>
    // con la clase 'p2'.
    document.write('El segundo valor p2 es:' + y[1]);
</script>
```

Encontrar elementos HTML en colecciones de objetos

Este sistema permite cargar en un array diferentes objetos que pertenecen a una colección de elementos. Por ejemplo, todos los elementos que pertenecen a un formulario se pueden considerar una colección de elementos.

```
// Formulario con tres campos (dos cuadros de texto y un botón)

<form id="frm1" action="/action_page.php">
    Nombre: <input type="text" name="fname" value="Pepe"><br>
    Apellido: <input type="text" name="lname" value="Lopez"><br>
    <input type="submit" value="Enviar">
</form>

// Al pulsar botón 'Enviar' lanza la función 'cargarFormulario'.
<button onclick="cargarFormulario()"> Cargar valores </button>

</script>

function cargarFormulario() {
    // Carga el objeto 'frm1' en la variable x.
    var x = document.forms["frm1"];
    var text = "";
    var i;

    // Recorre el array para extraer los diferentes valores.
    for (i = 0; i < x.length; i++) {
        // Con 'elements' recorre todo el array con la colección
        // Text concatena los valores del array
        text += x.elements[i].value + "<br>";
    }

    // Muestra el valor concatenado por pantalla.
    document.write('Los tres valores son:' + text);
}

</script>
```

Cada vez que obtenemos una referencia a un nodo podemos interactuar con sus propiedades para modificar sus estilos.

Cualquier objeto que haga referencia a un nodo tendrá como mínimo una propiedad que se corresponda con todos los atributos naturales que puede tener el nodo como elemento HTML. Es decir, si accedemos a un nodo “A”, podremos acceder y modificar su atributo “href”, si accedemos a un nodo “INPUT”, podremos acceder y modificar su atributo “value”.

Propiedad	Modificable	Descripción
<i>nodename</i>	NO	Si el nodo es un elemento retorna el nombre de la marca. Si el nodo es un atributo retorna el nombre del atributo. Para otros casos retorna distintos nombres según el tipo.
<i>nodeType</i>	NO	Si el nodo es un elemento retorna un 1 Si el nodo es un atributo retorna un 2 Si el nodo es un texto retorna un 3 Si el nodo es un comentario retorna un 8
<i>tagName</i>		Solo disponible para nodos de tipo element, retorna el nombre de la marca.
<i>innerHTML</i>	SI	Retorna el contenido del nodo indicado (sin mostrar los <>) y todos sus descendientes excepto los ocultos, <script> y <style>. Si se modifica, se sustituirá todo el contenido del nodo por el nuevo valor indicado sin interpretarlo como contenido HTML.
<i>textContent</i>	SI	Retorna el contenido del nodo indicado (sin mostrar los <>) y todos sus descendientes .Si se modifica, se sustituirá todo el contenido del nodo por el nuevo valor indicado sin interpretarlo como contenido HTML.
<i>Id</i>	SI	Establece o retorna el 'id' que tenga el nodo.
<i>classList</i>		Establece o retorna la lista de clases que contiene el nodo tipo element.
<i>className</i>	SI	Establece o retorna string con el contenido del atributo 'class'.
<i>title</i>	SI	Establece o retorna el nodo de un tipo element.
<i>style</i>	SI	Establece o retorna las propiedades CSS aplicadas a un nodo de tipo 'element'. No permite modificar la hoja de estilos sino que las modificaciones se realizan sobre el atributo 'style' de la marca.

Ejemplo para mostrar la información de un nodo

El siguiente código muestra las tres propiedades más importantes de un nodo (`nodeName`, `nodeValue`, y `nodeType`) extrayendo la información de un mismo elemento.

```
// Caja con hijo y el texto “Elemento caja”
<div id="myDIV"> Elemento caja.</div>

// Botón que lanza la función que carga valores
<button onclick="myFunction()">Pulsar</button>

// Párrafo que contendrá el resultado final.
<p id="result"> </p>
```

Elemento caja.

Pulsar

El nodeName: #text
The nodeValue: Elemento caja.
The nodeType: 3



```
<script>
    function myFunction() {
        // Carga el primer hijo del div en la variable, es el texto.
        var x = document.getElementById("myDIV").firstChild;

        // Se crea la variable y se inicializa sin contenido.
        var txt = "";

        // Carga el nombre del nodo, es tipo #text porque es el texto del div.
        txt += "El nodeName: " + x.nodeName + "<br>";

        // Carga el valor que contiene el hijo, es el texto.
        txt += "El nodeValue: " + x.nodeValue + "<br>";

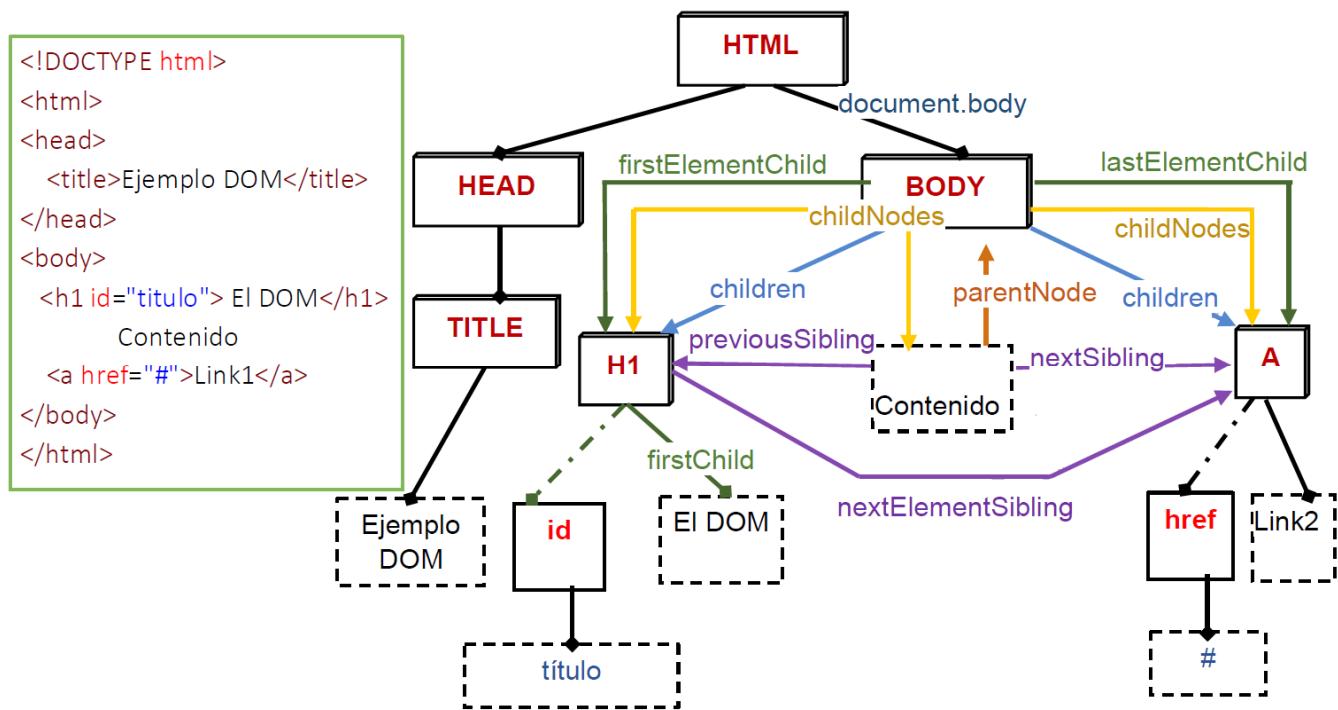
        // Es un número que determina el tipo de nodo
        txt += "El nodeType: " + x.nodeType;

        // Escribe el resultado dentro de la etiqueta <p>
        // Carga el valor mediante innerHTML
        document.getElementById("result").innerHTML = txt;
    }
</script>
```

DOM en HTML y CSS

A partir de la referencia de un nodo es posible acceder a los diferentes elementos del DOM y a los nodos que tengan un parentesco directo, es decir, padre, hermanos e hijos.

El siguiente esquema muestra las principales propiedades que tiene un nodo para recorrer el DOM.



Análisis del gráfico

La diferencia entre '**firstChild**' y '**firstElementChild**' reside en que con la segunda forma estamos queriendo acceder a un elemento que sea del tipo 'element' (etiqueta) mientras que el primero devuelve el primer hijo que no tiene por qué ser una etiqueta. Esto es muy importante porque si nos fijamos en el código HTML estrictamente, el primer hijo que tiene Body es el salto de línea que hay entre las marcas `<body>` y `<h1>`, y en tal caso el **firstChild** de **body** será ese salto de línea.

La propiedad '**childNodes**' retorna un array con todos los hijos (texto y etiquetas), pero en el caso de '**children**' retorna un array solo con los hijos que sean 'element' (etiqueta).

Es importante saber que los atributos no son considerados hijos de un nodo, sino que todos los nodos tipo 'element' (etiqueta) tienen un atributo 'atributes' que contienen un mapa con todos los atributos.

La propiedad '**lastElementChild**' retorna el último hijo de tipo 'element' (etiqueta). Se toma como referencia el padre.

La propiedad '**nextSibling**' y '**previousSibling**' retornan una referencia del siguiente hermano y del anterior respectivamente.

La propiedad '**nextElementSibling**' retorna su siguiente hermano que sea del tipo 'element' (etiqueta). La referencia de base es el hermano y no el padre.

La propiedad '**parentNode**' retorna una referencia a su padre. En este caso se toma como referencia un elemento que tenga un elemento padre.

Si se desea introducir delante de un nodo específico está el método **insertBefore()** donde se especifican dos parámetros, primero el nodo que se desea insertar , y segundo delante de que hermano se posicionará. Hay que tener en cuenta que se toma como referencia el nodo padre y no el hermano.

```
nodopadre.insertBefore(nuevonodo, hijo);
```

Desde el DOM también es posible **modificar los estilos de las propiedades CSS**. Para ello es necesario hacer una referencia al elemento y mediante la propiedad **style** seguido del valor del atributo se podrá modificar.

```
document.getElementById(id).style.property = new style
```

Ejemplos para modificar el estilo de CSS.

```
<p id="p1"> Web personal </p>

<script>

    document.getElementById("p2").style.color = "red";
    document.getElementById("p2").style.fontFamily = "Arial";
    document.getElementById("p2").style.visibility = "visible";
    document.getElementById("p2").style.textShadow = "5px 5px blue";
    document.getElementById("myDiv").style.borderColor = "red";
    document.getElementById("myDIV").style.borderRadius = "25px";
    document.getElementById("myDiv").style.borderWidth = "thick";
    document.getElementById("myDIV").style.display = "none";
    document.getElementById("myImg").style.filter = "grayscale(100%)";
    document.getElementById("myBtn").style.left = "100px";
    document.getElementById("myDiv").style.margin = "50px 10px";
    document.getElementById("myDIV").style.overflow = "scroll";
    document.getElementById("myP").style.textDecoration = "underline";
    document.getElementById("myBtn").style.top = "100px";
    document.getElementById("myDIV").style.transform = "rotate(7deg)";
    document.getElementById("myBtn").style.width = "300px";
```

```
document.getElementById("img1").style.zIndex = "1";  
</script>
```

Aspectos muy importantes a tener en cuenta si hay que modificar atributos CSS desde JavaScript

1. Los valores de los atributos se pasan siempre como una cadena de texto entre dobles comillas o utilizando una variable para almacenar el contenido, pero siempre en formato texto. Si se le pasa un valor en formato diferente a texto no modificará ningún estilo del código.
2. Algunos atributos aplicados desde JS no se corresponden con el mismo nombre que en CSS ya que no se llegó a un consenso. Por ejemplo, el atributo '**text-shadow**' de CSS en JavaScript pasa a ser '**textShadow**' (junto y con alguna letra en mayúscula). Dicho de otra forma, todas las propiedades CSS que contienen un guion se eliminan en JavaScript.
3. Todos los atributos CSS tienen su correspondencia en JavaScript y hay que respetar los prefijos en los diferentes navegadores para que se puedan ejecutar correctamente en todos ellos. Exactamente lo mismo que al programar el diseño en CSS para distintos navegadores. Esta característica con los años desaparecerá y todo será estandarizado por los navegadores.

```
<script>  
function myFunction() {  
    document.getElementById("miDIV").style.WebkitAnimationDuration="1s";  
    document.getElementById("miDIV").style.animationDuration="1s";}  
</script>
```

Desde los métodos del DOM es posible **cambiar el contenido** de los elementos HTML.

➤ Crear contenido

Para crear contenido dinámico se deben generar los diferentes nodos con sus elementos, establecer el tipo de relación entre ellos y su ubicación en el DOM. Los siguientes ejemplos muestran la creación de nodos añadiéndolos a otros elementos.

Ejemplo 1: Creación de un párrafo con contenido tipo texto en su interior.

```
// Crea un elemento del tipo <p> y lo almacena en variable.  
var parrafo = document.createElement("p");  
  
// Crea un contenido de tipo texto y lo almacena en variable.  
var texto = document.createTextNode("Texto del párrafo");  
  
// Añade el texto dentro del elemento <p> dándoles jerarquía.  
// Siempre lo añadirá a continuación del último hijo.  
parrafo.appendChild(texto);
```

Ejemplo 2: Creación de una imagen y posterior inclusión dentro de una etiqueta <div> que previamente ha sido capturada.

```
// Crea un elemento del tipo <img> y lo almacena en variable.  
var imagen = document.createElement("img");  
  
// Se asigna al atributo 'src' la imagen a mostrar. El método  
// utiliza dos parámetros, la propiedad de la etiqueta y el valor.  
// Con el método 'setAttribute()' se cargan propiedades, y con  
// 'getAttribute()' se recupera un valor de un atributo.  
imagen.setAttribute("src", "img/bicicleta.png");  
  
// Captura una caja de la página web mediante su 'id'.  
var capa = document.getElementById("capa");  
  
// Añade la imagen dentro de la capa dándoles jerarquía. El método  
// 'appendChild()' siempre añade al final del último nodo. El método  
// 'prepend()' añade al principio de todos los nodos.  
capa.appendChild(imagen);
```

➤ Cambiar contenido

En JavaScript mediante la propiedad **innerHTML** se puede modificar o añadir nuevo contenido a la web. El contenido se pasará como tipo texto permitiendo añadir etiquetas HTML que serán reconocibles siempre que

```
// Párrafo con un texto
<p id="p1"> Mi página </p>

<script>
    // Modifica el texto con el identificador 'p1'
    document.getElementById("p1").innerHTML = "Página Personal";

    // Modifica el texto y añade las etiquetas <b> dentro de 'p1'.
    document.getElementById("p1").innerHTML=<b>Web Personal</b>;
</script>
```

➤ Cambiar el valor de un atributo

Desde JavaScript se pueden modificar los valores de los atributos en HTML.

