CURSO DE PROGRAMACIÓN FULL STACK

# GUIA DE JAVASCRIPT


# GUIA DE JAVASCRIPT

## ¿QUÉ ES JAVASCRIPT?

JavaScript (abreviado comúnmente JS) es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Desde 2012, todos los navegadores modernos soportan completamente ECMAScript 5.1, una versión de JavaScript.

Se utiliza principalmente del lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas y JavaScript del lado del servidor (Server-side JavaScript o SSJS). Su uso en aplicaciones externas a la web, por ejemplo en documentos PDF, aplicaciones de escritorio (mayoritariamente widgets) es también significativo.

JavaScript se diseñó con una sintaxis similar al lenguaje de programación C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del *Document Object Model* (DOM).

Tradicionalmente se venía utilizando en páginas web HTML para realizar operaciones y únicamente en el marco de la aplicación cliente, sin acceso a funciones del servidor. Actualmente es ampliamente utilizado para enviar y recibir información del servidor junto con ayuda de otras tecnologías como AJAX. JavaScript se interpreta en el agente de usuario al mismo tiempo que las sentencias van descargándose junto con el código HTML.

Hay 3 formas de sumar JavaScript a nuestro HTML:

```
<!DOCTYPE html>
<html>

<head>
<script>
function miFuncion() {
  document.getElementById("demo").innerHTML = "Parrafo
cambiado!.";
}
</script>
</head>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo Inicial</p>
<button type="button" onclick="myFunction()">Intentalo</button>

</body>
</html>
```

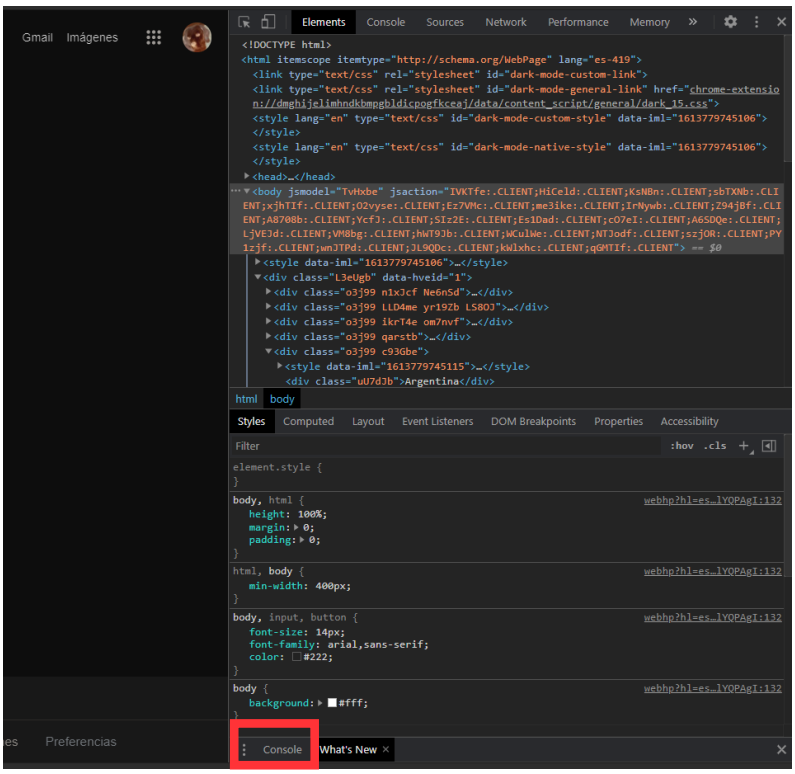
```
<!DOCTYPE html>
<html>

<head>
</head>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo Inicial</p>
<button type="button" onclick="myFunction()">Intentalo</button>

<script>
function miFuncion() {
  document.getElementById("demo").innerHTML = "Parrafo
cambiado!.";
}
</script>

</body>
</html>
```

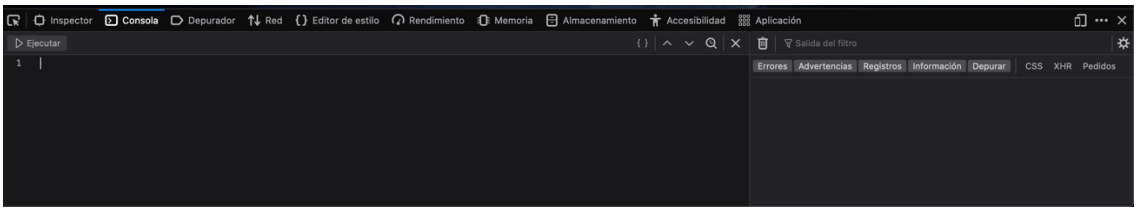
```
<!DOCTYPE html>
<html>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo</p>

<button id="pinchable" type="button">Intentalo</button>

<script src="script1.js"></script>
<script src="script2.js"></script>

</body>
</html>
```

# SALIDA Y ENTRADA DE DATOS

Una **consola web** es una herramienta que se utiliza principalmente para registrar información asociada a una página web como: solicitudes de red, JavaScript, errores de seguridad, advertencias, CSS, etc. Esta, nos permite interactuar con una página web ejecutando una expresión JavaScript en el contenido de la página.

En JavaScript, **Console,** es un objeto que proporciona acceso a la consola de depuración del navegador. Podemos abrir una consola en el navegador web Chrome, usando: Ctrl + Shift + J para Windows/Linux y Option + Command ⌘ + J para Mac.




El objeto **"console"** nos proporciona varios métodos diferentes, como:

- log()
- error()
- warn()
- clear()
- time() y timeEnd()
- table()
- count()
- group() y groupEnd()
- custom console logs

| Método | Descripción | Ejemplo |
|---|---|---|
| console.log() | Se utiliza principalmente para registrar (imprimir) la salida en la consola. Podemos poner cualquier tipo dentro del log (), ya sea una cadena, matriz, objeto, booleano, etc. | `console.log("abc");`<br>`console.log(123);`<br>`console.log([1,2,3,4]);` |

| | | |
|---|---|---|
| console.error()<br><br>console.warn | Error: se utiliza para registrar mensajes de error en la consola.<br><br>Warn: se usa para registrar mensajes de advertencia | `console.error("Mensaje de error");`<br><br>`console.warn("Mensaje de advertencia");` |
| console.clear() | Se usa para limpiar la consola. | `console.clear();` |
| console.time()<br><br>console.timeEnd() | Siempre que queramos saber la cantidad de tiempo empleado por un bloque o una función, podemos hacer uso de los métodos time() y timeEnd(). | `console.time('abc');`<br><br>`console.timeEnd('abc');` |
| console.table() | Este método nos permite generar una tabla dentro de una consola. La entrada debe ser una matriz o un objeto que se mostrará como una tabla. | `console.table({'a':1, 'b':2});` |
| console.count() | Este método se usa para contar el número que la función alcanza con este método de conteo. | `for(let i=0;i<5;i++){`<br>`  console.count(i);`<br>`}` |

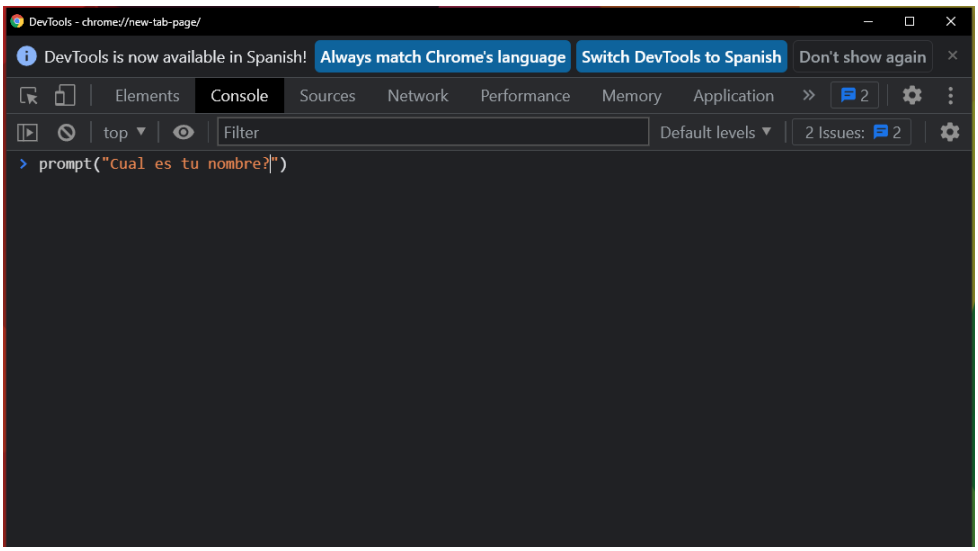
```
//console custom

const spacing = '10px';

const styles = `padding: ${spacing}; background-color: black; color:
yellow; font-style:

         italic; border: 1px solid black; font-size: 2em;`;

  console.log('%cEGG', styles);
```

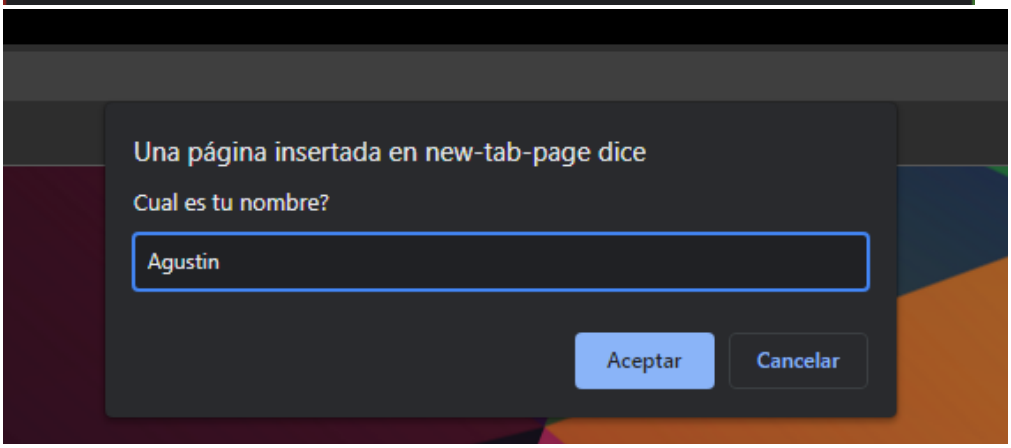
# OBJETO WINDOW

El objeto window de Javascript nos sirve para controlar la ventana del navegador. Es el objeto principal en la jerarquía y contiene las propiedades y métodos para controlar la ventana del navegador. Tiene algunos métodos como:

## PROMPT

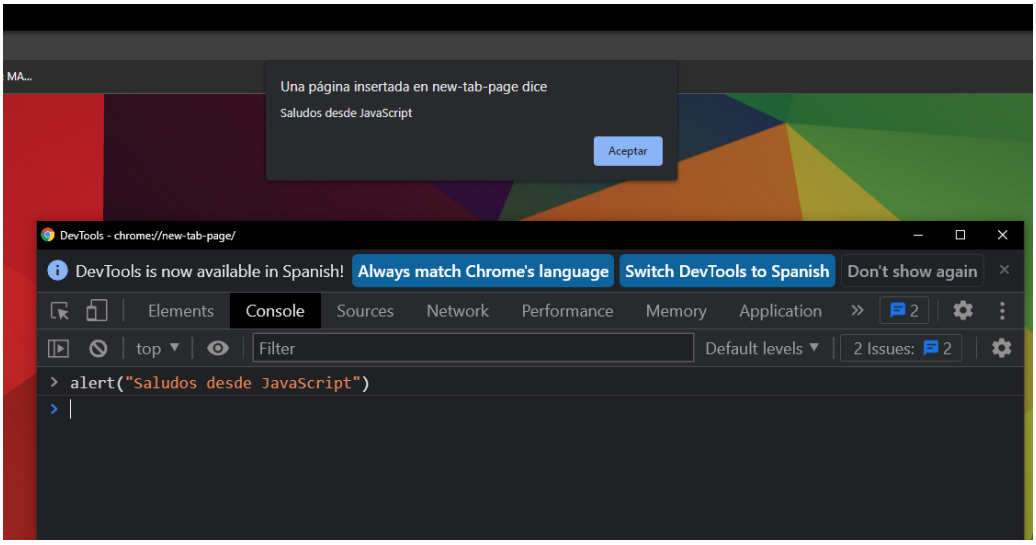
El método Window.prompt() muestra un diálogo con mensaje opcional, que solicita al usuario que introduzca un texto.




## ALERT

El método Window.alert() muestra un diálogo de alerta con contenido opcional especificado y un botón OK (Aceptar).



# VARIABLES EN JAVASCRIPT

En muchos lenguajes de programación hay unas reglas extrictas a la hora de declarar las variables, pero lo cierto es que Javascript es bastante permisivo.

Javascript se salta muchas reglas por ser un lenguaje un tanto libre a la hora de programar y uno de los casos en los que otorga un poco de libertad es a la hora de declarar las variables, ya que JavaScript, no nos obliga a declarar las variables, al contrario de lo que pasa en otros lenguajes de programación como Java, C, C# y muchos otros.

Otra cosa en la que JavaScript es más permisivo, es que, al ser de tipado suave, no tenemos que poner el tipo de dato de la variable a la hora de declararla, sino que al darle un valor, ahí toma el tipo de dato.

Aunque no es obligatorio declarar variables, recomendamos siempre declararlas antes de usarlas.

## VAR

Javascript cuenta con la palabra "var" que utilizaremos cuando queramos declarar una o varias variables. Como es lógico, se utiliza esa palabra para definir la variable antes de utilizarla.

Esta variable se convertirá en una propiedad del objeto global, es decir sin importar donde se declare, todos tendrán la oportunidad de llamarla y utilizarla.

```
var variable;
```

También se puede asignar un valor a la variable cuando se está declarando

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

A la hora de escribir nuestras variables mantenemos algunas buenas practicas de Java como:

- Siempre se escriben en minúsculas
- Usamos el CamelCase cuando queremos que sean dos palabras
- Podemos usar guion bajo, para dos palabras también
- Nunca usamos la Ñ, ni tildes, ni ningún carácter especial

# AMBITO DE VARIABLES

Antes de explicar la declaración de variables con la palabra let, tenemos que explicar que es el ámbito de variables

Se le llama ámbito de las variables al lugar donde estas están disponibles. Por lo general, cuando declaramos una variable hacemos que esté disponible en el lugar donde se ha declarado. Esto es igual a lo que hemos visto en Java, las variables globales pueden ser accedidas en cualquier lugar y las variable locales pueden ser accedidas solo donde se declararon, por ejemplo una función.