```


// Se cambian los valores de los tres atributos de <img>
<script>
    document.getElementById("myImage").src = "paisaje.jpg";
    document.getElementById("myImage").width = 320;
    document.getElementById("myImage").height = 400;
</script>
```

➤ Eliminar nodos

Cuando queramos eliminar un nodo debemos tener la referencia del elemento padre y del elemento hijo. El método utilizado será '**removeChild()**' y nos permitirá **eliminar un nodo** del cual tenemos la referencia. Otro método similar que nos permite reemplazar un nodo existente por otro es '**replaceChild()**'. La sintaxis en ambos casos será la siguiente:

```
nodoPadre.removeChild(nodoHijoEliminar);

nodoPadre.replaceChild(nodoNuevo, nodoHijoEliminar);
```

Los siguientes métodos y propiedades se pueden aplicar sobre todos los elementos HTML del DOM.

Propiedad / Método ()	Descripción
element.accessKey	Devuelve el atributo de <u>clave de acceso rápido</u> (combinación teclas) de un elemento. <pre> MI WEB var x = document.getElementById("w3s").accessKey;</pre>
element.addEventListener()	Conecta un controlador de eventos al elemento especificado. Se queda escuchando a la espera que el evento sea ejecutado por el usuario. <pre>var x = document.getElementById("myBtn"); x.addEventListener("click", funcion1);</pre>
element.appendChild()	Agrega un nodo secundario, a un elemento, y <u>se posiciona al final de los nodos hijos</u> . <pre>var para = document.createElement("p"); var node = document.createTextNode("Nuevo parrafo"); para.appendChild(node);</pre>
element.attributes	Devuelve el número de atributos HTML que tiene una etiqueta. <pre><input id="myInput" type="text" onkeydown="teclapulsada()"> var x = document.getElementById("myBtn").attributes.length; // Devuelve 3</pre>
element.blur()	Elimina o quita el foco de un elemento siempre que lo haya cogido previamente. <pre>document.getElementById("myAnchor").blur();</pre>
element.childElementCount	Devuelve el número de hijos de un elemento. <pre>var x = document.getElementById("myDIV").childElementCount;</pre>
element.childNodes	Devuelve una colección de nodos de un elemento y los almacena en un array (incluye texto y comentarios) <pre>// Captura todos los hijos del elemento body y los guarda en un array var c = document.body.childNodes;</pre>

element.children	Devuelve una colección de nodos de un elemento y los almacena en un array (excluye texto y comentarios)
<pre>// Captura todos los hijos en 'c' y aplica color rojo al segundo elemento var c = document.getElementById("myDIV").children; c[1].style.backgroundColor = "red";</pre>	
element.classList	Permite añadir o eliminar un estilo a un elemento HTML. Dispone de varios métodos para realizar estas acciones.
<pre>document.getElementById("myDIV").classList.remove("mystyle"); document.getElementById("myDIV").classList.add("mystyle", "mystyle2"); document.getElementById("myDIV").classList.contains("mystyle");</pre>	
element.className	Asigna o devuelve una clase a un elemento. Esta acción no se debería realizar mediante el método 'setAttribute()' aunque funcione ya que los atributos propios de las etiquetas HTML se recomienda asignarlos directamente desde la referencia que se tiene del objeto.
<pre>document.getElementById("myDIV").className = "mystyle"; // Asigna clase</pre>	
element.click()	Simula un clic de ratón sobre un elemento. Por ejemplo, en un botón 'radio' lo dejaría marcado.
<pre>document.getElementById("myCheck").click();</pre>	
element.clientHeight	Devuelve la altura de un elemento incluyendo el padding.
<pre>document.getElementById("myDIV").clientHeight;</pre>	
element.clientLeft	Devuelve el <u>ancho del borde izquierdo</u> de un elemento.
<pre>document.getElementById("myDIV").clientLeft;</pre>	
element.clientTop	Devuelve el <u>ancho del borde superior</u> de un elemento.
<pre>document.getElementById("myDIV").clientTop;</pre>	
element.clientWidth	<u>Devuelve el ancho del elemento</u> incluyendo el padding
<pre>document.getElementById("myDIV").clientWidth;</pre>	

element.cloneNode()	Clona un nodo. Tiene un parametro booleano para configurar. Con 'True' clona solo el nodo y con 'False' el nodo y sus descendientes.
<pre>// Captura el primer elemento DIV de la web, lo clona y posiciona al final var elemento = document.getElementsByTagName("DIV")[0]; var cln = elemento.cloneNode(true); document.body.appendChild(cln);</pre>	
elem.compareDocumentPosition()	Compara dos elementos y devuelve el tipo de relación entre ellos. Valores 1. No hay relación en el documento. 2. p1 posicionado antes p2 4. p1 posicionado después p2 8. p1 posicionado dentro p2 16. p2 posicionado dentro de p1
<pre>var p1 = document.getElementById("p1"); var p2 = document.getElementById("p2"); var x = p1.compareDocumentPosition(p2);</pre>	
element.contains()	Devuelve 'true' si un nodo es descendiente de un nodo, de lo contrario 'false'.
<pre>var span = document.getElementById("mySPAN"); var div = document.getElementById("myDIV").contains(span);</pre>	
element.contentEditable	Establece o devuelve si el contenido de un elemento es editable o no. Devuelve 'true' o 'false' si es o no eseditable.
<pre>var x = document.getElementById("myP").contentEditable;</pre>	
element.dir	Devuelve o establece el valor del atributo 'dir' (dirección texto).
<pre>document.getElementById("myP").dir = "rtl";</pre>	
element.firstChild	Devuelve el primer nodo hijo de un elemento (el texto que contenga o etiqueta).
<pre>var x = document.getElementById("myList").firstChild.innerHTML;</pre>	
element.firstElementChild	Devuelve el primer elemento hijo de tipo etiqueta.
<pre>var x = document.getElementById("myList").firstElementChild;</pre>	

element.focus()	Hace que un elemento coja el foco.
document.getElementById("myInput"). focus() ;	
element.getAttribute()	Devuelve el valor que contiene una clase específica. Para ello previamente todas las etiquetas se deben haber guardado en un array con getElementsByTagName()
// Captura todos elementos <h1> y extrae nombre class posición 1 del array var x = document.getElementsByTagName("H1")[1].getAttribute("class");	
element.getAttributeNode()	Similar a getAttribute, excepto que en vez de devolver una cadena devuelve una instancia a Attribute.
element.getElementsByTagName()	Devuelve un array de un nodo con todos los elementos hijos de una etiqueta especificada.
// Captura todos los elementos <p> y los guarda en variable 'x' como array var x = document.getElementsByTagName("P"); var i; // Recorre el array pintando rojo el color de fondo de los párrafos. for (i = 0; i < x.length; i++) { x[i].style.backgroundColor = "red"; } // Cambia el valor de la primera posición del array capturado. document.getElementsByTagName("P")[0].innerHTML = "Insertar texto"; // La propiedad length devuelve el numero de hijos que contenga el nodo. var x = document.getElementsByTagName("LI").length;	
element.getElementsByClassName()	Devuelve un array de un nodo con todos los elementos hijos de una clase especificada.
// Captura todos los elementos <p> y los guarda en variable 'x' como array var x = document.getElementsByClassName("clase1"); var i; // Recorre el array pintando rojo el color de fondo de los párrafos. for (i = 0; i < x.length; i++) { x[i].style.backgroundColor = "red"; }	

```
// Cambia el valor de la primera posición del array capturado.
document.getElementsByClassName("clase1")[0].innerHTML = "Insertar texto";

// La propiedad length devuelve el número de hijos que contenga el nodo.
var x = document.getElementsByClassName("clase1").length;
```

element.getFeature()	Devuelve un objeto que implementa las API de una característica especificada.
element.hasAttribute()	Devuelve 'True' si un elemento tiene un atributo especificado.
// Comprueba si el elemento 'myboton' tiene el atributo 'onclick'	var x = document.getElementById("myBoton").hasAttribute("onclick");
element.hasAttributes()	Devuelve 'True' si el elemento tiene algun atributo.
var x = document.getElementById("myBoton").hasAttributes();	
element.hasChildNodes()	El método hasChildNodes () devuelve true si el nodo especificado tiene nodos secundarios, de lo contrario false
var x = document.getElementById("myDIV").hasChildNodes();	
element.id	Asigna o devuelve el atributo id de un elemento.
// Asigna un nuevo valor id al elemento seleccionado.	document.getElementById("myDIV").id = "myNewDIV";
element.innerHTML	Asigna o devuelve el contenido de un elemento.
// Asigna al elemento 'box' el texto “Cambiar contenido”.	document.getElementById("box").innerHTML = "Cambiar contenido.;"
element.insertBefore()	Inserta un nuevo hijo nodo antes del especificado. El 1er parámetro es el nuevo nodo y el 2º parámetro el nodo delante del cual se ubicará.
var newNode = document.createElement("LI"); // Crea etiqueta 	var textnode = document.createTextNode("Agua"); // Crea texto para
newNode.appendChild(textnode); // Introduce texto dentro etiqueta 	
// Captura el elemento con el id 'mylist'	

```
var list = document.getElementById("myList");

// Inserta newNode delante del primer elemento de la lista.
list.insertBefore(newNode, list.childNodes[0]);
```

element.isContentEditable	Devuelve true si el contenido de un elemento es editable, de lo contrario devuelve false. Es una propiedad de <u>solo lectura</u> .
----------------------------------	---

```
var x = document.getElementById("myP").isContentEditable;
```

element.isDefaultNamespace()	Averigua si el espacio de nombres definido es el espacio de nombres por defecto.
-------------------------------------	--

```
var x = document.documentElement.isDefaultNamespace("http://www.w3.org");
```

element.isEqualNode()	Devuelve si dos elementos son iguales en el contenido (el texto interior).
------------------------------	--

```
// Coge el primer hijo de cada lista y los compara por si son iguales.
var item1 = document.getElementById("myList1").firstChild;
var item2 = document.getElementById("myList2").firstChild;
var x = item1.isEqualNode(item2);
```

element.isSameNode()	Comprueba si dos elementos son el mismo nodo, devuelve true. Se puede referenciar a un nodo de formas diferentes por tanto, se puede llamar al mismo nodo sin darnos cuenta.
-----------------------------	--

```
var x = item1.isSameNode(item2);
```

element.isSupported()	Devuelve true si se admite una característica especificada en el elemento.
------------------------------	--

element.lang	Establece o devuelve el valor del atributo lang de un elemento.
---------------------	---

```
var x = document.getElementById("myP").lang;
```

element.lastChild	Devuelve el último nodo hijo de un elemento como nodo elemento, nodo texto, y nodo comentario.
--------------------------	--

```
var list = document.getElementById("myList").lastChild;
document.getElementById("demo").innerHTML = list.innerHTML;
```

element.lastElementChild	Devuelve el último nodo hijo de un elemento.
---------------------------------	--

element.namespaceURI	La propiedad namespaceURI devuelve el URI del namespace del nodo especificado. Elemento de solo lectura
element.nextSibling	Devuelve una referencia al siguiente hermano que puede ser element(tag) o no.
element.nextElementSibling	Devuelve la referencia del siguiente hermano que sea de tipo element (tag)
element.nodeName	Devuelven el nombre de la etiqueta del nodo, como texto en mayúsculas (por ejemplo H1, DIV, SPAN...)
element.nodeType	Devuelve un número que identifica el tipo de nodo (9 para document, 1 para element, 3 para text, 8 para comment)
element.nodeValue	Asigna o devuelve el valor de un nodo de tipo comentario o de tipo texto exclusivamente.
element.normalize()	Joins adjacent text nodes and removes empty text nodes in an element
element.offsetHeight	Returns the height of an element, including padding, border and scrollbar
element.offsetWidth	Returns the width of an element, including padding, border and scrollbar
element.offsetLeft	Returns the horizontal offset position of an element
element.offsetParent	Returns the offset container of an element
element.offsetTop	Returns the vertical offset position of an element
element.ownerDocument	Returns the root element (document object) for an element
element.parentNode	Devuelve el nodo padre de un elemento
element.parentElement	Devuelve una referencia de tipo element (tag) del parent del element seleccionado.
element.previousSibling	Devuelve una referencia al anterior hermano que puede ser element(tag) o no.
element.previousElementSibling	Devuelve una referencia al anterior hermano de tipo element(tag) .

element.querySelector()	Devuelve el primer elemento hijo de un elemento que se encuentra en un nivel superior que coincide con un selector CSS especificado.
element.querySelectorAll()	Devuelve en un array con todos los elementos hijos de un elemento de nivel superior que coincide con un selector CSS especificado.
element.removeAttribute()	Elimina un atributo específico de un elemento.
element.removeAttributeNode()	Removes a specified attribute node, and returns the removed node
element.removeChild()	Elimina un nodo hijo de un elemento.
element.replaceChild()	Reemplaza un nodo hijo en un elemento
element.removeEventListener()	Removes an event handler that has been attached with the addEventListener() method
element.scrollHeight	Returns the entire height of an element, including padding
element.scrollIntoView()	Scrolls the specified element into the visible area of the browser window
element.scrollLeft	Establece o devuelve el número de píxeles que el contenido de un elemento se desplaza horizontalmente
element.scrollTop	Sets or returns the number of pixels an element's content is scrolled vertically
element.scrollWidth	Devuelve todo el ancho de un elemento, incluido el relleno
element.setAttribute()	Asigna o cambia un atributo específico. Este sistema no se debe utilizar si deseamos acceder a los atributos de una etiqueta como por ejemplo 'value', 'id', 'class' etc.. Para ello se accede directamente desde el elemento.
element.setAttributeNode()	Sets or changes the specified attribute node
element.style	Establece o devuelve el valor del atributo style de un elemento.
element.tabIndex	Sets or returns the value of the tabindex attribute of an element
element.tagName	Returns the tag name of an element
element.textContent	Sets or returns the textual content of a node and its descendants
element.title	Sets or returns the value of the title attribute of an element

element.toString()	Converts an element to a string
nodelist.item()	Returns the node at the specified index in a NodeList
nodelist.length	Devuelve el número de elementos de una lista de nodos.



OBJETO WINDOW (BOM)

JavaScript permite "comunicarse" o interactuar con el navegador. El objeto **Window** está soportado por todos los navegadores en la actualidad. Es el objeto de mayor jerarquía y hace referencia a la ventana del navegador utilizada. Todos los objetos de un documento HTML, ya sean variables globales, funciones o propiedades pertenecen al objeto Window, así como el objeto **Document**.

```
// Window es el objeto de mayor jerarquía y todos dependen de él.  
window.document.getElementById("header");
```

El objeto Window dispone de una serie de propiedades (para obtener información de la ventana) y métodos (para realizar acciones sobre la ventana).

Las variables globales y funciones globales pertenecen al objeto Window.

Listado de los métodos mas utilizados del objeto Window

- **Alert()**: Muestra un diálogo con un mensaje en la ventana del navegador.

```
// Muestra el siguiente mensaje por pantalla  
alert("Has pulsado una tecla");
```

- **Prompt()**: Sirve para introducir caracteres por teclado. Si el valor introducido es un número se deberá utilizar el método genérico parseInt() para realizar la conversión a número sino el navegador lo interpreta como texto.

```
// Primer param nombre ventana, segundo valor por defecto  
var nombre = prompt('Ingrese nombre:', ''');
```

- **Confirm()**: Muestra un dialogo de confirmación con los botones Confirmar y Cancelar. Devuelve True o False y esos valores son los que aprovecharemos para lanzar uno u otro método.

```
// 'respuesta' guarda el valor booleano devuelto por confirm  
var respuesta = confirm("Presione alguno de los dos botones");  
if (respuesta == true)  
    alert("presionó aceptar");  
else  
    alert("presionó cancelar");
```

- **Open() // Close()**: Abre o cierra una ventana del navegador. Estos métodos tienen múltiples atributos de configuración sobre una nueva ventana. Al ejecutar el método Open() se genera un objeto del mismo tipo que se puede almacenar en una variable.

En el siguiente ejemplo desde el botón se llama a la función abrir. Esta función crea una variable del tipo ventana (que no sería necesaria ya que se puede escribir directamente Open() y después escribe sobre la ventana. Open() devuelve un valor que corresponde al objeto creado.

```
// Abre una ventana con pocos parámetros. El resultado que
devuelve el método se guarda en un variable.
```

```
function openWin() {
    var myWindow = window.open("", "myWindow", "width=200,
height=100");
}
```

```
//Esta función abre una ventana y se le pasan varios parámetros
function abrirconParametros {
```

```
// Open() pasándole diferentes parámetros de configuración.
var ventana = open('http://www.google.com', '_blank',
'top=100, left=100, width=300, height=300, location=no,
status=yes, toolbar=no, menubar=no, scrollbars=yes,
resizable=yes');
```

Primer parámetro: — '<http://www.google.com>' — es el nombre de la página que se desea abrir.

Segundo parámetro: — '_blank' —, es la ventana de destino (_blank, _parent, _self, _top, o un nombre personalizado por si se tiene que hacer referencia a la ventana en algún momento).

Tercer parámetro: Son una serie de propiedades o especificaciones de configuración que afectan a la nueva ventana que va a ser creada.

- **width** y **height** son su anchura y altura.
- **top** y **left** son su distancia al borde superior e izquierdo de la pantalla.
- **menubar** especifica si se quiere que el menú del navegador sea visible para la nueva ventana.
- **resizable** especifica si se permite o no que la nueva ventana sea redimensionable.
- **location** especifica si se muestra o no la caja para introducir direcciones.
- **scrollbars** especifica si se permite que aparezcan barras de desplazamiento o no.
- **status** especifica si se quiere visible la barra inferior de estatus.
- **toolbar** especifica si se hace o no visible la barra de herramientas del navegador.

Ejemplos del método open() y close()

```
// Abre el enlace en la misma ventana y el objeto devuelto se almacena en la variable 'a'.
```

```
var a= window.open('URL1', '_self');
```

```
// Abre el enlace en una nueva ventana y el objeto devuelto se almacena en la variable 'b'.
```

```
var b= window.open( "URL2 ","_blank","toolbar=no, location=no, status=no, menubar=no, scrollbars=no, resizable=yes, width=500, height=185, top=185, left=240");
```

```
// Crea la URL en una ventana nueva con los parámetros especificados y los almacena en la variable 'c'. Posteriormente se cierra la ventana desde la variable que ha almacenado la ventana.
```

```
var c = window.open("https://www.bcn.es", "_blank", "toolbar=yes, scrollbars=yes, resizable=yes, top=500, left=500, width=400, height=400");
```

```
// El método close() cierra la ventana del navegador. Algunos navegadores no ejecutan correctamente este método.
```

```
c.close();
```

Propiedades objeto Window

A continuación, se muestra un listado con las propiedades del objeto Window más utilizadas:

- **closed:** Devuelve un valor booleano que determina si una ventana ha sido cerrada o no.

```
if (myWindow.closed){  
    alert("ventana cerrada");  
} else {  
    alert("ventana no cerrada"); }
```

- **frameElement:** Devuelve el elemento <iframe> donde se encuentra la ventana actual. Si la ventana no se encuentra dentro de un elemento <iframe> devuelve valor Null.