## LET

Desde ECMAScript 2015 existe la declaración **let**. La sintaxis es la misma que var a la hora de declarar las variables, pero en el caso de **let** la declaración afecta al bloque.

Bloque significa cualquier espacio acotado por unas llaves, como podría ser las sentencias que hay dentro de las llaves de un bucle for.

Al declarar una variable con la palabra clave **let**, dentro de un bloque, hacemos que esa variable solo pueda ser accedida dentro de ese bloque concreto, por lo que solo existe dentro del bloque que la contiene. Lo que la haría una **variable local.**

A diferencia de la palabra clave var, la cual define una variable global o local en una función, sin importar el ámbito del bloque.

```
function varPrueba() {
  var x = 31;
  if (true) {
    var x = 71; // Misma variable!
    console.log(x); // Imprime el valor 71
  }
  console.log(x); // Imprime el valor 71
}
function letPrueba() {
  let x = 31;
  if (true) {
    let x = 71; // variable diferente
    console.log(x); // Imprime el valor 71
  }
  console.log(x); // Imprime el valor 31
}
```

### CONST

La sentencia const sirve para declarar una constante cuyo alcance puede ser global o local para el bloque en el que se declara. Es necesario inicializar la constante, es decir, se debe especificar su valor en la misma sentencia en la que se declara, lo que tiene sentido, dado que no se puede cambiar posteriormente.

```
const PI = 3.141592653589793;

PI = 3.14;  // Esto dará un error

PI = PI + 10; // Esto también dará un error
```

## TEMPLATE STRINGS

Las template strings, o cadenas de texto de plantilla, son una de las herramientas de ES6 para trabajo con cadenas de caracteres que nos pueden venir muy bien para producir un código Javascript más claro. Usarlas es por tanto una recomendación dado que facilitará el mantenimiento de los programas, gracias a que su lectura será más sencilla con un simple vistazo del ojo humano.

### CONCATENACIÓN DE VARIABLES

En un programa realizado en JavaScript, y en cualquier lenguaje de programación en general, es normal crear cadenas en las que tenemos que juntar cadenas con los valores tomados desde las variables.

```
var sitioWeb = "DesarrolloWeb.com";

var mensaje = 'Bienvenido a ' + sitioWeb;
```

Eso es muy fácil de leer, pero a medida que el código se complica y en una cadena tenemos que concatenar el contenido de varias variables, el código comienza a ser más enrevesado.

```
var nombre = 'Miguel Angel';

var apellidos = 'Alvarez'

var profesion = 'desarrollador';

var perfil = ' ' + nombre + ' ' + apellidos +  ' es ' + profesion;
```

Quizás estás acostumbrado a ver esto así. El código está bien y no tiene ningún problema, pero podría ser mucho más bonito si usas los template strings.

### CREAR UN TEMPLATE STRING

Para crear un template string simplemente tienes que usar un carácter que se usa poco, como apertura y cierre de la cadena. Es el símbolo del acento grave. (`)

```
var cadena = `Esto es un template String`;
```

## USOS DE LOS TEMPLATE STRINGS

Los template strings tienen varias características interesantes que, cómo decíamos, facilitan la sintaxis. Veremos a continuación algunos de ellos con código de ejemplo.

## CONCATENACION DE VALORES

Creo que lo más interesante es el caso de la concatenación que genera un código poco claro hasta el momento. Echa un vistazo al código siguiente que haría lo mismo que el que hemos visto anteriormente del perfil.

```
var nombre = 'Miguel Angel';

var apellidos = 'Alvarez'

var profesion = 'desarrollador';

var perfil = `<b>${nombre} ${apellidos}</b> es ${profesion}`;
```

Como puedes comprobar, dentro de un template string es posible colocar expresiones encerradas entre llaves y precediendo de un símbolo "$". Algo como ${ expresion }.

En las expresiones podemos colocar código que queramos volcar, dentro de la cadena. Las usamos generalmente para colocar valores de variables, pero también servirían para colocar operaciones matemáticas, por ejemplo.

```
var suma = `45 + 832 = ${45 + 832}`;
```

O bien algo como esto:

```
var operando1 = 7;

var operando2 = 98;

var multiplicacion = `La multiplicación entre ${operando1} y ${operando2}
equivale a ${operando1 * operando2}`;
```

## SALTOS DE LÍNEA DENTRO DE CADENAS

Hasta ahora, si queremos hacer una cadena con un salto de línea teníamos que usar el caracter de escape "contrabarra n".

```
var textoLargo = "esto es un texto\ncon varias líneas";
```

Con un template string tenemos la alternativa de colocar el salto de línea tal cual en el código y no producirá ningún problema.

```
var textoLargo = `esto es un texto

con varias líneas`;
```

# TIPOS DE DATOS EN JAVASCRIPT

Los tipos de datos JavaScript se dividen en dos grupos: tipos primitivos y tipos objeto.

| PRIMITIVOS | |
|---|---|
| **Dato** | **Descripción.** |
| Numérico | Números, ya sea enteros o reales. |
| String | Cadenas de texto. |
| Boolean | Valores lógicos como true o false. |
| null | Cuando un dato no existe. |
| undefined | Cuando no se le asigna un valor a la variable. |

Los tipos objeto tienen sus propias subdivisiones

| OBJETOS | |
|---|---|
| **Tipo De Objeto** | **Descripción** |
| Tipo predefinido de JavaScript | Date: fechas |
| Tipo predefinido de JavaScript | RegExp: expresiones regulares |
| Tipo predefinido de JavaScript | Error: datos de erros |
| Tipos definidos por el programador / usuario | Funciones Simples y Clases |
| Arrays | Serie de elementos o formación tipo vector o matriz. Lo consideramos un objeto especial |

| | |
|---|---|
| Objetos Especiales | Objeto Global |
| Objetos Especiales | Objeto prototipo |
| Objetos Especiales | Otros |

## OPERADORES EN JAVASCRIPT

Al igual que Java tendremos operadores para trabajar con datos, aunque hay algunos operadores que son distintos a los que conocemos en Java.

| OPERADOR DE ASIGNACIÓN | |
|---|---|
| = | Operador de Asignación Simple |
| **OPERADORES ARITMÉTICOS** | |
| + | Operador de Suma |
| - | Operador de Resta |
| * | Operador de Multiplicación |
| ** | Exponenciación |
| / | Operador de División |
| % | Operador de Módulo |
| **OPERADORES UNARIOS** | |
| ++ | Operador de Incremento. |
| -- | Operador de Decremento. |
| += | y += x |

| | |
|---|---|
| -= | y -= x |
| *= | y *= x |
| /= | y /= x |
| %= | y %= x |
| **= | y **= x |

## OPERADORES LOGICOS Y RELACIONALES

| | |
|---|---|
| == | Es igual Ejemplo: 3 == “3” |
| === | Es estrictamente igual Ejemplo: 3 === 3 |
| != | Distinto |
| !== | Estrictamente Distinto |
| > | Mayor que |
| >= | Mayor o igual que |
| < | Menor que |
| <= | Menor o igual que |

## OPERADORES CONDICIONALES

| | |
|---|---|
| && | AND |
| \|\| | OR |
| ! | Operador Lógico de Negación. |

| OPERADORES DE COMPARACIÓN DE TIPO | |
|---|---|
| typeof | Devuelve el tipo de dato de una variable |
| Instanceof | Devuelve true si el objeto es una instancia de. |

### TYPEOF

La función typeof se utilizada para obtener el tipo de dato que tiene una variable.

```
console.log(typeof 42);
// expected output: "number"
console.log(typeof 'blubber');
// expected output: "string"
console.log(typeof true);
// expected output: "boolean"
```

## CONDICIONALES EN JAVASCRIPT

Al igual que en Java, existen los condicionales que nos van a ayudar a modificar el flujo de ejecución del programa.

### IF

El condicional if es un condicional lógico que evalúa el camino a tomar en base a la evaluación de una condición. Supongamos el siguiente ejemplo, mi sobrino quiere subirse a una montaña rusa, pero para ello tiene que aprobar las dos siguiente condiciones: tener mas de 18 años y medir mas de 160 cm. La evaluación de esas dos condiciones da por verdadero se podrá subir de lo contrario no podrá.

```
let edad = 15;
let altura = 166;
if(edad>18 && altura>160){
    console.log("Puedes subirte :D");
}else{
    console.log("No te podes subir”);
}
```

Como se puede ver, si la condición a evaluar se cumple, es decir, da verdadero, mostrara el mensaje “Puedes subirte :D”, en caso que de falso mostrara “No te puedes subir ”. Por otra parte, JavaScript permite también agregar la condición else if

```
if(a == 2){
    console.log("a es igual a 2");
}else if(a < 2){
    console.log("a es menor que 2");
}else{
    console.log("a es mayor que 2");
}
```

## IF TERNARIO

El if ternario nos permite resolver en una línea una expresión lógica asignando un valor. Proviene del lenguaje C, donde se escriben muy pocas líneas de código y donde cuanto menos escribamos más elegantes seremos. Este operador es un claro ejemplo de ahorro de líneas y caracteres al escribir los scripts. Lo veremos rápidamente, pues la única razón que lo veamos es para que sepan que existe y si lo encuentran en alguna ocasión sepan identificarlo y cómo funciona.

```
Variable = (condición) ? valor1 : valor2
```

Este ejemplo no sólo realiza una comparación de valores, además asigna un valor a una variable. Lo que hace es evaluar la condición (colocada entre paréntesis) y si es positiva asigna el valor1 a la variable y en caso contrario le asigna el valor2. Veamos un ejemplo:

```
momento = (hora_actual < 12) ? "Antes del mediodía" : "Después del
mediodía"
```

## SWITCH

La declaración switch evalúa una expresión, comparando el valor de esa expresión con una instancia case, y ejecuta declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.

El programa primero busca la primer instancia case cuya expresión se evalúa con el mismo valor de la expresión de entrada (usando comparación estricta, ===) y luego transfiere el control a esa cláusula, ejecutando las declaraciones asociadas. Si no se encuentra una cláusula de case coincidente, el programa busca la cláusula default opcional, y si se encuentra, transfiere el control a esa instancia, ejecutando las declaraciones asociadas

Al igual que Java, la declaración break es opcional y está asociada con cada etiqueta de case y asegura que el programa salga del switch una vez que se ejecute la instrucción coincidente y continúe la ejecución en la instrucción siguiente. Si se omite el break el programa continúa la ejecución en la siguiente instrucción en la declaración de switch .

```
switch (expr) {
  case 'Naranjas':
    console.log('El kilogramo de naranjas cuesta $0.59.');
    break;
  case 'Mangos':
  case 'Papayas':
    console.log('El kilogramo de mangos y papayas cuesta $2.79.');
    break;
  default:
    console.log('Lo lamentamos, por el momento no disponemos de ' + expr +
'.');
}
```

# ESTRUCTURAS REPETITIVAS

Veamos como son las estructuras repetitivas en JavaScript

## WHILE

Crea un bucle que ejecuta una sentencia especificada mientras cierta condición se evalúe como verdadera. Dicha condición es evaluada antes de ejecutar la sentencia

```
let a = 0;
while(a != 10){
    console.log(++a);
}
```

## DO WHILE

La sentencia (hacer mientras) crea un bucle que ejecuta una sentencia especificada, hasta que la condición de comprobación se evalúa como falsa. La condición se evalúa después de ejecutar la sentencia, dando como resultado que la sentencia especificada se ejecute al menos una vez.

```
let a = 0;
do{
    console.log(++a);
}while(a!=10);
```

## FOR

El bucle FOR se utiliza para repetir una o más instrucciones un determinado número de veces. De entre todos los bucles, el FOR se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute. La sintaxis del bucle for va a ser la misma que en Java:

```
for ([expresion-inicial]; [condicion]; [expresion-final]){
}
```

```
for(let i = 0; i < 10; i++){
    console.log("El valor de i es " + i);
}
```

## BREAK

Termina el bucle actual, sentencia switch o label y transfiere el control del programa a la siguiente sentencia.

```
for (let i = 0; i < 10; i++) {
    if(i == 5){
        break;
    }
    console.log("Estamos por la vuelta "+i);
}
```

## CONTINUE

Termina la ejecución de las sentencias de la iteración actual del bucle actual o la etiqueta y continua la ejecución del bucle con la próxima iteración.

En contraste con la sentencia break, continue no termina la ejecución del bucle por completo; en cambio,

- En un bucle while, salta de regreso a la condición.
- En un bucle for, salta a la expresión actualizada.

La sentencia continue puede incluir una etiqueta opcional que permite al programa saltar a la siguiente iteración del bucle etiquetado en vez del bucle actual. En este caso, la sentencia continue necesita estar anidada dentro de esta sentecia etiquetada.

```
for (let i = 0; i < 10; i++) {
    if(i == 5){
    continue;
}
console.log("Estamos por la vuelta " + i);
}
```

### LABEL

Proporciona a una sentencia con un identificador al que se puede referir al usar las sentencias break o continue. Por ejemplo, puede usar una etiqueta para identificar un bucle, y entonces usar las sentencias break o continue para indicar si un programa debería interrumpir el bucle o continuar su ejecución.

```
label o etiqueta : sentencia
```

Ejemplo:

```
exterior: for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
       if(i == 4 && j == 4){
         console.log("Vamos a cortar ambos for");
         break exterior;
    }
 console.log(i+j+10*i);
}}
```

## FUNCIONES EN JAVASCRIPT

Una función, en JavaScript es, al igual que Java, como una serie de instrucciones que englobamos dentro de un mismo proceso. Este proceso se podrá luego ejecutar desde cualquier otro sitio con solo llamarlo. Por ejemplo, en una página web puede haber una función para cambiar el color del fondo y desde cualquier punto de la página podríamos llamarla para que nos cambie el color cuando lo deseemos.