```
var frame = window.frameElement;
```

- **frames:** Devuelve una matriz o array con todos los elementos <iframe> de la ventana actual.

```
var frames = window.frames;
```

- **innerWidth // innerHeight:** Devuelven el ancho y el alto interior respectivamente de la ventana. Estas propiedades son de solo lectura. El objeto 'screen' realiza la misma acción que 'window' pero en este caso la información la extrae del navegador y no del documento.

```
var w = window.innerWidth; // Anchura en pixeles de la ventana
var h = window.innerHeight; // Altura en pixeles de la ventana

var w = screen.width; // Anchura en pixeles de la ventana
var h = screen.height; // Altura en pixeles de la ventana
```

- **length:** (-- Obsoleta --) Devuelve el número de elementos <iframe> de la ventana actual. La propiedad se puede utilizar con frame pero NO se admite desde HTML5.

```
var x = window.length; // Cantidad elementos iframe en la ventana
```

- **name:** Devuelve o establece el nombre de una ventana.

```
var myWindow = window.open("", "MsgWindow", "width=200, height=100");
myWindow.document.write("<p> Nombre ventana:" + myWindow.name + "</p>");
```

- **opener:** Devuelve una referencia de la ventana padre que creó la ventana.

```
var myWindow = window.open("", "myWindow", "width=200, height=100");
myWindow.opener.document.write("<p> Esta es la ventana padre </p>");
```

- **outerWidth // outerHeight:** Devuelven el ancho y el alto respectivamente de la ventana incluyendo la barra de herramientas y la barra de desplazamiento.

```
var w = window.outerWidth; // Anchura en pixeles de la ventana
var h = window.outerHeight; // Altura en pixeles de la ventana
```

- **pageXOffset // pageYOffset:** Devuelven en píxeles el desplazamiento del documento actual tomando como referencia la esquina superior izquierda. Las propiedades scrollX y scrollY realizan la misma función.

```
window.scrollBy(100, 100); // Desplaza la ventana
alert(window.pageXOffset + window.pageYOffset);
```

- **self:** Devuelve una referencia de la ventana a si misma.

```
var w = window.self;
```

- **top:** Devuelve la posición más alta de la ventana del navegador.

```
var w = window.top;
```

Resto de métodos del objeto Window

A continuación, se muestra un listado con los métodos más habituales del objeto Window:

- **blur():** Elimina el foco de la ventana. Por defecto las ventanas al ser llamadas y creadas cogen el foco. Con este método les quitamos el foco a una ventana activa.

```
// Abre una nueva ventana
var myWindow = window.open("", "", "width=200, height=100");

// Quita el foco a la ventana que hemos creado.
myWindow.blur();
```

- **focus():** Asigna el foco de la ventana. Todas las ventanas en el momento de ser invocadas reciben el foco de la aplicación.

```
// Abre una nueva ventana
var myWindow = window.open("", "", "width=200, height=100");

// Asigna el foco a la ventana que hemos creado.
myWindow.focus();
```

- **moveBy():** Permite modificar la posición de la ventana a partir de unas coordenadas X e Y, siempre tomando como referencia la posición relativa donde se encuentra ubicada, es decir, su posición actual.

```
//Abre una nueva ventana
myWindow = window.open("", "myWindow", "width=200, height=100");

// Mueve la ventana tantos píxeles en los ejes de coordenadas 'x'
// e 'y' tomando como punto de inicio la posición actual.
myWindow.moveBy(250, 250);
```

- **moveTo()**: Permite modificar la posición de la ventana a partir de unas coordenadas X e Y. En este caso las coordenadas pasadas corresponden a la posición absoluta de la pantalla y no como el método anterior que toma como referencia su posición actual.

```
// Abre una nueva Ventana.  
myWindow = window.open("", "myWindow", "width=200, height=100");  
  
// La posiciona dentro de la ventana.  
// Toma como referencia los datos absolutos de la pantalla.  
myWindow.moveTo(300, 200);
```

- **print()**: Esta método muestra el dialogo para imprimir la página.

```
window.print();
```

- **resizeBy()**: Redimensiona la ventana al tamaño especificado sumando los valores pasados como parámetros al tamaño que tenga la ventana en ese momento.

```
// Crea una nueva ventana.  
myWindow = window.open("", "", "width=100, height=100");  
  
// Redimensiona la ventana creada sumándole 250px (en eje X) y  
// 200px (en eje Y) al tamaño que tenga la ventana en ese momento.  
myWindow.resizeBy(250, 200);
```

- **resizeTo()**: Redimensiona la ventana al tamaño especificado sin tener en cuenta los valores que tengan la ventana en ese momento, le da el tamaño especificado en los parámetros.

```
// Crea una nueva ventana.  
myWindow = window.open("", "", "width=100, height=100");  
  
// Redimensiona la ventana que hemos a 250 x 200 pixeles la  
// ventana sin tener en cuenta los valores que tenga en ese momento.  
myWindow.resizeTo(250, 200);
```

- **scrollBy()**: Desplaza la barra de desplazamiento tantos píxeles como se especifique pero desde la posición actual de la barra. El primer parámetro hace referencia a la barra horizontal (X) y el segundo parámetro hace referencia a la barra vertical (Y).

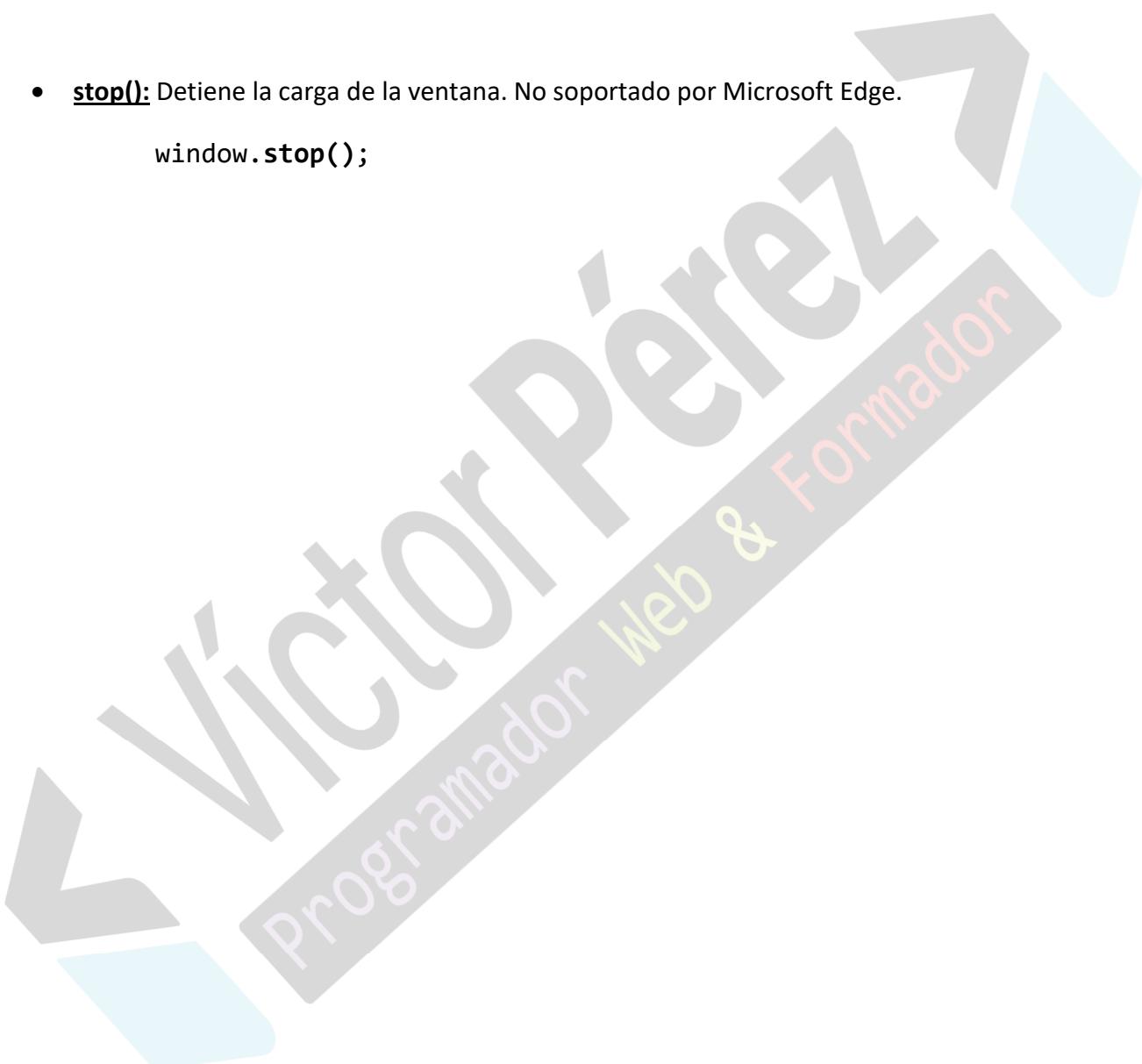
```
window.scrollBy(0, 100); // Desplaza barra vertical  
window.scrollBy(200, 0); // Desplaza barra horizontal
```

- **scrollTo()**: Desplaza la barra de desplazamiento tantos píxeles como se especifique pero **tomando como referencia relativa**, es decir, la ventana del navegador. El primer parámetro hace referencia a la barra horizontal (X) y el segundo parámetro hace referencia a la barra vertical (Y).

```
window.scrollTo(300, 100); // Desplaza barra vertical y horiz.  
window.scrollTo(200, 0); // Desplaza barra horizontal.
```

- **stop()**: Detiene la carga de la ventana. No soportado por Microsoft Edge.

```
window.stop();
```



OBJETO NAVIGATOR

El objeto **NAVIGATOR** contiene información sobre el navegador web. Las propiedades más comunes de este objeto son las que se muestran a continuación. Es posible que la implementación de este objeto en determinados navegadores varíe del estandar.

El correcto funcionamiento de estos métodos depende sobretodo del navegador ya que es quien determina si estos métodos podrán ser o no utilizados.

- **appCodename**: (-- Obsoleta --). Devuelve el nombre del código del navegador.
- **appName** : (-- Obsoleta --). Almacena el nombre oficial del navegador.
- **appVersion** : Retorna la versión del navegador.
- **cookieEnabled** : Retorna booleano si las cookies están activas en el navegador.

```
navigator.cookieEnabled; // Devuelve 'true' o 'false'
```

- **geolocation**: Devuelve un objeto geolocation que permite obtener la posición del usuario.
- **language**: Devuelve el idioma del navegador, ("es", "es-ES", "en", "fr", "fr-FR")
- **online**: Determina si el navegador tiene o no conexión a Internet.
- **platform** : Retorna información sobre el sistema operativo.

```
navigator.platform; // Puede devolver "MacIntel", "Win32", etc..
```

- **product** : (-- Obsoleta --). Devuelve el nombre del motor del navegador.
- **user-agent**: Devuelve la versión del navegador.
- **vibrate**: Si el dispositivo y el navegador lo permiten, ejecuta una vibración que dura tantos milisegundos como los indicados por parámetro. Si por parámetro se le pasa un array de números se realizan tantas vibraciones como números se hayan indicado con una pausa entre ellos y con la duración de cada valor.

```
navigator.vibrate(200); // Realiza una vibración de 200 milis
```

```
navigator.vibrate([100, 30, 50]); // Realiza varias vibraciones
```

El objeto Navigator dispone del método **javaEnabled()** que devuelve true o false que determina si el navegador está preparado para soportar la ejecución de programas escritos en Java (Applets).

El siguiente código muestra un ejemplo de cada una de las propiedades del objeto navigator

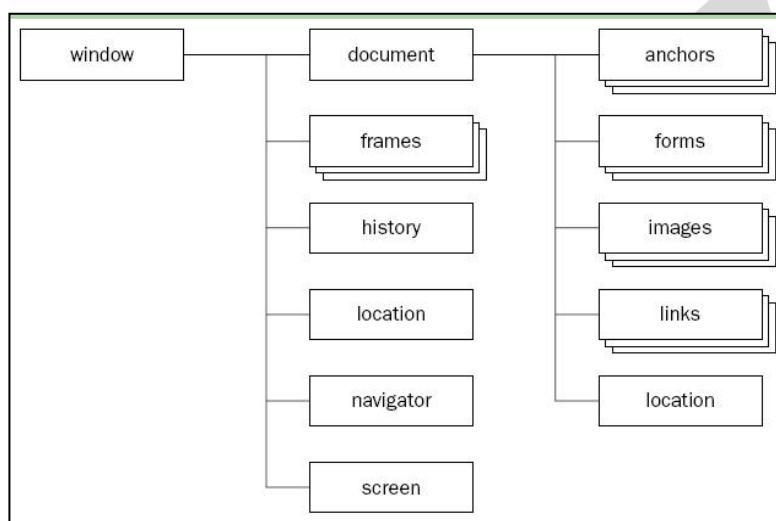
```
var txt = ""  
  
txt += "<p>Browser CodeName: " + navigator.appCodeName + "</p>";  
  
txt += "<p>Browser Name: " + navigator.appName + "</p>";  
  
txt += "<p>Browser Version: " + navigator.appVersion + "</p>";  
  
txt += "<p>Cookies Enabled: " + navigator.cookieEnabled + "</p>";  
  
txt += "<p>Browser Language: " + navigator.language + "</p>";  
  
txt += "<p>Browser Online: " + navigator.online + "</p>";  
  
txt += "<p>Platform: " + navigator.platform + "</p>";  
  
txt += "<p>User-agent header: " + navigator.userAgent + "</p>";  
  
document.getElementById("demo").innerHTML = txt;
```

```
Browser CodeName: Mozilla  
Browser Name: Netscape  
Browser Version: 5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36  
Cookies Enabled: true  
Browser Language: es  
Browser Online: true  
Platform: Win32  
User-agent header: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36
```

OBJETO LOCATION

El objeto **LOCATION** depende directamente del objeto **window**. Cuando le asignamos una nueva dirección a la propiedad location del objeto **window**, el navegador redirecciona a dicha página. Este objeto corresponde a la URL de la página web en uso. Su principal función es consultar las diferentes partes que forman la URL, como el dominio, el protocolo o el puerto.

Su finalidad principal es, por una parte, modificar el objeto **location** para cambiar a una nueva URL, y extraer los componentes de dicha URL de forma separada para poder trabajar con ellos de forma individual si es el caso.



Los **Métodos** utilizados por el objeto Location son los siguientes:

- [Reload\(\)](#): Carga de nuevo el documento actual.

```
<input type="button" value="recargar"
      onclick="location.reload();">
```

- [Replace\(\)](#): Sustituye la URL actual por otra URL. Elimina la dirección URL del historial.

```
<input type="button" value="Con replace"
      onclick="location.replace('users.html');">
```

- [Assign\(\)](#): Carga un nuevo documento con la diferencia que modifica el historial de URL visitadas. Equivalente a location.href = "http://www.ejemplo.com".

```
location.assign("https://www.w3schools.com");
```

Las **Propiedades** del objeto Location son:

- **hash**: Retorna la parte de la URL detrás del hashtag (#).
- **host**: Devuelve o asigna el nombre del servidor y el número del puerto, dentro de la URL.

```
var x = location.host; // Devuelve 'www.bcn.cat:80'
```

- **hostname**: Devuelve o asigna el nombre de dominio del servidor (o IP), dentro de la URL.

```
var x = location.hostname; // Devuelve 'www.bcn.cat'
```

- **href**: Devuelve la cadena que contiene la URL completa. También es posible asignar un valor a esta propiedad que nos redireccionará a la web especificada.

```
// Enlaza con una página web o un recurso.
```

```
location.href = "https://www.bcn.cat/recursos/documentos.html";
```

```
location.href = "#top"; // Enlaza con ancla en web
```

- **origin**: Devuelve la URL canónica de la web.

```
location.origin; // Devuelve 'https://www.bcn.cat'
```

- **pathname**: Retorna la ruta del recurso sin mostrar la URL canónica empezando con '/'.

```
location.pathname; // Devuelve '/recursos/documentos.html'
```

- **port**: Devuelve o asigna el número de puerto del servidor, dentro de la URL. Si el valor no se muestra es porque está utilizando por defecto el 80:http o el 443:https.

```
var x = "Port: " + location.port;
```

- **protocol**: Retorna una cadena que contiene el protocolo utilizado (incluyendo los dos puntos), dentro de la URL.

```
location.protocol; // Devuelve cadena texto 'https:'
```

- **search**: Devuelve o asigna el listado de parámetros que continene la URL.