La sintaxis de una función en JavaScript es la siguiente:

```
function nombrefuncion (){
   instrucciones de la función
   ...
}
```

Primero se escribe la palabra function, reservada para este uso. Seguidamente se escribe el nombre de la función, que como los nombres de variables puede tener números, letras y algún carácter adicional como en guión bajo y los paréntesis donde irán nuestros parámetros. A continuación se colocan entre llaves las distintas instrucciones de la función. Las llaves en el caso de las funciones no son opcionales, además es útil colocarlas siempre como se ve en el ejemplo, para que se reconozca fácilmente la estructura de instrucciones que engloba la función.

Un ejemplo de función que escribe hola mundo en la consola sería:

```
function holaMundo(){
console.log('Hola Mundo');
}
```

Después para llamar a esta función, hay que invocarla mediante su nombre

```
holaMundo();
```

## DÓNDE COLOCAMOS LAS FUNCIONES JAVASCRIPT

En principio, podemos colocar las funciones en cualquier parte de la página, siempre entre etiquetas **<script>**. No obstante existe una limitación a la hora de colocarla con relación a los lugares desde donde se la llame. Usualmente lo más fácil es colocar la función antes de cualquier llamada a la misma y así seguro que nunca nos equivocaremos.

Existen dos opciones posibles para colocar el código de una función:

a) **Colocar la función en el mismo bloque de script:** En concreto, la función se puede definir en el bloque <script> donde esté la llamada a la función, aunque es indiferente si la llamada se encuentra antes o después del código de la función, dentro del mismo bloque <script>.

```
<script>
miFuncion();
function miFuncion(){
   //hago algo...
   console.log("Esto va bien");
}
</script>
```

b) **Colocar la función en otro bloque de script:** También es válido que la función se encuentre en un bloque <SCRIPT> anterior al bloque donde está la llamada.

```
<html>
<head>
   <title>MI PÁGINA</title>
<script>
function miFuncion(){
   //hago algo...
   console.log("Esto va bien");
}
</script>
</head>
<body>
<script>
miFuncion()
</script>
</body>
</html>
```

## PARAMETROS DE LAS FUNCIONES

Los parámetros se usan para mandar valores a las funciones. Una función trabajará con los parámetros para realizar las acciones. Por decirlo de otra manera, los parámetros son los valores de entrada que recibe una función.

Los parámetros en JavaScript, son iguales que en Java, la única diferencia es que, a diferencia de Java, no tenemos que especificar el tipo de dato que recibe la función, sino que va a ser el tipo de dato que enviemos como parámetro.

```
function escribirBienvenida(nombre){
console.log('Hola ' + nombre);
}
escribirBienvenida('Agustin');
// O podemos hacerlo con una variable
let nombre = 'Agustin';
escribirBienvenida(nombre);
```

## DEVOLVER VALORES EN FUNCIONES

Las funciones en Javascript también pueden **retornar valores**. De hecho, ésta es una de las utilidades más esenciales de las funciones.

Veamos un ejemplo de función que calcula la media de dos números. La función recibirá los dos números y retornará el valor de la media.

```
function media(valor1,valor2){
      let resultado;
      resultado = (valor1 + valor2) / 2;
      return resultado;
}
```

Para especificar el valor que retornará la función se utiliza la palabra **return** seguida de el valor que se desea devolver. En este caso se devuelve el contenido de la variable resultado, que contiene la media calculada de los dos números.

Pero, cómo podemos recibir un dato que devuelve una función. Cuando una función devuelve un valor simplemente se sustituye la llamada a la función por ese valor que devuelve. Así pues, para almacenar un valor de devolución de una función, tenemos que asignar la llamada a esa función como contenido en una variable, y eso lo haríamos con el operador de asignación **=**.

```
let miMedia
miMedia = media(12,8);
console.log(miMedia);
```

# FUNCIONES FLECHA

Hay otra sintaxis muy simple y concisa para crear funciones, que a menudo es mejor que las Expresiones de funciones.

Se llama **"funciones de flecha"**, porque se ve así:

```
let func = (arg1, arg2, ..., argN) => expresion
```

Esto crea una función **func (nombre de la función)** que acepta parámetros arg1..argN, luego evalúa la **expresion** de la derecha con su uso y devuelve su resultado.

En otras palabras, es la versión más corta de:

```
let func = function(arg1, arg2, ..., argN) {
  return expresion;
};
```

Veamos un ejemplo completo:

```
let sum = (a, b) => a + b;

/* Esta función de flecha es una forma más corta de:

let sum = function(a, b) {
  return a + b;
};

*/

console.log(sum(1, 2)); // 3
```

Como puedes ver (a, b) => a + b significa una función que acepta dos parámetros llamados a y b. Tras la ejecución, evalúa la expresión a + b y devuelve el resultado.

Si solo tenemos un argumento, se pueden omitir paréntesis alrededor de los parámetros, lo que lo hace aún más corto.

```
let double = n => n * 2;

// Más o menos lo mismo que: let double = function(n) { return n * 2 }

console.log(double(3)); // 6
```

Si no hay parámetros, los paréntesis estarán vacíos (pero deben estar presentes):

```
let saludar = () => console.log("Hola!");

saludar();
```

Este tema lo hemos visto para que sepan que existe y si lo encuentran en alguna ocasión sepan identificarlo y cómo funciona. Recomendamos que antes de pasar a las funciones flecha, aprendamos a trabajar las funciones "normales".

# OBJETOS

Vamos a introducirnos en un tema muy importante de Javascript como son los objetos. Javascript no es un lenguaje de programación orientado a objetos **puro** porque, aunque utiliza objetos en muchas ocasiones, no necesitamos programar todos nuestros programas en base a ellos. De hecho, lo que vamos a hacer generalmente con Javascript es **usar objetos** y no tanto programar orientado a objetos. Por ello, lo que hemos visto hasta aquí relativo a sintaxis, sigue siendo perfectamente válido y puede ser utilizado igual que se ha indicado. Solo vamos a **aprender** una especie de estructura nueva como son los **objetos.**

**Un objeto es una colección de propiedades** y una propiedad es una asociación entre un **nombre (o clave)** y **un valor**. El valor de **una propiedad puede ser una función**, en cuyo caso **la propiedad se conoce como método**.

Las claves de un objeto, **solo pueden ser de tipo String**, no podemos ponerle de nombre a una propiedad un numero.

En lenguajes de programación orientados a objetos puros, como puede ser Java, tienes que programar siempre en **base a objetos**. Para programar tendrías que crear "clases", que son a partir de los cuales se crean objetos. El programa resolvería cualquier necesidad mediante la creación de objetos en base de nuestras clases, existiendo varios objetos de diversas clases. Los objetos tendrían que colaborar entre si para resolver cualquier tipo de acción, igual que en sistemas como un avión existen diversos objetos (el motor, hélices, mandos...) que colaboran entre sí para resolver la necesidad de llevar pasajeros.

Sin embargo, como dijimos previamente, en JavaScript no es tanto programar orientado a objetos, **sino usar objetos**. Muchas veces serán objetos ya creados por el propio navegador (la ventana del navegador, una imagen o un formulario HTML, etc), y otras veces serán objetos creados por ti mismo o por otros desarrolladores. Por tanto, lo que nos interesa saber para comenzar es la sintaxis que necesitas para usar los objetos, básicamente acceder a sus propiedades y ejecutar sus métodos.