```
var x = location.search; // Ej: ?email=someone@example.com
```

Ejemplo con algunas propiedades comentadas arriba (capturando valor):

```
document.write("Location <b>href</b>: " + location.href + "<br>");  
document.write("Location <b>host</b>: " + location.host + "<br>");  
document.write("Location <b>hostname</b>:" + location.hostname + "<br>");  
document.write("Location <b>pathname</b>: " + location.pathname + "<br>");  
document.write("Location <b>port</b>: " + location.port + "<br>");  
document.write("Location <b>protocol</b>: " + location.protocol + "<br>");
```

Location **href**: http://www.webestilo.com/javascript/ejem/ejem10.html
Location **host**: www.webestilo.com
Location **hostname**: www.webestilo.com
Location **pathname**: /javascript/ejem/ejem10.html
Location **port**:
Location **protocol**: http:



OBJETO HISTORY

El objeto **HISTORY** tambien depende directamente del objeto Window. Este objeto almacena todas las páginas que visitamos. Los métodos que tiene se utilizan para extraer de la memoria del pc las páginas visitadas sin tener que pedirlas nuevamente al servidor.

Los métodos del objeto History son:

// Retrocede a la página anterior

```
window.history.back();
```

Ejemplo: Atras

// Avanza a la página siguiente almacenada en la caché de la máquina.

```
window.history.forward();
```

Ejemplo: Adelante

// Avanza o retrocede en la lista de páginas visitadas.

```
window.history.go();
```

Ejemplo: Volver 3

El siguiente ejemplo la función avanzar() llama al objeto **history**. El parámetro de history puede ser un valor en positivo o un valor en negativo que representa las páginas que avanza o retrocede.

```
<script type="text/javascript">
    function avanzar()
    {
        window.history.go(1);
    }
</script>
```

Retornar

Las propiedades del objeto History son:

- **Length**: Entero que contiene el número de entradas del historial (i.e., cuántas direcciones han sido visitadas).

```
if (window.history.length>0){
    window.history.go(1);
} else { alert('no hay otra página en la cache hacia adelante'); }
```

OBJETO SCREEN

El objeto **SCREEN** colabora directamente con el objeto window. Este objeto **muestra información acerca del monitor donde se está ejecutando el navegador**. Este objeto tiene seis propiedades de lectura y ningun método.

Propiedades del objeto Screen:

- **availHeight** : El alto de la pantalla en pixeles disponible para el navegador. Tiene en cuenta los elementos permanentes y semipermanentes y afecta al resultado retornado.
- **availWidth** : El ancho de la pantalla en pixeles disponible para el navegador.
- **colorDepth** : Representa el número de bits (24 o 32 bits) usados para representar los colores.
- **colorpixelDepth** : Retorna la resolución de la pantalla expresada en bits por pixel. En la mayoría de los casos retorna 24 por compatibilidades con monitores.
- **orientation**: Retorna un objeto con información sobre la orientación de la pantalla y si tiene un evento asociado al hecho de girar la pantalla.
- **height** : El alto de la pantalla en pixeles incluyendo la ventana del navegador.
- **width** : El ancho de la pantalla en pixeles incluyendo la ventana del navegador.

El siguiente código muestra un ejemplo de cada una de las propiedades.

```
</script type="text/javascript">

    document.write('Valores de las propiedades del objeto screen:<br>');

    document.write('availHeight :' + screen.availHeight + '<br>');
    document.write('availWidth :' + screen.availWidth + '<br>');
    document.write('height :' + screen.height + '<br>');
    document.write('width :' + screen.width + '<br>');
    document.write('colorDepth :' + screen.colorDepth);

</script>
```

Valores de las propiedades del objeto screen:
availHeight:728
availWidth:1366
height:768
width:1366
colorDepth:24

API Formularios

Es un grupo de propiedades, métodos y eventos que podemos utilizar para procesar formularios y crear nuestro propio sistema de validación. Los formularios son el claro ejemplo de validar el código antes de ser enviado a la base de datos.

Para acceder a los elementos de formulario se puede realizar, por ejemplo, con el método `getElementById()`. También es posible acceder a todos los elementos de un formulario si lo cargamos como colección de objetos a través de un array.

- **Cuadro de texto:** La captura se realiza siguiendo la función `getElementById()` seguida entre paréntesis del nombre del identificador del elemento del cuadro de texto.

```
Name: <input type="text" id="myText"> // Elemento de formulario texto  
Var nombre = document.getElementById("myText").value; // Captura valor  
  
// Activa o desactiva el campo.  
document.getElementById("myText").disabled = true;  
  
// Valor por defecto del cuadro texto.  
document.getElementById("myText").defaultValue = "Goofy";  
  
// Valor informativo campo (HTML5).  
document.getElementById("myText").placeholder = "Nombre...";  
  
// Activa el campo como solo lectura.  
document.getElementById("myText").readOnly = true;  
  
// Tamaño del cuadro de texto, no del número de letras.  
document.getElementById("myText").size = "10";  
  
// Devuelve el tipo de elemento, en este caso 'text'.  
document.getElementById("myText").type;  
  
// Devuelve el valor del atributo name.  
document.getElementById("myText").name;  
  
// Activa el foco sobre el cuadro de texto (HTML5)  
document.getElementById("myText").autofocus;
```

Tiene un método que es **select()**

```
// Selecciona el texto que hay dentro del cuadro texto
document.getElementById("myText").select();
```

- **Texto(password)**: Exactamente igual que el elemento 'text', pero con algún pequeño cambio en alguna propiedad. La forma de acceder a las propiedades del objeto mediante JS se muestra a continuación.

```
// Declaración de un objeto tipo 'password'
Password: <input type="password" id="myPsw" maxlength="8">
```

```
// Devuelve el tamaño atributo maxlength.
var x = document.getElementById("myPsw").maxLength;
```

```
// Reduce el tamaño del cuadro de texto password.
document.getElementById("myPsw").size="50";
```

```
// Desactiva el campo.
document.getElementById("myPsw").disabled=true;
```

```
// Deja el campo como estado en solo lectura.
document.getElementById("myPsw").readOnly=true;
```

Tiene un método que es 'select()'

```
// Selecciona el texto que hay dentro del cuadro texto.
document.getElementById("myText").select();
```

- **Select:** Este objeto difiere mucho de los anteriores. Para empezar el elemento es un menú desplegable y el evento que hace saltar a la función es **onChange()**, es decir, para cada cambio de valor en el menú saltará el evento.

```
// El elemento desplegable tiene 3 opciones.

<select id="select1" onChange="cambiarColor()">
    <option value="0xff0000"> Rojo </option>
    <option value="0x00ff00"> Verde </option>
    <option value="0x0000ff"> Azul </option>
</select>
```

El contenido del SELECT tiene tres datos por elemento o por opción. El índice, el texto y el value. Suponemos que hemos realizado la captura de la variable del formulario con la siguiente orden. La variable se llama selección.

```
// Captura el objeto del menú desplegable
var seleccion = document.getElementById('select1');
```

- Indice: Simplemente dice el **número de posición dentro del array**. En este caso tenemos 0, 1 y 2. Hay que tener en cuenta que los arrays empiezan siempre por la posición cero. Es poco habitual ir a buscar una posición del array pero existe la posibilidad. La orden para capturar el índice del elemento seleccionado es:

```
// Determina el índice que se ha seleccionado.
var x = seleccion.selectedIndex;

// Asigna el valor del índice a la propiedad.
document.getElementById("mySelect").selectedIndex = "2";
```

- Value: Esta captura es menos frecuente pero también se puede realizar. Es posible con el atributo 'value' asignar un valor y entonces directamente capturar el valor de esa variable para empezar a trabajar. La orden que se precisa para capturar el contenido de 'value' es la siguiente. Es idéntica que la anterior pero solo cambia el último parámetro. El valor que se mostrará será el color en código hexadecimal.

```
// Captura el campo 'value' a partir del indice
seleccion.options[seleccion.selectedIndex].value;
```

- Options: Devuelve el listado de los elementos en un array. El array se deberá recorrer para poder acceder a todos los elementos del menú desplegable.

```
// Captura el objeto menú desplegable 'mySelect'.  
var x = document.getElementById("mySelect");  
var txt = "";  
  
// Recorre array mostrando elementos uno debajo de otro.  
for (var i = 0; i < x.length; i++) {  
    txt = txt + x.options[i].text + "<br>"; }  
  
// Nos permite seleccionar más de un elemento.  
document.getElementById("mySelect").multiple=true;  
  
// Devuelve el número de elementos que tiene el desplegable  
var x = document.getElementById("mySelect").length;  
  
// Muestra el menú con más de una opción, en este caso 4.  
document.getElementById("mySelect").size="4";  
  
// Desactiva el elemento de menú desplegable.  
document.getElementById("mySelect").disabled=true;
```

Este objeto tiene dos métodos para añadir o eliminar elementos del menú desplegable.

- add(): Este método **añade elementos** a un menú desplegable. Para poder trabajar con estos métodos es necesario conocer el funcionamiento del DOM con sus propiedades y métodos de creación y eliminación de nodos.

```
function myFunction() {  
  
    // Captura el menú desplegable.  
    var x = document.getElementById("mySelect");  
  
    // Crea un nuevo nodo del tipo 'option'.  
    var option = document.createElement("option");  
  
    option.text = "Kiwi"; // Asigna un texto al nodo creado.  
  
    x.add(option); } // Añade nuevo nodo al menú desplegable
```

- remove(): Este método **elimina elementos del menú desplegable**.

```
// Captura el objeto y elimina un elemento del array
function myFunction() {
    var x = document.getElementById("mySelect");
    x.remove(x.selectedIndex);
}
```

- CheckBox: Los checkbox son grupos de elementos que pueden o no estar seleccionados. Todos los elementos del checkbox pertenecen al mismo type. Después cada uno tendrá su identificador. La creación de un checkbox es sencilla. El botón que lanza la función siempre utilizará el evento 'onClick()'.

```
<form>
    <input type="checkbox" id="checkbox1" value="JS"> JavaScript <br>
    <input type="checkbox" id="checkbox2"> PHP <br>
    <input type="checkbox" id="checkbox3"> JSP <br>
    <input type="button" value="Mostrar" onClick="contarSeleccion()">
</form>
```

La orden para comprobar si el elemento está chequeado es la siguiente instrucción.
El resultado es TRUE o FALSE.

```
document.getElementById('checkbox1').checked;
```

La orden para capturar el valor del campo 'value', si se ha configurado, es la siguiente.

```
document.getElementById('checkbox1').value;
```

Las principales características que nos interesan como programadores son:

1. Los *checkbox* marcados tienen el atributo *checked*. Cuando accedamos a un *checkbox* sabremos si el usuario lo ha marcado comprobando el atributo *checked*.
2. El *value* no lo escribe el usuario sino que está prefijado por el desarrollador HTML.
3. Todos los *checkbox* tienen el mismo nombre. Cuando accedamos a un *checkbox* por su *name* nos va a retornar un *HTMLCollection* con los *checkbox* seleccionados.

- Radio: Esta opción de formulario es muy similar a checkbox con la diferencia que solo se puede seleccionar una de las opciones disponibles. La creación de una opción tipo radio se realiza de la siguiente forma.

```
<form>
    <input type="radio" id="radio1" name="estudios">Sin estudios<br>
    <input type="radio" id="radio2" name="estudios">Primarios<br>
    <input type="radio" id="radio3" name="estudios">Secundarios<br>
    <input type="radio" id="radio4" name="estudios">Universitarios<br>
    <input type="button" value="Mostrar" onClick="mostrarSeleccion()">
</form>
```

La orden para poder capturar desde la función el valor seleccionado es.

```
// Determina si esta activado o no True o False
document.getElementById('radio1').checked;

// Desactiva el elemento radio.
document.getElementById("cats").disabled=true;

// Captura el valor del atributo value.
document.getElementById("myRadio").value;

// Devuelve valor atributo name.
<input type="radio" name="colors" value="red" id="myRadio">Red<br>
var x = document.getElementById("myRadio").name;

// Campo obligatorio antes de validar el formulario (HTML5)
var x = document.getElementById("myRadio").required;

// Asigna el valor de autofocus al elemento (HTML5).
document.getElementById("myRadio").autofocus;
```

NOTA: Este elemento de formulario dispone de algunas propiedades más de las explicadas en este manual y que pueden ser modificadas y utilizadas como lectura y escritura.

- TextArea: Es muy similar a área. En este caso el cuadro de texto solo es de una línea, pero el 'textarea' puede ser de varias.

```
<form>
    <textarea id="curriculum" rows="10" cols="50" > </textarea> <br>
    <input type="button" value="Mostrar" onClick="controlCaracter()">
</form>
```

Para obtener el valor de la variable la orden es la misma que para un texto.

```
// Este ejemplo muestra el contenido del textarea.
Var nombre = document.getElementById('curriculum').value;

// Poco utilizada y determina el tipo de elemento que se esta
// refereciando, en este caso dará como resultado 'textarea'.
nombre = document.getElementById('curriculum').type;

// Amplia el número de filas, hace mas alto el textarea
<textarea id="myTextarea" rows="2" cols="20" name="t">
document.getElementById("myTextarea").rows="10";

// Amplia el número de columnas del textarea, lo hace mas ancho.
<textarea id="myTextarea" rows="2" cols="20" name="t">
document.getElementById("myTextarea").cols="100";

// Devuelve el valor del atributo wrap.
<textarea id="myTextarea" rows="2" cols="20" name="t" wrap="hard">
var x = document.getElementById("myTextarea").wrap;

// Valor por defecto del textarea.
document.getElementById("myTextarea").defaultValue="New York";
```

NOTA: Este elemento de formulario dispone de más propiedades de las explicadas en este manual y que pueden ser modificadas y utilizadas como lectura y escritura.

La API de formularios dispone de dos métodos para trabajar la validación de los elementos del formulario.

- **checkValidity()**: Este método devuelve un valor booleano que **indica si el formulario es válido o no antes de ser enviado**. Se puede capturar un elemento del formulario o directamente todo el formulario. El usuario posteriormente deberá determinar que acción hay que realizar en caso de no ser correcta la validación del objeto.

```
// Dos elementos de formulario, uno tipo número y otro un botón.
```

```
<input id="id1" type="number" min="100" max="300" required>
<button onclick="myFunction()"> OK </button>
```

```
// Párrafo donde se mostrará el resultado final.
```

```
<p id="demo"> </p>
```

```
// Captura el elemento que es input para introducir números. Pasa a la validación mediante el método checkValidity() y escribe un mensaje en el párrafo si la validación no ha sido correcta.
```

```
function myFunction() {
    var inpObj = document.getElementById("id1");
    if (inpObj.checkValidity() == false) {
        document.getElementById("demo").innerHTML=
        inpObj.validationMessage; }
}
```

- **setCustomValidity(mensaje)**: Este método declara un error personalizado y el mensaje a mostrar solo si se intenta enviar el formulario. Si no se especifica ningún mensaje el error se anula.

```
<input type="text" id="nombre">
<input type="text" id="apellido">
function validación (nombre) {
    if( nombre.value=="" || apellido.value=="") {
        nombre.setCustomValidity("Error"); }}
```

Cada vez que el usuario envía un formulario se desencadena de forma automática un evento si se detecta un campo no válido. El evento de formulario es '**invalid**', y se puede dejar programado para lanzar un método que realice alguna acción frente al error.

```
formulario.addEventListener("invalid", validación, true);
```

```
// 'target' referencia el objeto que generó el evento.  
function validación (evento) {  
    var elemento = evento.target;  
    elemento.style.background="#FFDDDD";  
}
```

Considerando la necesidad de un sistema de validación dinámico, la API formularios incluye el objeto **ValidityState** que ofrece una serie de propiedades para indicar el estado de validez de un elemento del formulario. Las propiedades se muestran a continuación:

- valid: Esta propiedad devuelve 'true' si el valor del elemento es válido. Esta propiedad devuelve 'true' considerando todos los demás estados de validez (los que se explican a continuación).