## OBJETOS Y PROPIEDADES

Una propiedad de un objeto se puede explicar como una **variable adjunta al objeto**. Las propiedades de un objeto básicamente son lo mismo que las variables comunes de JavaScript, excepto por el nexo con el objeto. Las propiedades de un objeto definen las características del objeto. Accedes a las propiedades de un objeto con una simple notación de puntos:

```
nombreObjeto.nombrePropiedad;
```

Como todas las variables de JavaScript, tanto el nombre del objeto (que puede ser una variable normal) como el nombre de la propiedad son **sensibles a mayúsculas y minúsculas**. Puedes definir propiedades asignándoles un valor. Por ejemplo, vamos a crear un objeto llamado miAuto y le vamos a asignar propiedades denominadas marca, modelo, y año de la siguiente manera:

```
var miAuto = new Object();
miAuto.marca = 'Ford';
miAuto.modelo = 'Mustang';
miAuto.anio = 1969;
```

El ejemplo anterior también se podría escribir usando un **iniciador de objeto**, que es una lista delimitada por comas de cero o más pares de nombres de propiedad y valores asociados de un objeto, encerrados entre llaves ({}):

```
var miAuto = {
marca : 'Ford',
modelo : 'Mustang',
anio : 1969
}
```

Las propiedades se separan por comas y se coloca siempre el nombre de la propiedad, el carácter ":" y luego el valor de la propiedad.

Las propiedades no asignadas de un objeto son **undefined** (y no null).

```
miAuto.color; // undefined
```

## USAR UNA FUNCIÓN CONSTRUCTORA

Como alternativa, puedes crear un objeto con estos dos pasos:

1. Definir el tipo de objeto escribiendo una **función constructora**. Existe una fuerte convención, con buena razón, para utilizar en mayúscula la letra inicial.
2. Crear una instancia del objeto con el operador **new**.

Para definir un tipo de objeto, crea una **función para el objeto** que especifique su nombre, propiedades y métodos. Por ejemplo, supongamos que deseas crear un tipo de objeto para autos. Quieres llamar Auto a este tipo de objeto, y deseas que tenga las siguientes propiedades: marca, modelo y año. Para ello, podrías escribir la siguiente función:

```
function Auto(marca,modelo,anio){
this.marca = marca;
this.modelo = modelo;
this.anio = anio;
}
```

Observa el uso de **this** para asignar valores a las propiedades del objeto en función de los valores pasados a la función.

Ahora puedes crear un objeto llamado **miAuto** de la siguiente manera:

```
var miAuto = new Auto('Ford','Mustang',1969);
```

Esta declaración crea miAuto y le asigna los valores especificados a sus propiedades. Entonces el valor de miAuto.marca es la cadena "Ford", para miAuto.anio es el número entero 1969, y así sucesivamente.

Puedes crear cualquier número de objetos Auto con las llamadas a new. Por ejemplo,

```
var auto1 = new Auto('Nissan', '300ZX', 1992);
var auto2 = new Auto('Mazda', 'Miata', 1990);
```

Un objeto puede tener una propiedad que en sí misma es otro objeto. Por ejemplo, supongamos que defines un objeto llamado Persona de la siguiente manera:

```
function Persona(nombre, edad, sexo) {
  this.nombre = nombre;
  this.edad = edad;
  this.sexo = sexo;
}
```

y luego instancias dos nuevos objetos persona de la siguiente manera:

```
var agus = new Persona('Agustina Gomez', 33, 'F');
var valen = new Persona('Valentin Perez', 39, 'M');
```

Entonces, puedes volver a escribir la definición de Auto para incluir una propiedad propietario que tomará el objeto persona, de la siguiente manera:

```
function Auto(marca,modelo,anio, propietario){
this.marca = marca;
this.modelo = modelo;
this.anio = anio;
this.propietario = propietario;
}
```

Para crear instancias de los nuevos objetos, utiliza lo siguiente:

```
var auto1 = new Auto('Nissan', '300ZX', 1992, agus);
var auto2 = new Auto('Mazda', 'Miata', 1990, valen);
```

Nota que en lugar de pasar un valor de cadena o entero cuando se crean los nuevos objetos, las declaraciones anteriores pasan al objetos agus y valen como argumentos para los propietarios. Si luego quieres averiguar el nombre del propietario del auto2, puedes acceder a la propiedad de la siguiente manera:

```
auto2.propietario.nombre;
```

Ten en cuenta que siempre se puede añadir una propiedad a un objeto previamente definido. Por ejemplo, la declaración:

```
auto1.color = 'negro';
```

## DEFINICION DE METODOS

Un método es una función asociada a un objeto, o, simplemente, un método es una propiedad de un objeto que es una función. Los métodos se definen normalmente como una función, con excepción de que tienen que ser asignados como la propiedad de un objeto. Un ejemplo puede ser:

```
nombreObjeto.nombreMetodo = nombreFuncion;
```

Ejemplo:

```
var miObj = {
  miMetodo: function(parametros) {
    // ...hacer algo
  }
};
```

Entonces puedes llamar al método en el contexto del objeto de la siguiente manera:

```
objeto.nombreMetodo(parametros);
```

Puedes definir métodos para un tipo de objeto incluyendo una definición del método en la función constructora del objeto. Podrías definir una función que muestre las propiedades de los objetos del tipo Car previamente definidas, por ejemplo:

```
function mostrarAuto() {
  var resultado = `Un hermoso ${this.year} ${this.make} ${this.model}`;
  console.log(resultado);
}
```

Puedes hacer de esta función un método de Car agregando la declaración a la definición del objeto:

```
this.mostrarAuto = mostrarAuto;
```

Por lo tanto, la definición completa de Car ahora se verá así:

```
function Auto(marca,modelo,anio, propietario){
this.marca = marca;
this.modelo = modelo;
this.anio = anio;
this.propietario = propietario;
this.mostrarAuto = mostrarAuto;
}
```

Entonces, podemos llamar al método mostrarAuto, de la siguiente manera:

```
auto1.mostrarAuto;
auto2.mostrarAuto;
```

# OBJETOS INCORPORADOS EN JAVASCRIPT

JavaScript define algunos objetos de forma nativa, por lo que pueden ser utilizados directamente por las aplicaciones sin tener que declararlos. Las clases que se encuentran disponibles de manera nativa en Javascript, y que vamos a ver a continuación, son las siguientes:

- String, para el trabajo con cadenas de caracteres.
- Date, para el trabajo con fechas.
- Math, para realizar funciones matemáticas.
- Number, para realizar algunas cosas con números
- Boolean, trabajo con boleanos.
- Array, trabajo con listas

# STRING

En JavaScript las variables de tipo texto son objetos de la clase String. Esto quiere decir que cada una de las variables que creamos de tipo texto tienen una serie de propiedades y métodos. Para crear un objeto de la clase String lo único que hay que hacer es asignar un texto a una variable.