```
var elemento = evento.target;  
if( elemento.validity.valid ) { ... }
```

- valueMissing: Esta propiedad devuelve 'true' cuando se ha declarado el atributo 'required' y el campo está vacío.
- typeMismatch: Devuelve 'true' si la sintaxis del campo no es la correcta. Por ejemplo, si en un campo del tipo 'mail' no se introduce una cuenta de correo electrónico.
- patternMismatch: Devuelve 'true' cuando el texto introducido no respeta el formato establecido por el atributo 'pattern'.
- tooLong: Devuelve 'true' cuando el texto introducido supera el valor establecido para el atributo 'maxlength'.
- rangeUnderflow: Devuelve 'true' cuando se ha declarado el atributo 'min' y el valor introducido es menor que el especificado por el atributo.
- rangeOverflow: Devuelve 'true' cuando se ha declarado el atributo 'max' y el valor introducido es mayor que el especificado por el atributo.
- stepMismatch: Devuelve 'true' cuando se ha declarado el atributo 'step' y el valor introducido no corresponde con el valor de los atributos 'min', 'max' y 'value'.
- customError: Devuelve 'true' cuando declaramos un error personalizado con el método '*setCustomValidity()*'.

API Geolocation

Esta API se utiliza para poder detectar la ubicación desde donde navegan los usuarios, es decir, utiliza la ubicación proporcionada por el navegador. La ubicación es mucho más exacta en un dispositivo móvil tipo SmartPhone ya que disponen de GPS incorporados. La captación de la ubicación solo es permitida si el usuario lo acepta previamente.

Hay diferentes formas de poder geolocalizar la posición:

- HTML5 con API Geolocation (es la que nos interesa)
- Localización a través de IP o cualquier técnica de redes.
- Bases de datos IP que contienen las direcciones y ubicaciones de determinadas IP's.
- Otros sistemas de geolocalización conocidos y no conocidos.

NOTA: Desde la versión 50 de Google Chrome solo es posible utilizar el sistema de geolocalización en contextos seguros como HTTPS. Fuera de este contexto las solicitudes de ubicación no funcionarán. El resto de los navegadores, por ahora, no son tan exigentes aunque podrían cambiar esta política en el futuro por cuestiones de privacidad.

La API Geolocation está estructurada básicamente en 3 métodos:

- **getCurrentPosition (ubicación, errores, configuración)** -> Este método forma parte del objeto geolocation que a su vez pertenece al objeto navigator. Solo es obligatorio pasarle el primer parámetro para que la función se pueda ejecutar.

Sintaxis: `navigator.geolocation.getCurrentPosition(funcion_ok, funcion_error, config)`

- **Primer parámetro:** Determina que función se ejecutará **si la ejecución ha tenido éxito**. En este caso la función invocada recibe un objeto '*Position*' que almacena los datos de la ubicación del usuario.
- **Segundo parámetro:** Es la función que se ejecutará **si no ha habido éxito** en la ejecución del método (falta conexión, bloqueo por parte del navegador, tiempo de espera excedido, o cualquier otro motivo). La función invocada recibe un objeto de tipo '*error*' que dispone de dos propiedades:
 - **code:** Un número que es una constante que representa el código de error.
 - 1 - Permiso denegado.
 - 2 - Ubicación no disponible.
 - 3 - Tiempo para detectar la ubicación excedido.

- **message**: Muestra un mensaje con el motivo del error generado. Es un texto que corresponde con el código de error generado.

Ejemplo: "Num error:" + **error.code** +"Msg:" + **error.message**
- **Tercer parámetro**: Es la configuración del método. Permite establecer diferentes valores para el acceso a los datos de ubicación:
 - **enableHighAccuracy**. Permite determinar la **exactitud de la ubicación**. No tiene ninguna utilidad para los Smartphone que dispongan de la función GPS. Los valores son true o false (valor predeterminado). Esta opción consume muchos recursos, con lo que se corre el riesgo de agotar rápidamente la batería del periférico.
 - **timeout**. Establece el **tiempo límite** para obtener la posición del usuario. Expresa en milisegundos el tiempo de espera de la aplicación antes de hacer la llamada a la función de gestión de errores. El valor predeterminado es 0.
 - **maximumAge**. Establece **cada cuanto tiempo queremos obtener la posición** del usuario incluso accediendo a la caché del equipo si fuese necesario. Determina en milisegundos el tiempo en que la posición actual está cacheada. Si el valor es 0 (valor predeterminado), la posición no se cacheará, sino que se renovará constantemente. También se prevé el valor infinito.

Ejemplo práctico para geolocalizar la ubicación usuario

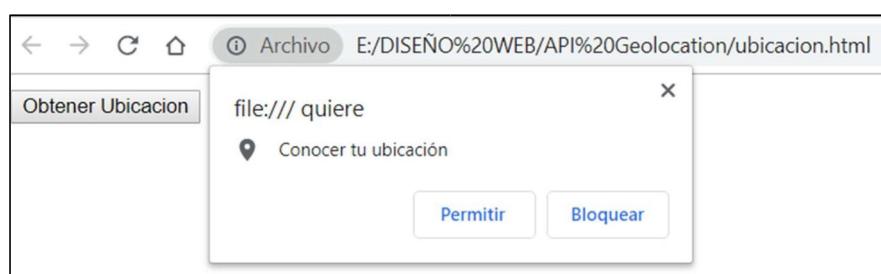
1. El método 'obtener()' invoca a **getCurrentPosition()** pasándole las funciones de ok y error.
2. El método '**mostrar_posicion()**' recibe objeto 'position' con datos de la ubicación.
3. El método '**errores()**' se ejecuta si no ha tenido éxito la ejecución de **getCurrentPosition()** y recibe como parámetro un objeto con los posibles errores de la incidencia.

```
function obtener(){  
    //Ejecuta metodo de geolocalizacion y llama funcion 'mostrar_posicion'  
    //Si se produce algun error al ejecutarse invoca funcion 'errores'  
    navigator.geolocation.getCurrentPosition(mostrar_posicion, errores);  
}
```

```
//Recibe como parametro el objeto Position con la informacion del metodo ejecutado.  
function mostrar_posicion(position){  
  
    var ubicacion=document.getElementById("ubicacion");  
  
    var resultado="Latitud: " + position.coords.latitude +"  
resultado += "Longitud: " + position.coords.longitude +"  
resultado += "Exactitud (accuracy): " + position.coords.accuracy +"  
  
    ubicacion.innerHTML=resultado;  
}
```

```
//Recibe el objeto que ha generado el error.  
function errores(error){  
  
    alert("Error al capturar la ubicacion" + error.code +" "+ error.message);  
}
```

Los navegadores muestran un mensaje de advertencia cuando se intenta acceder a la ubicación del usuario. Por este motivo es necesario dar permiso al navegador previamente y entonces la aplicación accederá a las coordenadas del usuario.



Este método al ser ejecutado devuelve un objeto '**Position**' siempre que se haya ejecutado con éxito. Solo retorna la información de geolocalización una sola vez. El objeto Position tiene dos propiedades que son las que nos facilitan toda la información:

- **Coords**: Contiene un conjunto de valores que determinan la posición.
 - *latitude*: Latitud como número decimal, siempre devuelto.
 - *longitude*: Longitud como número decimal, siempre devuelto
 - *altitude*: Altitud en metros sobre el nivel del mar.
 - *accuracy*: Exactitud o precisión en metros de la posición.
 - *latitudeAccuracy*: Exactitud en altura.
 - *heading*: Orientación en grados de la posición.
 - *speed*: Desplazamiento en metros/segundos si está disponible.
 - *timestamp*: Propiedad que indica el momento exacto en el que se determinó la posición.

Ejemplo práctico para configurar el tercer parámetro de getCurrentPosition()

```
function obtener(){  
    //Configuracion para el tercer parámetro de getCurrentPosition()  
    var params={enableHighAccuracy:true, timeout:10000, minimumAge:60000}  
  
    navigator.geolocation.getCurrentPosition(mostrar_posicion, errores, params);  
}
```

Seguimiento del desplazamiento

- **watchPosition (ubicación, errores, configuración)** -> Es un método asíncrono muy similar al método 'getCurrentPosition()' pero nos permite refrescar la ubicación del usuario cada cierto intervalo de tiempo.

Este método es muy útil si el usuario se está moviendo y necesitamos conocer las diferentes ubicaciones que genera. La propiedad '**maximumAge**' es la que especifica cada cuánto tiempo se debe recuperar la ubicación del dispositivo. Además en este método no se tiene en cuenta la memoria caché igual que en el método anterior. Esta función no se aplica a los Smartphone.

```
navigator.geolocation.watchPosition(mostrar_posicion, errores, params);
```

- **clearWatch(id)** -> Resetea el método **watchPosition()** para que no nos devuelva la posición cada cierto tiempo. El funcionamiento es exactamente el mismo que los métodos **setInterval()** y **clearInterval()**, es decir, que resetea el objeto para que finalice su ejecución.

Códigos de error más frecuentes.

Cuando la geolocalización no ha sido positiva se generan una serie de errores (errores de localización) que se pueden capturar. Algunos de estos errores ya se han comentado ('*error.code*', '*error.message*'), pero existen otros tipos de errores definidos como propiedades globales del método que son valores booleanos que cambian su valor al coincidir con el error.

```
function showError(error) {
    switch(error.code) {
        case error.PERMISSION_DENIED:
            x.innerHTML = "El usuario no permitió acceder a su ubicación."
            break;
        case error.POSITION_UNAVAILABLE:
            x.innerHTML = "Información solicitada no está disponible"
            break;
        case error.TIMEOUT:
            x.innerHTML = "La consulta produjo un timeout"
            break;
        case error.UNKNOWN_ERROR:
            x.innerHTML = "Error desconocido"
            break;
    }
}
```

API Web Storage

La API permite almacenar información **de forma temporal o permanente en el navegador**. Para ello se reserva un espacio en el disco duro que pertenecerá exclusivamente al navegador. Concretamente cada dominio web tendrá su espacio exclusivo y reservado en memoria para almacenar datos, es decir, en ningún momento comparten espacio entre dominios.

Las aplicaciones web y los navegadores siempre han almacenado la información a través de dos formas:

1. Cookies: Permite almacenar texto en ficheros en los archivos temporales de internet. No admiten mucha información y llegan a un máximo de 5Mb.
2. Servidor web: Es el sistema tradicional y más óptimo. Almacena la información en bases de datos de forma permanente y son accesibles desde las aplicaciones web.



Con la llegada de nuevos dispositivos (Smartphones y Tablets) estas dos formas de almacenar información no eran las más adecuadas. De esta necesidad surgió la API Web Storage que nos da diferentes posibilidades de almacenar información.

En JS para comprobar si el navegador acepta la API Web Storage y se puede utilizar hay que preguntarle mediante código JS con el método 'typeof'. Hoy en día muy improbable que un navegador de los populares no acepte esta tecnología pero estas líneas de código se siguen introduciendo muchas veces.

```

if (typeof(Storage) !== "undefined") {
    resultado = "Storage Aceptado";
} else { resultado = "Storage NO aceptado"; }
  
```

Web Storage guarda la información utilizando uno de los siguientes sistemas:

- **sessionStorage**: Es almacenamiento **temporal**, es decir, solo está disponible mientras la sesión esté activa. La sesión activa hace referencia a la pestaña del navegador y solo cuando esta se cierra la información almacenada se pierde.
- **localStorage**: El almacenamiento es **permanente**. La información está disponible aun cerrando el navegador. Es más seguro y cada dominio dispone de su propio localStorage.

Para **almacenar información** en alguno de los dos sistemas dispone del método **setItem()** y para recuperar la información una vez ha sido almacenada mediante **getItem()**.

- **setItem():** Añade información para alguno de los dos sistemas anteriores. La información se pasa normalmente como String o número ya que un objeto tipo JSON daría error. En estos casos el objeto JSON se debe convertir a String antes de ser pasado al método 'setItem()' como parámetro mediante el método **stringify()**.

```
// Almacenando una variable y un valor tipo texto.  
localStorage.setItem("pelicula", "Spiderman");  
  
// Almacena un objeto JSON que previamente tiene que ser  
// convertido a string mediante el método stringify().  
  
var peliculas = [{titulo: "Superman", year: 2015, pais: "EUA"},  
                 {titulo: "Batman", year: 2016, pais: "Canada"},  
                 {titulo: "Celda211", year: 2018, pais: "Spain"}];  
  
// Convierte el objeto a String  
localStorage.setItem("pelis", JSON.stringify(peliculas));
```

- **getItem():** Recupera información de un campo que haya sido almacenado. El valor recuperado se puede almacenar en una variable. Si lo que se desea recuperar es un objeto JSON que previamente había sido convertido a String, entonces hay que volver a convertir el String a objeto JSON con el método **parse()**.

```
// Recupera un valor almacenado en formato texto.  
var pelicula = localStorage.getItem("pelicula");  
  
// Convierte el String recuperado de localStorage a objeto con el  
// método parse() y lo almacena en una variable completamente usable  
var peliculas = JSON.parse(localStorage.getItem("pelis"));
```

- **removeItem():** Elimina el objeto localStorage o sessionStorage que se especifique.

```
localStorage.removeItem("pelis");
```

- **clear():** Vacía todos los elementos localStorage o sessionStorage que hubiera guardados.

```
localStorage.clear();
```

La variable de tipo **sessionStorage** NO es un array de variables, es decir, cada variable generada con sessionStorage es independiente del resto.

Es posible generar tantas variables de sesión como se desee pero es recomendable utilizar el menor número posible para no hacer más vulnerable nuestra aplicación. Además estas variables que se pueden considerar "**superglobales**" mientras el navegador está abierto serán accesibles, por tanto, el nombre deberá contener una combinación de letras y números para dificultar accesos no deseados.

Desde los navegadores es posible realizar la supervisión de las variables mediante el depurador de elementos (pestaña 'Application'). La ventana muestra información tanto de sessionStorage como localStorage. La siguiente imagen muestra una captura de la opción del depurador.

The screenshot shows the Chrome DevTools Application tab interface. On the left, there's a sidebar with 'Application' and 'Storage' sections. Under 'Storage', 'Local Storage' is expanded, showing an item for 'file://'. The 'localStorage' section contains one entry: 'contenido' with the value 'Código y café es una gran combinación'. A red box highlights the 'LocalStorage' section in the sidebar, and another red box highlights the entry in the main table.

Key	Value
contenido	Código y café es una gran combinación

La API Web Storage también acepta la nomenclatura del punto tanto para asignar como para recuperar valores de Storage.

```
localStorage.pelicula = "Spiderman";  
var pelicula = localStorage.pelicula;
```

COOKIES

Las cookies son datos de la página modificados mediante JavaScript que **pueden guardarse en el ordenador del usuario**, de manera que éste pueda recuperarlos.

Por ejemplo, si mediante JavaScript hemos dado a la página un aspecto o configuración distinta a la inicial, o hemos escrito en la página una serie de datos, al cerrar la página estos datos se perderán si no podemos guardarlos. Las **cookies permiten guardar estos datos**, de manera que puedan recuperarse, cuando se carga la página de nuevo o al accionar un evento.

Por otra parte las cookies en JavaScript permiten **leer los datos almacenados desde otra página distinta** de la que se han cargado. Por tanto podemos pedir datos al usuario en una página, y mostrarlos en otra distinta. Se guardarían en lo denominado variables superglobales.

Las cookies tienen ciertas limitaciones, así **un usuario no puede almacenar más de 300 cookies** y el tamaño máximo de cada cookie es de **4000 bytes**.

Las cookies **deben estar habilitadas en el navegador** para poder almacenar información.

Propiedad 'document.cookie'

Esta propiedad en JS nos permite almacenar las cookies y volver a leerlas. Si no se especifica nada la cookie se elimina al cerrar el navegador. Por este motivo es posible asignarle una **fecha de expiración**. Para almacenar una cookie se realizará con la sintaxis:

```
document.cookie = "nombre=valor; expiración (opcional); ruta (opcional);  
document.cookie = "username=John";  
document.cookie = "username=John; expires=Thu, 18 Dec 2013 12:00:00 UTC";
```

Para **leer una cookie** se accederá a **document.cookie** y se guardarán todas las cookies en una variable que deberá leerse troceándola para acceder a la cookie que nos interesa.

```
var x = document.cookie;
```

Para **modificar una cookie** se deberá volver asignar el mismo nombre de cookie con un valor diferente. En este caso se sobrescribirá la cookie.

Para **eliminar una cookie** o uno de sus nombres se realizará modificando la fecha de expiración a un valor anterior de la fecha actual. Algunos navegadores no permiten eliminar una cookie si no se especifica la ruta de acceso.

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;  
path=/";
```

Las acciones para leer o almacenar una cookie se suelen realizar mediante funciones en JS.

- La función de lectura recupera todas las cookies, las trocea a partir del punto y coma (;) y busca cookie que conviene al usuario.
- La función de asignación se le pasan los 3 parámetros que son necesarios, nombre, valor y expiración. A continuación se muestran las 3 funciones más habituales para trabajar con cookies.
- La función de chequeo comprueba si una cookie tiene un nombre en concreto con su correspondiente valor. Esta acción se utiliza para saber si el usuario es la primera vez que accede a la web o no.

```
// Crea cookie con una fecha de expiración.  
function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime() + (exdays * 24 * 60 * 60 * 1000));  
    var expires = "expires="+ d.toUTCString();  
    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";  
}  
  
// Recupera todas las cookies, trocea el texto a través del punto y coma, y comprueba si el nombre coincide con el que estamos buscando. Si es así devuelve la segunda parte de la cookie, es decir después del carácter igual  
function getCookie(cname) {  
    var name = cname + "=";  
    var ca = document.cookie.split(''); // Trocea cookie y guarda en array  
    for(var i = 0; i < ca.length; i++) {  
        var c = ca[i];  
        while (c.charAt(0) == ' ') {  
            c = c.substring(1);  
        }  
        if (c.indexOf(name) == 0) { // Busca le nombre de la cookie  
            return c.substring(name.length, c.length);  
        }  
    }  
    return "";  
}
```

```
// Comprueba si un nombre de cookie se encuentra almacenado
function checkCookie() {
    var user = getCookie("username"); // Busca 'username'
    if (user != "") {
        alert("Welcome again " + user); // Si está llena muestra mensaje
    } else {
        user = prompt("Please enter your name:", "");
        if (user != "" && user != null) {
            setCookie("username", user, 365);
        }
    }
}
```

Persistencia de cookies

Cada vez que el cliente pide una página web al servidor, el servidor retorna un HTML que el navegador tiene que interpretar y realizar una petición por cada recurso contenido en ella (si la página tiene 3 fotos, se harán 4 peticiones contando el mismo HTML). Lo curioso es que el servidor por defecto no tiene ni idea de qué hemos pedido anteriormente. Así que si le pido una foto el servidor ni sabe ni le importa que se lo esté pidiendo porque anteriormente le he pedido la web con la foto.

Esta característica hace que los servidores web de por sí NO recuerden la información de las interacciones que hemos hecho con ellos. Entonces, si entramos en una página web con un login, *¿cómo es posible que al navegar por ese sitio web nos recuerde quiénes somos?*

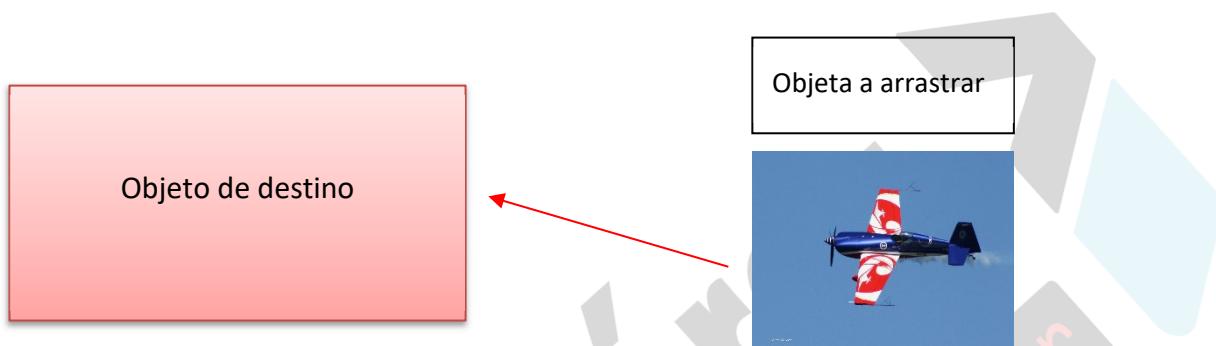
Para conseguir que nos recuerde de alguna forma deberemos conseguir que algún dato sea “permanente”. Y que además esos datos sean independientes para cada navegador que hace consultas e incluso en cada navegador independientes de cada sitio web (no quiero que otro navegador o sitio web acceda a mis datos). Quiero, en definitiva, información vinculada a mi “sesión”, es decir, vinculada al navegador y al sitio web.

Cuando navegamos en **modo “incognito”** estaremos navegado como si fuéramos otro navegador y toda esa información vinculada a la “sesión” **será destruida** al cerrar el navegador. A continuación veremos las técnicas más ampliamente soportadas para la persistencia de datos: las “cookies”.

API Drag & Drop

Esta API se utiliza para poder arrastrar elementos de un sitio a otro de la web, como por ejemplo, generar un puzzle, modificar un menú de navegación o cambiar una interfaz de usuario.

Para utilizar esta API hay que tener claros una serie de conceptos asociados. Hay que identificar el objeto que será arrastrado, la zona de destino, y los métodos relacionados con las dos zonas anteriores.



Métodos relacionados

- **dragenter:** Se lanza cuando el elemento entra en el área de destino.
- **dragover:** Desencadena la acción cuando el ratón se mueve sobre el área de destino.
- **drop:** Cuando el elemento es soltado en el área de destino.
- **dragleave:** Cuando el elemento arrastrado sale de la zona de destino

Métodos relacionados

- **dragstart:** Desencadena la acción cuando comienza a arrastrar.
- **drag:** Desencadena la acción durante la operación de arrastre.
- **dragend:** Desencadena la acción cuando ha terminado de arrastrar.

Para finalizar la configuración será necesario establecer en el elemento arrastrado, que tipo de información se desea compartir. En el elemento de destino se debe especificar que información ha sido compartida. Para ello se utiliza el objeto **DataTransfer** y dos métodos:

- **setData:** Declara que datos serán transferidos
- **getData:** Declara que datos serán capturados.

En el siguiente código se muestran los tres eventos del elemento a arrastrar y como lanzan un mensaje por la consola del navegador. En este caso los eventos '**dragstart**' y '**dragend**' se lanzarán una vez, mientras que el evento '**drag**' se irá ejecutando cada vez que estemos arrastrando el objeto, de ahí el resultado que se puede ver en la siguiente imagen.

```

var imagen = document.getElementById("imagen"); //Capturamos elemento

// Ponemos a la escucha los 3 eventos del objeto de arrastre
imagen.addEventListener("dragstart",function(){
  console.log("Empezó el drag"), false); //Inicio drag

imagen.addEventListener("drag",function(){
  console.log("Esta arrastrandose con el drag"), false); //Durante

imagen.addEventListener("dragend",function(){
  console.log("Finalizó el drag"), false); //Finalizado drag
  
```



A continuación, se debe configurar el elemento de destino (zona de destino). Para empezar los eventos '**'dragenter'** y '**'dragover'**' se deben resetear para que no ejecuten ninguna acción al ser lanzados. Esto se realizará con el evento '**'preventDefault()'**' que elimina por defecto el comportamiento del evento. Algunos eventos ejecutan acciones de forma implícita como por ejemplo un '**'submit'**' en un formulario y para que no actúen se deben resetear con '**'preventDefault()'**'.

El siguiente código muestra los '**'listeners'**' para los diferentes eventos posibles en un elemento origen y otro de destino. En ambos casos lo mas importante antes de configurar los escuchadores para cada evento es capturar la referencia al objeto.

```

// EXPLICACION PARA EL ELEMENTO ARRASTRE Y ZONA DESCARGA
var imagen = document.getElementById("imagen"); //Capturamos elemento origen
imagen.addEventListener("dragstart",comienzoDrag, false); //Inicio drag
imagen.addEventListener("drag", function(){ console.log("Arrastrando el drag")}, false); //Durante
imagen.addEventListener("dragend",terminado, false); //Finalizado drag

var destino = document.getElementById("zonadestino"); //Capturamos elemento destino
//Detiene la acción del evento por defecto para que no ejecute nada ni al entrar ni al posicionarse.
destino.addEventListener("dragenter", entrandoDestino, false);
destino.addEventListener("dragleave", saliendoDestino, false);
destino.addEventListener("dragover", function(e){ e.preventDefault();}, false);
destino.addEventListener("drop", soltarElemento, false);
  
```

Si hubiese un error en alguno de los listeners o no se ejecutase correctamente automáticamente lo mas probable es que todo el Drag & drop dejase de funcionar ya que los diferentes eventos de la API están directamente relacionados y se ejecutan muchos de ellos de forma secuencial.

Para poder compartir la información entre la zona de origen y de destino será necesario decir en la zona de origen que elemento/s deseamos compartir (`setData`) y en la zona de destino recuperar esa información del contenedor de información (`getData`).

El contenedor de información es el objeto **dataTransfer** y actúa como un elemento contendor donde se "dejan cosas" para que posteriormente otro elemento "recoja las cosas". Drag & drop utiliza este sistema para pasarse la información entre los elementos.

La siguiente imagen muestra como se informa justo al ejecutarse el evento 'dragstart' y como se recupera al soltar el elemento 'drop' la información de un elemento **dataTransfer**. Cabe recordar que este objeto se utiliza para almacenar o recuperar información.

```

function comienzoDrag(e){
    //Extrae el atributo del objeto imagen para construir un elemento
    var codigo = "<img src='" + imagen.getAttribute("src") + "' height='250px'"
    //Establecer información a compartir en dataTransfer
    e.dataTransfer.setData("text", codigo);
}

function soltarElemento(e){
    e.preventDefault();
    // Recupera la información de dataTransfer
    valor=e.dataTransfer.getData("text"); //Captura el texto del dataTransfer
    destino.innerHTML = valor; //Inserta la imagen en la zona de destino.
}

```

Comparte información y la introduce en el contenedor

Recupera la información guardada en el contenedor

Posteriormente se debe acabar de ajustar todos los eventos como por ejemplo eliminar la imagen del inicio para que no salga duplicada, o determinar si un elemento puede dejarse en una zona de destino en concreto.

Además de poder arrastrar elementos por la web, es posible desde el explorador arrastrar imágenes. Para ello se debe utilizar el objeto **dataTransfer** con la propiedad '**files**'.

```

function soltado(e){
    e.preventDefault();

    //files genera un array y en cada posicion almacena las propiedades del archivo
    var archivos = e.dataTransfer.files;

    var listado = ""; //Es necesario inicializarlo sino puede dar error mas adelante

    listado=archivos[0].size; //Puede ser 'name', 'size', 'type', 'lastmodified'.
    destino.innerHTML=listado; //Inserta la propiedad

```

archivos: FileList
 length: 1
 ▾ 0: File
 name: "CCD34000"
 lastModified: 1504006244000
 ▷ lastModifiedDate: Tue Aug 29 2017 13:30:44 GMT+0200
 webkitRelativePath: ""
 size: 0
 type: ""

API IndexedDB

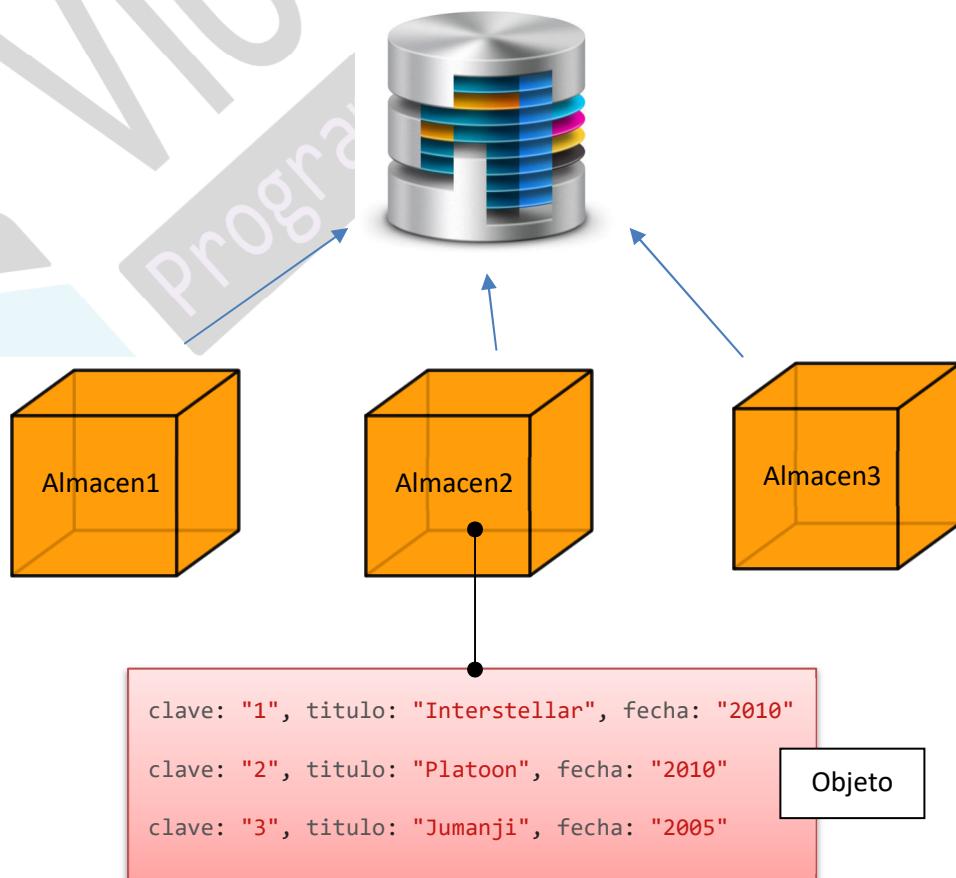
Esta API se utiliza para **almacenar un gran volumen de información** en la maquina local y las opciones de Web Storage no es suficiente y se quedan cortos. Por ejemplo, podemos tener una aplicación que recibe mucha información de un usuario y posteriormente esta información debe ser enviada a un servidor.

La estructura de un sistema creado con IndexedDB no tiene nada que ver con las formas más habituales de creación de bases de datos (Access, MariaDB, Oracle, etc..), es decir, no dispone de tablas, campos y registros. Todo y con eso, aun no teniendo la misma nomenclatura que otras bases de datos el sistema de distribución de los elementos es similar.

- Tabla -> Almacén de datos.
- Registros -> Objetos.
- Campos -> Valores.
- Clave primaria -> Campo clave

Una base de datos en IndexedDB está formada por **almacenes de datos** (tablas) y en su interior se introducen **objetos** (registros). Los objetos de cada almacén no tienen por qué tener siempre la misma estructura, es decir, de un objeto podemos almacenar 3 campos mientras que de otro solo 2. Por tanto, no mantiene la misma lógica que una base de datos convencional.

Cada navegador almacena las bases de datos creadas con IndexedDB en una ubicación diferente.



Pasos para la creación y manipulación de la base de datos

1. Crear la base de datos en 'indexedDB'.

- Atributo '**indexedDB**'. Es un objeto que se puede acceder directamente porque depende del objeto 'Window'.
- Método '**open()**'. Recibe dos parámetros aunque solo es necesario el primero que hace referencia al nombre de la base de datos.

```
// CREACION DE LA BASE DE DATOS
var peticion = indexedDB.open('Datos'); //Abrir base de datos

//Carga valores de la bd si se abrió correctamente.
peticion.onsuccess = function(e){
    bd = e.target.result; //Almacenamos la base datos en la variable bd.
}
```

2. Crear almacenes de datos.

- Método '**createObjectStore()**'. Este método dispone de dos parámetros, el primero hace referencia al nombre del almacén de datos, y el segundo determinará cual será el campo clave de los que se hayan pasado.

```
//Crea el almacen de datos SOLO si no ha sido creado previamente
peticion.onupgradeneeded = function(e){

    bd = e.target.result; //Almacenamos la base datos en la variable bd.
    // Creacion del nombre del almacen de datos y su campo clave
    bd.createObjectStore("clientes", {keyPath:"clave"})
}
```

3. Realizar transacción (agregar, borrar, modificar elementos)

- Método '**transaction()**'. El método recibe dos parámetros, el primero el almacén de datos con dobles comillas y entre corchetes, y el segundo el tipo de acción que se realizará sobre el almacén ('read', 'write', 'readwrite').

```
//Es necesario crear una transaccion con la base de datos.
//El nombre del almacen va entre corchetes y la acción puede ser de 3 tipos
//(lectura, escritura, o lectura-escritura)
var transaccion = bd.transaction(["clientes"],"readwrite");

//Almacena la transacción que acabamos de realizar, de esta forma cualquier acción
//sobre el almacen de datos se porá referencia desde esta variable.
var almacen = transaccion.objectStore("clientes");
```

4. Agregar objetos

- a. Método **'add()'**. Se debe introducir la información de los campos previamente capturados de la web entre llaves. En cada pareja de valores, el primer valor hace referencia al valor que tendrá dentro de la base de datos, mientras que el segundo valor será el que se haya pasado a través de la variable.

```
//El método agregar nos pide que información queremos almacenar.y el nombre que
tendrán dentro del almacen de datos.
var agregar = almacen.add({ clave:clave, titulo:titulo, fecha:fecha });
```

Cuando añadimos los valores a la base de datos de IndexedDB la consola del navegador (F12) muestra la información contenida dentro de la base de datos. Esta información se encuentra en la pestaña '**Application**'. Desde la pestaña se puede visualizar el nombre de la base de datos, el objeto creado dentro de la base de datos, y los diferentes elementos introducidos.

Si se añade algún dato **en tiempo de ejecución** será necesario actualizar la base de datos para que se puedan ver desde el depurador.

The diagram illustrates the IndexedDB storage structure in the browser developer tools Application tab. It shows four boxes with labels:

- Ventana de diferentes formas de almacenamiento de contenido en la web.
- Actualizar base de datos
- Campo clave
- Registro de datos almacenados

Red arrows point from these labels to specific parts of the developer tools interface:

- A red box surrounds the 'Storage' section in the left sidebar, which contains 'Local Storage', 'Session Storage', and 'IndexedDB'. A red arrow points from the 'Ventana de diferentes formas de almacenamiento de contenido en la web.' box to this section.
- A red box surrounds the 'Key (Key path: "clave")' input field in the main table header. A red arrow points from the 'Actualizar base de datos' box to this field.
- A red box surrounds the 'Value' column in the table, which displays three objects representing database entries. A red arrow points from the 'Campo clave' box to this column.
- A red box surrounds the 'Total entries: 3' text at the bottom of the table. A red arrow points from the 'Registro de datos almacenados' box to this text.