## PROPIEDADES DE STRING

### Length

La clase String sólo tiene una propiedad: length, que guarda el número de caracteres del String. Por ejemplo:

```
let cadena = "Hola";
console.log(cadena.length); // 3
```

## MÉTODOS DE STRING

Los objetos de la clase String tienen una buena cantidad de métodos para realizar muchas cosas interesantes.

| Método | Descripción. |
|---|---|
| charAt(indice) | Devuelve el carácter que hay en la posición indicada como índice. Las posiciones de un String empiezan en 0. |
| toString() | Este método lo tienen todos los objetos y se usa para convertirlos en cadenas. |

| | |
|---|---|
| indexOf(carácter,desde) | Devuelve la posición de la primera vez que aparece el carácter indicado por parámetro en un String. Si no encuentra el carácter en el String devuelve -1. El segundo parámetro es opcional y sirve para indicar a partir de que posición se desea que empiece la búsqueda. |
| lastIndexOf(carácter,desde) | Busca la posición de un carácter exáctamente igual a como lo hace la función indexOf pero desde el final en lugar del principio. El segundo parámetro indica el número de caracteres desde donde se busca, igual que en indexOf. |
| toLowerCase() | Pone todas los caracteres de un string en minúsculas. |
| toUpperCase() | Pone todas los caracteres de un string en mayúsculas. |
| replace(substring_a_buscar,nuevoStr) | Sirve para reemplazar porciones del texto de un string por otro texto, por ejemplo, podríamos uilizarlo para reemplazar todas las apariciones del substring "xxx" por "yyy". El método no reemplaza en el string, sino que devuelve un resultante de hacer ese reemplazo. |
| substring(inicio,fin) | Devuelve el substring que empieza en el carácter de inicio y termina en el carácter de fin. Si intercambiamos los parámetros de inicio y fin también funciona. |

## MATH

La clase Math es una de las clases nativas de JavaScript. Proporciona los mecanismos para realizar operaciones matemáticas en JavaScript. Algunas operaciones se resuelven rápidamente con los operadores aritméticos que ya conocemos, como la multiplicación o la suma, pero hay una serie de operaciones matemáticas adicionales que se tienen que realizar usando la clase Math como pueden ser calcular un seno o hacer una raíz cuadrada.

## MÉTODOS DE MATH

Tenemos una serie de métodos para realizar operaciones matemáticas típicas, aunque un poco complejas. Para utilizar los métodos de la clase Math, la sintaxis será la siguiente:

```
Math.metodo;
```

| Método | Descripción. |
|---|---|
| abs() | Devuelve el valor absoluto de un número. El valor después de quitarle el signo. |
| ceil() | Devuelve el entero igual o inmediatamente siguiente de un número. Por ejemplo, ceil(3) vale 3, ceil(3.4) es 4. |
| exp() | Retorna el resultado de elevar el número E por un número. |
| floor() | Lo contrario de ceil(), pues devuelve un número igual o inmediatamente inferior. |
| max() | Retorna el mayor de 2 números. |
| min() | Retorna el menor de 2 números. |
| pow() | Recibe dos números como parámetros y devuelve el primer número elevado al segundo número. |
| random() | Devuelve un número aleatorio entre 0 y 1. |
| round() | Redondea al entero más próximo. |
| PI | No es método, es una propiedad, que nos permite tener el valor de PI |

# DATE

Sobre la clase **Date** recae todo el trabajo con fechas en JavaScript, como obtener una **fecha**, el **día, la hora actuales** y otras cosas. Para trabajar con fechas necesitamos instanciar un objeto de la **clase Date** y con él ya podemos realizar las operaciones que necesitemos.

Un objeto de la clase Date se puede crear de dos maneras distintas. Por un lado podemos crear el objeto con el día y hora actuales y por otro podemos crearlo con un día y hora distintos a los actuales. Esto depende de los parámetros que pasemos al construir los objetos.

Para crear un objeto fecha con el día y hora actuales colocamos los paréntesis vacíos al llamar al constructor de la clase Date.

```
miFecha = new Date()
```

Para crear un objeto fecha con un día y hora distintos de los actuales tenemos que indicar entre paréntesis el momento con que inicializar el objeto.

```
miFecha = new Date(año,mes,dia)
```

Los objetos de la clase Date no tienen propiedades pero si un montón de métodos, que vamos a detallar a continuación.

| Método | Descripción. |
|---|---|
| getDate() | Devuelve el día del mes. |
| getDay() | Devuelve el día de la semana. |
| getHours() | Retorna la hora. |
| getMinutes() | Devuelve los minutos. |
| getMonth() | Devuelve el mes (el mes que empieza en 0). |
| getFullYear() | Retorna el año con todos los dígitos. |
| setDate() | Actualiza el día del mes. |
| setMonth() | Cambia el mes (el mes empieza en 0). |
| setHours() | Actualiza la hora. |
| setMinutes() | Cambia los minutos. |
| setFullYear() | Cambia el año de la fecha al número que recibe por parámetro. El número se indica completo ej: 2005 o 1995. |

# ARRAYS

El objeto Array de JavaScript es un objeto global que es usado en la construcción de arrays. Los arrays son objetos similares a una lista cuyo prototipo proporciona métodos para efectuar operaciones de recorrido y de mutación. Tanto la longitud como el tipo de los elementos de un array son variables. Dado que la longitud de un array puede cambiar en cualquier momento, y los datos se pueden almacenar en ubicaciones no contiguas, no hay garantía de que los arrays de JavaScript sean densos; esto depende de cómo el programador elija usarlos. En general estas características son cómodas, pero si, en su caso particular, no resultan deseables, puede considerar el uso de arrays con tipo.

## DECLARACIÓN

Hay dos sintaxis para crear un array vacío:

```
let arr = new Array();
let arr = [];
```

Casi siempre se usa la segunda. Podemos suministrar elementos iniciales entre los corchetes:

```
let frutas = ["Manzana", "Naranja", "Uva"];
```

Los elementos del array están numerados comenzando desde cero.

Podemos obtener un elemento por su número entre corchetes:

```
let frutas = ["Manzana", "Naranja", "Uva"];
console.log(frutas[0]);
console.log(frutas[1]);
console.log(frutas[2]);
```

Podemos reemplazar un elemento:

```
frutas[2] = "Pera" // ["Manzana", "Naranja", "Pera"];
```

...o agregar uno nuevo al array:

```
frutas[3] = "Limon" // ["Manzana", "Naranja", "Pera", "Limon"];
```

Un array puede almacenar elementos de cualquier tipo.

```
let frutas = ["Manzana", 1, true, function()];
```

La cuenta total de elementos en el array es su longitud length:

```
let frutas = ["Manzana", "Naranja", "Uva"];
console.log(frutas.length); // 3
```

# BUCLES

Una de las formas más viejas de iterar los items de un array es el bucle for sobre sus índices:

```
let frutas = ["Manzana", "Naranja", "Uva"];

for (let i = 0; i < frutas.length; i++) {

  console.log( frutas[i] );

}
```

Pero existen otros bucles, para iterar Arrays que nos van a servir para distintas situaciones y es importante conocerlos todos.

## .FOREACH

El for each es un método incluido dentro de los datos del tipo array. Este método nos permite recorrer el array de principio a fin y ejecutar una función o sentencia sobre cada elemento dentro del array. La sintaxis es la siguiente:

```
array.forEach(function(valorActual, indice, array){}, thisValor);
```

Los parámetros con los que se manejara el método son los siguientes:

- function *(callback)*: Función a ejecutar por cada elemento, que recibe tres argumentos:
  - valorActual: El elemento actual siendo procesado en el array.
  - index *(opcional)*: El índice del elemento actual siendo procesado en el array.
  - array (*opcional)*: El vector en el que forEach() está siendo aplicado.
- thisValor *(opcional)*: Valor que se usará como this cuando se ejecute el callback.

forEach() ejecuta la función callback una vez por cada elemento presente en el array en orden ascendente.