The developer tools interface shows the following data in the table:

#	Key (Key path: "clave")	Value
0	"1"	<pre>{clave: "1", titulo: "Interstellar", fecha: "2010"} clave: "1" titulo: "Interstellar" fecha: "2010"</pre>
1	"2"	<pre>{clave: "2", titulo: "Platoon", fecha: "2010"} clave: "2" titulo: "Platoon" fecha: "2010"</pre>
2	"3"	<pre>{clave: "3", titulo: "Jumanji", fecha: "2005"} clave: "3" titulo: "Jumanji" fecha: "2005"</pre>

A box labeled 'Total registros base de datos.' is placed over the value of entry 2.

5. Mostrar información almacenada. Será necesario '**abrir un cursor**' que es como generar una tabla virtual que contiene la información.

- a. Método '**opencursor()**' para recorrer los datos por la tabla virtual.

```
//Abre cursor para poder recorrer todo el contenido del almacen.  
var cursor = almacen.openCursor();  
  
//Si el cursor se ha creado correctamente sobre almacen de datos pintara valores  
cursor.addEventListener("success", mostrardatos, false);
```

```
function mostrardatos(e){  
  
    //Contiene la información del almacen de la base de datos  
    var cursor = e.target.result;  
  
    //Mostramos información en pantalla.  
    if(cursor){  
        zonadatos.innerHTML+="

" + cursor.value.clave + " - " + cursor.value.titulo  
        + " - " + cursor.value.fecha + "

";  
  
        cursor.continue(); //Avanzamos el cursor  
    }  
}
```

AJAX (Asynchronous JavaScript and XML)

AJAX es una tecnología que permite realizar **consultas de forma asíncrona a un servidor web**. Hasta ahora solo conocíamos dos técnicas para poder realizar peticiones al servidor:

- A través de un link.
- Mediante un formulario HTML

Ambas formas cuando obtienen la respuesta del servidor borran el contenido actual de la web y es sustituido por la respuesta obtenida. El problema de recargar por completo la página se produce cuando deseamos actualizar exclusivamente una sola zona de nuestra página sin modificar el resto de la web.

Este problema de actualizar toda la web provoca una peor experiencia de usuario (los campos del formulario desaparecen) y un tiempo de carga superior de la página al tener que volver a pedir todos los recursos al servidor.

AJAX pretende solucionar este problema y las características principales de la tecnología son:

- a) Las peticiones **se realizan con Javascript**, por tanto podemos programar diferentes peticiones con diferentes parámetros.
- b) Las **peticiones son asíncronas**, es decir, no sabemos cuánto tiempo tardará el servidor en contestarnos. La respuesta suele ir muy rápido en la actualidad pero según el servidor donde se realicen las peticiones puede tardar más tiempo.
- c) Al realizar una petición AJAX en Javascript **el resto del código de la página se sigue ejecutando** mientras se espera la respuesta.
- d) Cuando recibamos una respuesta **se ejecutará la función que nosotros hayamos definido** para gestionar la respuesta. Esta función extrae los datos de la respuesta y realiza las modificaciones sobre el DOM si fuese necesario.
- e) AJAX fue pensado inicialmente para trabajar con archivos XML con la información retornada por el servidor pero actualmente se trabaja sobre **archivos estructurados del tipo JSON**.
- f) Las funciones para realizar consultas AJAX las encontramos en dos APIs de Javascript:
 - a. XMLHttpRequest.
 - b. Fetch.

Para trabajar con AJAX es recomendable conocer los aspectos básicos y necesarios de envío y recepción de información a servidores web mediante .PHP como por ejemplo la forma de receptionar el archivo .php los datos recibidos por un formulario o una petición AJAX, como tratarlos y como devolver un resultado nuevamente al cliente.

API XMLHttpRequest

Esta es la primera API que ofreció un conjunto de métodos para realizar operaciones AJAX. Para utilizarla es necesario **crear un objeto con el constructor XMLHttpRequest**. Cada uno de estos objetos representarán una petición AJAX, y será a través de ellos que configuraremos, enviaremos y recibiremos dicha petición.

Para realizar una petición AJAX hay que seguir los siguientes pasos:

1. Declarar un objeto XMLHttpRequest.

```
let xmlhttp = new XMLHttpRequest();
```

2. Configurar la petición con su método 'open()' y sus tres parámetros.

- a) El método de la petición (GET o POST)
- b) La URL de la petición con sus parámetros si fuese modo GET. Si el método es POST los parámetros se especifican en el envío 'send()'.
- c) Un valor booleano 'true' o 'false' si deseamos que sea la petición asíncrona o no (por defecto, y recomendado 'true').

```
xmlHttp.open("GET", "pagina_respuesta.php?edad=30&sexo=h,true");
```

3. Establecer con el método 'setRequestheader()' el header de la petición si fuese necesario.

```
xmlHttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
```

4. Configurar la función que se va a encargar de controlar cada cambio de estado en la petición. Cualquier objeto XMLHttpRequest tiene una propiedad 'readyState' que nos indica con un valor numérico de 0 a 4 el estado de la comunicación:

- a) 0: Aun no se ha enviado petición
- b) 1: Se ha enviado la petición.
- c) 2: Se han recibido las cabeceras de la respuesta.
- d) 3: Se está cargando parte del contenido de la respuesta.
- e) 4: Se ha terminado de cargar toda la respuesta.

El evento '**onreadystatechange()**' es el evento que se lanza cuando se produce un cambio en la propiedad readyState.

```
xmlHttp.onreadystatechange = function(){  
    if(xmlHttp.readyState == 4){ alert("Respuesta recibida") }  
}
```

Pero que hayamos tenido respuesta del servidor no significa que esta haya sido correcta y con el recurso solicitado. Por ello, además de conocer el estado de la comunicación será necesario conocer el estado de la respuesta. Para ello será necesario acceder a la propiedad '**status**' del objeto XMLHttpRequest devolviéndonos dos posibles valores:

- 200: Se ha encontrado y obtenido el recurso pedido.
- 404: Indica que la URL del recurso se ha encontrado.

```
xmlHttp.onreadystatechange = function(){
    if(xmlHttp.readyState == 4 && xmlHttp.state == 200){
        alert("Respuesta recibida y petición correcta") }
}
```

5. Si ya hemos configurado la petición y controlado sus estados será necesario gestionar la respuesta recibida por el servidor. Lo más frecuente es utilizar uno de los 3 posibles modos de respuesta:

- a) **Texto plano**: Es la forma más sencilla. Los objetos XMLHttpRequest tiene una propiedad denominada '**responseText**' que contiene el texto retornado por el servidor. En este caso solo se precisa de una acción.

```
let textoRespuesta = xmlhttp.responseText; // 1-Recibe texto
```

- b) **Formato JSON**: Es el formato más utilizado con diferencia. Si el dato recibido desde el servidor es un texto que representa un objeto JSON se deberá convertir a objeto propiamente JSON con el método '**parse()**', es decir, hay que hacer dos acciones.

```
let textoRespuesta = xmlhttp.responseText; // 1-Recibe texto
let objetoJSON = JSON.parse(textoRespuesta); // 2-Convierte
```

- c) **Formato XML**: Es el formato original de AJAX por eso los objetos XMLHttpRequest disponen de la propiedad '**responseXML**' que retorna un tipo de documento con su DOM y que puede ser recorrido por los métodos del DOM excepto alguno como por ejemplo '**getElementById()**'.

```
let domXML = xmlhttp.responseXML; // Recupera XML
```

6. Por último solo queda enviar la petición con el método '**send()**'. Si la petición AJAX se ha configurado con POST, el método puede recibir el listado de parámetros al enviar al servidor.

```
xmlHttp.send(); // Envío petición AJAX con GET
xmlHttp.send('name=pep&age=30'); // Envío AJAX con POST
```

Ejemplo 1: Envía los valores de dos inputs a un servidor a través del **método GET**. El servidor los suma y devuelve el resultado. En este caso la respuesta recibida es en formato JSON.

Documento HTML

```
<head>
  <title>Ejemplo AJAX</title>
<script src="cliente.js" defer></script>
</head>
<body>
  <input type="number" id="num1" />
  <input type="number" id="num2" />
  <button onclick="sendAJAX()">
    SEND AJAX
  </button>
  <div id="respuesta"></div>
</body>
```

Documento 'respuesta.php'

```
<?php
$numero1 = $_GET["n1"];
$numero2 = $_GET["n2"];

$resp=$numero1+$numero2;
echo '{"resp":"' . $resp . '"}';
```

Documento 'cliente.js'

```
function sendAJAX() {
  //obtenemos los valores a enviar al servidor
  let num1 = document.getElementById("num1").value;
  let num2 = document.getElementById("num2").value;

  //Declarar el objeto XMLHttpRequest
  let xmlhttp = new XMLHttpRequest();
  //Configurar la petición pasando los parámetros que queramos
  xmlhttp.open("GET", "respuesta.php?n1=" + num1 + "&n2=" + num2, true);
  //Establecer el HEADER de la petición
  xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

  //asignar una función encargada de gestionar cada cambio
  xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4) {
      if (xmlhttp.status == 200) {
        //obtenemos la respuesta y la parseamos a JSON
        let textoRespuesta = xmlhttp.responseText;
        let objetoJSON = JSON.parse(textoRespuesta);
        let respuesta = objetoJSON.resp;
        //mostramos la respuesta en el HTML
        document.getElementById("respuesta").innerHTML = respuesta;
      }
    }
  }
  xmlhttp.send(); //enviar la petición
}
```

Los parámetros con GET se pasan en el método **open()**

Ejemplo 2: Envía los valores de dos inputs a un servidor mediante **POST** y añadiéndolos como parámetros a la función '**send()**'. El servidor devuelve el resultado en formato **XML**.

Documento HTML

```
<head>
  <title>Ejemplo AJAX</title>
  <script src="cliente.js" defer></script>
</head>
<body>
  <input type="number" id="num1" />
  <input type="number" id="num2" />
  <button onclick="sendAJAX()">
    SEND AJAX
  </button>
  <div id="respuesta"></div>
</body>
```

Documento 'respuesta.php'

```
<?php
$numero1 = $_POST["n1"];
$numero2 = $_POST["n2"];
$resul=$numero1+$numero2;
//Tipo de archivo a retornar:
@header("Content-type: text/xml");
$xml='<?xml version="1.0" encoding="utf-8"?>';
$xml.= '<respuesta>';
$xml.= '<resul>' . $resul . '</resul>';
$xml.= '</respuesta>';
echo $xml;
```

Documento 'cliente.js'

```
function sendAJAX() {
  //obtenemos los valores a enviar al servidor
  let num1 = document.getElementById("num1").value;
  let num2 = document.getElementById("num2").value;

  //Declarar el objeto XMLHttpRequest
  let xmlhttp = new XMLHttpRequest();
  xmlhttp.open("POST", "respuesta.php", true);
  //Establecer el HEADER de la petición
  xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

  //asignar una función encargada de gestionar cada cambio
  xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4) {
      console.log("respuesta recibida!");
      if (xmlhttp.status == 200) {
        //obtenemos la respuesta y extraemos el XML
        let respuestaXML = xmlhttp.responseXML;
        //extraemos la respuesta moviéndonos por el DOM
        let resultado = respuestaXML.firstChild.firstChild.innerHTML;
        //mostramos la respuesta en el HTML
        document.getElementById("respuesta")
          .innerHTML = resultado;
      }
    }
  }
  xmlhttp.send("n1=" + num1 + "&n2=" + num2); //enviar la petición con parámetros
}
```

Los parámetros con POST se pasan con el método **send()**.

API Fetch

El método **fetch()** nos permite realizar **peticiones AJAX de forma más simple** sin tener que recurrir al complicado sistema de JavaScript clásico (XMLHttpRequest). Se puede decir que es una evolución del sistema anterior que tiene ciertas limitaciones en algunos aspectos. Con el método '**fetch()**' el asincronismo se consigue porque retornará un objeto del tipo **Promise** con mayor capacidad de control y mayor flexibilidad a la hora de realizar llamadas al servidor.

Mas técnicamente cabe comentar que funciona tanto en '**window**' como en '**worker**'. Está disponible en Node.JS por lo que es posible utilizarlo de forma isomórfica, es decir, tanto en el cliente como el servidor. Además es soportado por todos los navegadores.

Nomenclaturas básicas

Comunicación Asíncrona: Se produce cuando después que el emisor envíe una petición al receptor, el emisor puede seguir trabajando **sin tener que quedarse bloqueado** esperando la respuesta del receptor.

AJAX (Asynchronous JavaScript and XML): Son un conjunto de tecnologías que permiten hacer llamadas o peticiones a un servidor desde el navegador **sin necesidad de refrescar la página**. En muchos casos actualizar una página web entera no es óptimo para su funcionamiento y solo es necesario refrescar parte de ella. Bajo este contexto AJAX tiene sentido y es una práctica común realizar peticiones AJAX entre navegador y servidor.

SPA (Single Page Application): Son aplicaciones web de página única que recargan exclusivamente la zona de la web donde los datos deben ser renovados. El resto de la página (códigos HTML, JS y CSS) se cargan una única vez liberando de forma considerable al servidor. Es un formato muy extendido en la actualidad por la gran optimización de recursos.

REST (API Rest): Un servicio REST es cualquier interfaz entre sistemas que utiliza el protocolo HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON. Dicho de otra manera, es la forma de acceder a datos que se encuentran ubicados en la nube utilizando el protocolo web HTTP en forma de archivo JSON o XML.

El objeto **Promise** devuelto por **fetch()** dispone de dos métodos:

'then()' que recibe como parámetro una función que se ejecutará cuando se obtenga respuesta del servidor. Cuando los datos se reciben correctamente la propiedad 'status' será igual a 200 y cuando no localice el recurso el valor devuelto será 404.

'catch()' que se ejecutará cuando se produzca un error de red básicamente. Hay que tener muy presente que si se devuelve un http correspondiente a un error no se ejecutará este método sino el método '**then()**' porque ha recibido datos.

Para obtener el body o cuerpo del mensaje devuelto por el servidor será necesario **obtener una segunda promesa** por medio de los métodos del objeto 'Response' devuelto.

Para configurar una petición AJAX con el método **fetch()** hay que seguir los siguientes pasos:

1. **Configurar petición** (NO es obligatorio): El objeto `fetch()` debe ser configurado con sus 3 parámetros. El parámetro '`body`' no se configura si la petición es mediante GET, en cambio si la petición es mediante POST es necesario establecer los parámetros.

- `method`: El método de la petición (GET, POST, etc...)
- `body`: Los parámetros de la petición si se envía por POST.
- `headers`: Las cabeceras de la petición.

```
let configFetch = {  
    method: "POST",  
    body: "n1="+num1+"&n2="+num2,  
    headers:{ 'Content-Type': 'application/x-www-form-urlencoded' }  
};
```

Utilizando POST los parámetros se pasan en la cabecera. Con GET se pasan en `fetch()`.
`fetch("ejemplo.php?n1=2&n2=3", configFetch);`

2. **Lanzar la petición AJAX**: Se realiza con el objeto global **fetch()** pasándole como parámetro la URL o recurso de la petición y el objeto JSON de configuración (que no es obligatorio). Este método generará un objeto del tipo Promise a través del cual se gestionará la respuesta.

```
let promesa = fetch('respuesta.php', configFetch); //POST  
let promesa = fetch('respuesta.php ?n1=2&n2=3', configFetch); //GET
```

3. Configurar la función que va a recibir la respuesta siempre desde dentro del método '`then()`'
4. Gestionar la respuesta recibida en función del formato del elemento recibido ya que podría ser texto plano, un objeto JSON o un XML. Si fuese un objeto JSON debería pasearse (convertirse) mediante el método '`parse()`',

Lo primero que hay que realizar para acceder a un servicio tipo REST es ejecutar la petición mediante el método **fetch()** a una url específica o recurso que contiene los **datos en formato JSON o XML**. A continuación se muestran algunos ejemplos con diferentes niveles de dificultad.

Carencias de la API Fetch()

- NO es posible cancelar una petición ya solicitada.
- NO es posible controlar el proceso de carga de una solicitud larga.

Ejemplo 1: El método **fetch()** en la forma más básica sin cabecera recibiendo un texto plano. No se especifica el método de envío, por tanto será en **método GET**.

```
fetch('https://jsonplaceholder.typicode.com/users')  
  .then(function(response) {  
    return response.text();  
  })  
  .then(function(data) {  
    console.log('data = ', data);  
  })  
  .catch(function(err) {  
    console.error(err);  
  });
```

Explicación del código

1. Se llama al método '**fetch()**' con la URL a la que queremos acceder sin pasarle parámetros y sin configuración de la cabecera ya que es un recurso de internet.
2. Esta llamada nos devuelve una **promesa** que corresponde al primer 'then()'.
3. El método '**then()**' de esa promesa nos **entrega un objeto response**
4. Del objeto **response** llamamos al método '**text()**' para obtener el cuerpo retornado en forma de texto.
5. Nos devuelve **otra promesa** correspondiente al segundo 'then()' que se resolverá cuando se haya obtenido el contenido después de ejecutar el método '**text()**'
6. El método '**then()**' (el segundo) de esa promesa **recibe el cuerpo** (texto) devuelto por el servidor en formato de texto y lo muestra a través del método '**console.log()**'.
7. Hemos incluido un '**catch()**' por si se produce algún error que recibirá por parámetro el error devuelto si no se ha podido obtener los datos. Este método saltará cuando se produzca un error grave pero no si los datos recibidos no son los correctos.

Ejemplo 2: El método fetch() con cabecera enviado por POST con parámetros y recibiendo JSON.

```
fetch('https://httpbin.org/post', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded'
    },
    body: `a=${num1}&b=${num2}`
})
.then(function(response) {
    return response.json();
})
.then(function(data) {
    console.log('data = ', data);
})
.catch(function(err) {
    console.error(err);
});
```

Explicación del código

1. Se llama al método 'fetch()' con la URL a la que queremos acceder, pasándole la cabecera, el modo envío (POST) y dos parámetros.
2. Esta llamada nos devuelve una respuesta (1^{er} then) que será retornado y convertido previamente a JSON.
3. La nueva promesa (2^º then) recibe como parámetro los datos ya convertidos a JSON y los muestra a través de console.log.
4. Hemos incluido un 'catch()' por si se produce algún error que recibe por parámetro el error devuelto si no se ha podido obtener los datos. Este método saltará cuando se produzca un error grave pero no si los datos recibidos no son los correctos.

Ejemplo 3: El método **fetch()** pasándole los datos en la cabecera como JSON previa conversión del array mediante el método '**stringify()**'. El resto del método es igual que el ejemplo anterior.

```
fetch('https://httpbin.org/post',{
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({ "a": 1, "b": 2}),
    cache: 'no-cache'
})
.then(function(response) {
    return response.json();
})
.then(function(data) {
    console.log('data = ', data);
})
.catch(function(err) {
    console.error(err);
});
```

Explicación de la cabecera

- **method:** Método a utilizar (GET o POST)
- **headers:** Cabeceras que se deben enviar (ver objeto **Headers**).
- **body:** Cuerpo que se envía al servidor, que puede ser una cadena de texto, un objeto **Blob**, **BufferSource**, **FormData** o **URLSearchParams**.
- **mode:** Modo de la solicitud: 'cors', 'no-cors', 'same-origin', 'navigate'.
- **credentials:** Credenciales utilizadas: 'omit', 'same-origin', 'include'.
- **cache:** Forma de utilización de la caché: 'default', 'no-store', 'reload', 'no-cache', 'force-cache', 'only-if-cached'.
- **redirect:** Forma de gestionar la redirección: 'follow', 'error', 'manual'.
- **referrer:** Valor utilizado como referrer: 'client', 'no-referrer' una URL.
- **referrerPolicy:** Especifica el valor de la cabecera referer: 'no-referrer', 'no-referrer-when-downgrade', 'origin', 'origin-when-cross-origin', 'unsafe-url'.
- **integrity:** Valor de integridad de la solicitud.

Ejemplo 4: Aquí el método `fetch()` recibe un objeto '**Response**' después de la petición. El siguiente código muestra las diferentes propiedades de Response y la información que nos muestra a través de `console.log()`.

```
fetch('https://httpbin.org/ip')
  .then(function(response) {
    console.log('response.body =', response.body);
    console.log('response.bodyUsed =', response.bodyUsed);
    console.log('response.headers =', response.headers);
    console.log('response.ok =', response.ok);
    console.log('response.status =', response.status);
    console.log('response.statusText =', response.statusText);
    console.log('response.type =', response.type);
    console.log('response.url =', response.url);
    return response.json();
  })
  .then(function(data) {
    console.log('data =', data);
  })
  .catch(function(err) {
    console.error(err);
  });
}

=> Promise {}
response.body = ReadableStream {}
response.bodyUsed = false
response.headers = Headers {}
response.ok = true
response.status = 200
response.statusText = OK
response.type = cors
response.url = https://httpbin.org/ip
data = { origin: '83.48.36.133, 83.48.36.133' }
:>
```

El contenido del '`body`' no está disponible directamente desde el objeto **Response** y tenemos que invocar a uno de los métodos que dispone para que nos devuelva una promesa donde recibiremos el valor enviado por el servidor.

Una característica a tener en cuenta es que solo podemos obtener una vez los datos desde `body`, tras la cual no se puede realizar ninguna otra conversión más. Si deseamos hacer varias gestiones sobre el '`body`' deberemos duplicarlo con el método '`clone()`'.

Los métodos disponibles para el objeto Response son:

- `response.text()` para que nos devuelva el contenido en formato texto
- `response.json()` para que lo devuelva como objeto Javascript en formato JSON
- `response.arrayBuffer()` para obtenerlo como ArrayBuffer
- `response.blob()` como valor que podemos manejar con `URL.createObjectURL()`
- `response.formData()` para obtenerlos como `FormData`

Ejemplo 5: En este ejemplo el método **fetch()** se ejecuta pasándole como parámetro de configuración de la petición un objeto **Request**. Este objeto se debe crear previamente a la ejecución de fetch() y contendrá todas las características de la petición. De esta forma se podrían crear diferentes peticiones AJAX con sus correspondientes parámetros cada una.

Aunque podemos incluir las cabeceras por medio del objeto Request o como parte del objeto que se pasa como segundo parámetro a **fetch()**, tenemos a nuestra disposición el objeto **Headers** que nos ayuda a gestionar las cabeceras de una forma más precisa.

```
// Almacena la configuración de la petición
var request = new Request('https://httpbin.org/get', {
    method: 'GET',
    mode: 'cors',
    credentials: 'omit',
    cache: 'only-if-cached',
    referrerPolicy: 'no-referrer'
});

fetch(request) // Solo recibe la variable de configuración Request
    .then(function(response) { return response.text(); })
    .then(function(data) { console.log('data = ', data); })
    .catch(function(err) { console.error(err); });
```

Ejemplo 6: Utilización del método **fetch()** en combinación con arrow functions.

```
var usuarios = [];
var data;

fetch('https://jsonplaceholder.typicode.com/users')

    .then( response => { return response.text(); })

    .then( data => { console.log('data = ', data); })

    .catch( err => { console.error(err); });
```

Depurar Código AJAX

Uno de los aspectos más relevantes es poder realizar la depuración del código asíncrono con el depurador del navegador. Como AJAX es una tecnología en la que interviene el cliente y el servidor, su depuración es un poco más compleja.

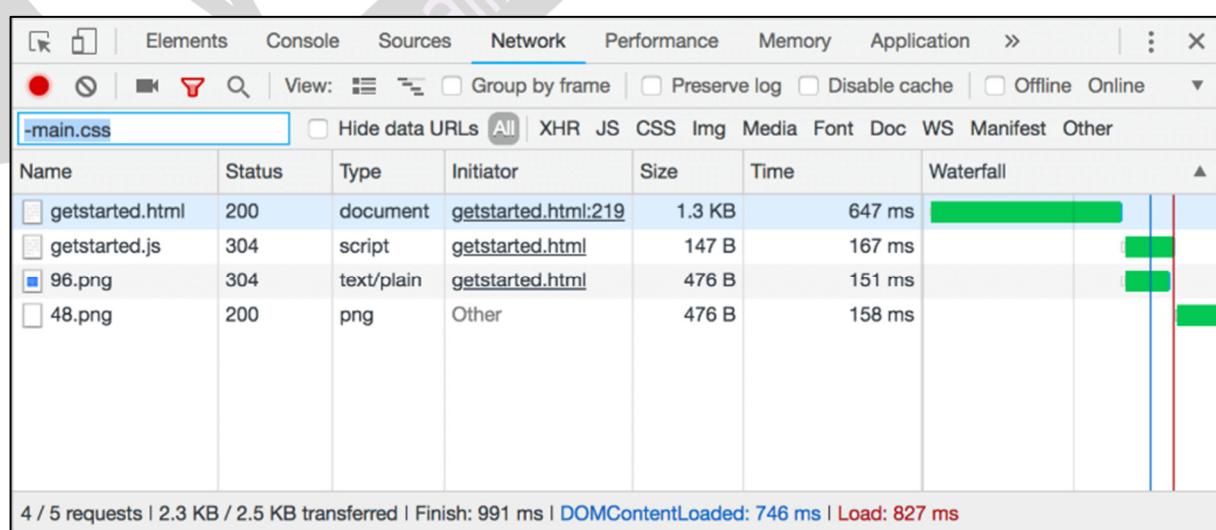
Ahora no solo vamos a poder tener errores en el mismo código AJAX sino que los errores se van a poder producir también en la misma comunicación , en el servidor o en su respuesta. Por todo ello para depurar los programas desarrollados con AJAX es esencial utilizar el inspector de red en las herramientas de desarrollo del navegador

Para ello es necesario disponer instalado en nuestro editor el plugin '**PHP Server**' e iniciararlo.

Además puede ser necesario acceder al sistema operativo, '*Propiedades del Sistema > Variables entorno*' y añadir una variable de entorno que contenga la ruta donde se encuentra nuestro intérprete .php.

La ventana del depurador es capaz de mostrar información de las peticiones AJAX como:

- Nombre del archivo solicitado.
- Estado de la petición. Si ha sido satisfactoria la respuesta es el valor de 200.
- Tipo de documento que se ha solicitado (documento, texto plano, javascript, png, etc...).
- Tamaño del archivo solicitado.
- Tiempo que ha tardado la transacción.
- Orden de secuencia de las diferentes peticiones.



PROMESAS en JavaScript

Las promesas (PROMISE) son unos **objetos en JavaScript** que representan la terminación o fracaso de una **operación asíncrona**. Una promesa o petición a un servidor puede tardar desde unos pocos milisegundos hasta varios segundos, por tanto, la promesa se ejecuta y no se sabe ni el resultado ni el tiempo que tardará en ejecutarse.

Es una petición asíncrona a un servidor que puede dar como resultado que se ha ejecutado satisfactoriamente ('**Fulfilled**') o que ha fracasado ('**Rejected**').

- Forman parte del estándar desde la versión ECMAScript 6.
- Las promesas representan una acción que eventualmente se tiene que completar. A priori una promesa se hace porque se tiene que cumplir y se espera que no de ningún error.
- La promesa se ejecuta solo una vez de forma predeterminada. Si se quisiese volver a ejecutar se tendría que programar mediante código.
- Las promesas resueltas bien pueden dar como resultado la ejecución de nuevas promesas.
- Las **promesas se pueden concatenar**, es decir, hasta que no llega satisfactoriamente el resultado de una promesa no se ejecuta la siguiente promesa.
- Las promesas **NO se ejecutan de forma secuencial** ya que forman parte de una conexión asíncrona, es decir, puede haber diferentes promesas ejecutándose al mismo tiempo.
- Una promesa puede tener 3 estados diferentes, pero nunca tener dos estados a la vez.
 1. Pending (Pendiente): La promesa aún no ha finalizado la petición, pero la está intentando ejecutar.
 2. Fulfilled (Completada): La promesa se ha ejecutado de forma correcta y ha devuelto un resultado.
 3. Rejected (Rechazada): La promesa no se ha podido ejecutar y ha sido denegada, por ejemplo, por falta de conexión al servidor.

Para crear una promesa desde cero se realiza con el objeto **Promise()**. El siguiente código muestra cómo se ejecutaría una promesa en caso que los datos JSON no se pudieran convertir a 'string'. El objeto Promise recibe una función de flecha por callback y devolverá el resultado.

```
return new Promise((resolve, reject) => {
    var alumnos_string = JSON.stringify(alumnos_datos);
    if(typeof alumnos_string != 'string') {
```

```
        return reject('Error en la conversion');

    } else {

        return resolve (alumnos\_string);
    }
}
```



API Media

Son las API para trabajar con video y audio. HTML5 incluye un elemento con el propósito de cargar y reproducir videos. Antes de HTML5 para visualizar un video se necesitaba un plug-in (Flash, Quicktime, etc...). La nueva etiqueta <video> de HTML5 permite incrustar videos en la web. La siguiente imagen muestra a partir de qué versión de cada navegador es aceptada la etiqueta <video>.

Element					
<video>	4.0	9.0	3.5	4.0	10.5

El siguiente código muestra la inserción básica de un elemento de video. A continuación, está la explicación de cada uno de los atributos.

```
<video width="320" height="240" controls autoplay loop>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.ogg" type="video/ogg">
  <source src="movie.webm" type="video/webm">
  Texto si el navegador no puede mostrar el video.
</video>
```

- Los atributos 'width' y 'height' especifican el tamaño (ancho y alto respectivamente) de la ventana que contendrá el video.
- El atributo 'controls' perteneciente a la etiqueta <video> permite mostrar una serie de controles debajo del video (play, pause, volumen, línea de reproducción, y descarga).
- El atributo 'autoplay' comenzará a reproducir el archivo en el momento que esté disponible.
- El atributo 'loop' realiza la reproducción ininterrumpida del video mientras no se pause, el navegador reproduce el video una y otra vez.
- El atributo 'src' de la etiqueta <source> enlaza con el video que se desea cargar. Es posible incluir el mismo video en diferentes formatos por si el navegador no es capaz de reproducir el archivo. En este caso siempre mostrará el primer video que pueda ejecutar.
- El atributo 'type' de la etiqueta <source> (nueva desde HTML5) especifica el formato del archivo a reproducir. Es posible especificar varios formatos de carga del archivo.
- El texto inferior solo se mostrará si no se pudiera reproducir el video por algún problema que se pueda producir con la carga del archivo desde el servidor.

Atributos adicionales para la etiqueta <video>:

- '**muted**' especifica que la salida de video o audio debe estar silenciada.
- '**poster**' especifica la imagen que se mostrará mientras se carga el video o hasta que sobre el video pulse el botón de reproducción.
- '**preload**' especifica cuando y como se debe cargar el video en la página. Los valores para este atributo se muestran en un ejemplo a continuación (solo puede haber uno de ellos). El valor '*auto*' es el más utilizado por defecto y solicita al video que descargue el archivo tan pronto como sea posible.

```
<video preload="auto|metadata|none">
```

Atributos adicionales para la etiqueta <source>:

- '**media**' determina el tipo de dispositivo donde será visualizada la página web. Este tipo de acción se suele realizar desde CSS, pero también es posible realizarlo desde esta etiqueta. Los valores más utilizados en esta etiqueta para los diferentes tipos de dispositivos son:
 - All: Disponible para todos los dispositivos, es la opción por defecto.
 - Aural: Sintetizadores de voz.
 - Braille: Dispositivos para Braille
 - Handheld: dispositivos pequeños, de pantalla pequeña y ancho banda limitado.
 - Projection: Para proyectores de video.
 - Print: modo vista preliminar.
 - Screen: Pantallas de ordenadores.
 - Tv: Dispositivos de Tv, baja resolución y limitaciones de desplazamiento con barra.

Además, es posible combinar el tipo de dispositivo de la etiqueta <source> con una serie de valores relacionados con el aspecto, orientación, tamaño, etc... Estos valores serán enlazados con la conjunción 'and'. A continuación, se muestran unos ejemplos con los posibles valores y tipos de dispositivos.

Ancho mínimo: <source media="screen and (min-width:500px)">

Alto mínimo: <source media="screen and (max-height:700px)">

Ancho de la pantalla de destino: <source media="screen and (device-width:500px)">

Formatos de Video

A priori, con la etiqueta **<video>** debería ser más que suficiente para poder incluir videos en una web. Pero en realidad al haber tantos formatos no hay un formato standard para la web y aunque se han reducido mucho todos los que había en la actualidad quedan 3 formatos que son los más adecuados. Los formatos que se muestran a continuación soportan y son aceptados por HTML5.

.OGG: Desarrollado por Xiph.Org Foundation (organización sin ánimo lucro que crea formatos multimedia y software libre). Posiblemente sea el más utilizado en la actualidad y es compatible con todos los navegadores. Utiliza los codificadores de video de Theora y de audio de Vorbis.

.WEBM: Desarrollado por los gigantes web (Mozilla, Google, Adobe y Opera). Es compatible con todos los navegadores. Utiliza el codificador de video VP8 y de audio el de Vorbis.

.MP4: Desarrollado por el Moving Pictures Expert Group. Es el recomendado por Youtube pero no está soportado por todos los grandes navegadores y teóricamente se distribuye bajo licencia comercial. Utiliza el codificador de video H.264 y de audio el AAC.

Cuando se deben especificar varias fuentes para un mismo video se deberá utilizar la etiqueta **<source>** de HTML5. Una vez especificados todas las fuentes, el navegador seleccionará la que considere más fácil para reproducir.

NOTA: NO todos los servidores web aceptan todos los formatos de video, es decir, hay que especificar el tipo MIME (ejemplo: text/html para página web). Una forma sencilla es hacerlo en el archivo de configuración **.htaccess** que incluyen todos los servidores incluyendo la línea:

- AddType video/ogg ogg
- AddType video/WebM WebM

Para reproducir archivos de audio HTML5 dispone de la etiqueta **<audio>** que básicamente inserta audio en un documento .html. Este elemento trabaja y comparte los mismos atributos que la etiqueta **<video>**. Igual que sucedía con los archivos de video