## IMPRIMIR EL CONTENIDO DE UN ARRAY

Veamos un forEach que nos va a servir para mostrar los elementos de un array.

```
function mostrarElementosArray(elemento, indice, array) {
    console.log("a[" + indice + "] = " + elemento);
}
// Nótese que se evita el 2° índice ya que no hay ningún elemento en esa
posición del array
[2, 5, , 9].forEach(mostrarElementosArray);
// salida:
// a[0] = 2
// a[1] = 5
// a[2] = 9
```

Es este ejemplo hacemos la función por separado y despúes se la pasamos al forEach, veamos como sería todo juntos

```
let array = [2, 5, 9].
array.forEach(function mostrarElementosArray(elemento, indice, array) {
    console.log("a[" + indice + "] = " + elemento);
});
// salida:
// a[0] = 2
// a[1] = 5
// a[2] = 9
```

## FOR OF

El for of es un bucle que itera sobre un elemento, como por ejemplo un array, desde su inicio a fin, la particularidad del for of es que tomara cada uno de los elementos, y los almacenará en una variable temporal, nosotros usaremos esa variable temporal, por ejemplo para mostrar todos los elementos de nuestro array.

```
for (variable of objeto) {
  // sentencias
}
```

En el siguiente ejemplo que veremos, tendremos un array de frutas y los almacena en la variable temporal "fruta", posterior a esto podremos hacer lo que se desee.

```
let frutas = ["Manzana", "Naranja", "Uva"];
for(let fruta of frutas){
console.log(fruta); // ["Manzana", "Naranja", "Uva"];
}
```

Ahora supongamos que tenemos un array de objetos de empleados:

```
for (let empleado of empleados){
   console.log(empleado.nombre);
   console.log(empleado.apellido);
}
```

Nosotros usamos la variable temporal para acceder a cada propiedad de los objetos dentro del array.

## FOR IN

El for in a diferencia del for of iterara sobre los elementos dentro de un dato. Por ejemplo, si utilizamos un for in sobre un array, lo que hara será darnos los índices de los elementos dentro del array, pero si usamos un for in sobre un objeto, nos dara cada uno de los atributos dentro del mismo. En el siguiente ejemplo, lo que se hace es usar un for of para recorrer la lista de empleados, y luego se toma un empleado con el for in para recorrer cada una de las propiedades del empleado.

```
for (let empleado of empleados){
    for(let dato in empleado){
        console.log(empleado[dato]);
    }
}
```

Por lo que este bucle, por más que podríamos usarlo para arrays, está mayormente pensado para recorrer objetos.

## METODOS OBJETO ARRAY

El objeto Array de JavaScript cuenta con muchos métodos. Para hacer las cosas más sencillas, a la hora de trabajar con arrays, vamos a ver algunos en mayor detalle.

### splice()

¿Cómo podemos borrar un elemento de un array?

Los arrays son objetos, por lo que podemos intentar con delete:

```
let arr = ["voy", "a", "casa"];
delete arr[1]; // remueve "a"
console.log(arr[1]); // undefined
// ahora arr = ["voy",  , "casa"];
console.log(arr.length); // 3
```

El elemento fue borrado, pero el array todavía tiene 3 elementos, podemos ver que arr.length == 3.

Es natural, porque delete obj.key borra el valor de key, pero es todo lo que hace. Esto está bien en los objetos, pero en general lo que buscamos en los arrays es que el resto de los elementos se desplace y se ocupe el lugar libre. Lo que esperamos es un array más corto.

Por lo tanto, necesitamos utilizar métodos especiales.

El método arr.splice funciona como una navaja suiza para arrays. Puede hacer todo: insertar, remover y remplazar elementos.

```
arr.splice(inicio[, cantEliminar, elem1, ..., elemN])
```

Esto modifica arr comenzando en el índice **inicio**: remueve la cantidad **cantEliminar** de elementos y luego inserta **elem1, ..., elemN** en su lugar. Lo que devuelve es un array de los elementos removidos.

```
let arr = ["Yo", "estudio", "JavaScript"];

arr.splice(1, 1); // desde el índice 1, remover 1 elemento

console.log(arr); // ["Yo", "JavaScript"]
```

Empezando desde el índice 1 removió 1 elemento.

En el próximo ejemplo removemos 3 elementos y los reemplazamos con otros 2:

```
let arr = ["Yo", "estudio", "JavaScript", "ahora", "mismo"];

// remueve los primeros 3 elementos y los reemplaza con otros

arr.splice(0, 3, "a", "bailar");

console.log(arr) // ahora ["a", "bailar", "ahora", "mismo"]
```

### slice()

El método **slice()** devuelve una copia de una parte del array dentro de un nuevo array empezando desde **inicio** hasta **fin** (fin no incluido). El array original no se modificará. La sintaxis es la siguiente:

```
arr.slice([inicio], [fin]);
```

Por ejemplo:

```
let arr = ["t", "e", "s", "t"];

console.log( arr.slice(1, 3) ); // e,s (copia desde 1 hasta 3)
```

### split()

Analicemos una situación de la vida real. Estamos programando una app de mensajería y el usuario ingresa una lista de receptores delimitada por comas: Celina, David, Federico. Pero para nosotros un array sería mucho más práctico que una simple string. ¿Cómo podemos hacer para obtener un array?

El método **str.split(delim)** hace precisamente eso. Separa la string en elementos según el delimitante delim dado y los devuelve como un array.

En el ejemplo de abajo, separamos por "coma seguida de espacio":

```
let nombres = 'Bilbo, Gandalf, Nazgul';

let arr = nombres.split(', ');

for (let name of arr) {

console.log( `Un mensaje para ${name}.` ); // Un mensaje para Bilbo y los
                                            // otros nombres
}
```

### reverse()

El método arr.reverse revierte el orden de los elementos en arr.

Por ejemplo:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();
console.log( arr ); // 5,4,3,2,1
```

### sort(fn)

Cuando usamos arr.sort(), este ordena el propio array cambiando el orden de los elementos.

También devuelve un nuevo array ordenado pero éste usualmente se descarta ya que arr en sí mismo es modificado.

Por ejemplo:

```
let arr = [ 1, 2, 15 ];
// el método reordena el contenido de arr
arr.sort();
console.log( arr );  // 1, 15, 2
```

Los elementos fueron reordenados a 1, 15, 2. Pero ¿por qué pasa esto?

Los elementos son ordenados como Strings (cadenas de caracteres) por defecto

Todos los elementos son convertidos a String para ser comparados. En el caso de Strings se aplica el orden lexicográfico, por lo que efectivamente "2" > "15".

Para usar nuestro propio criterio de reordenamiento, necesitamos proporcionar una función como argumento de arr.sort().

La función debe comparar dos valores arbitrarios y devolver el resultado, sería parecido al comparator que conocemos en Java:

```
function compare(a, b) {
  if (a > b) return 1; // si el primer valor es mayor que el segundo
  if (a == b) return 0; // si ambos valores son iguales
  if (a < b) return -1; // si el primer valor es menor que el segundo
}
```

Por ejemplo, para ordenar como números:

```
function compareNumeric(a, b) {
 if (a > b) return 1;
 if (a == b) return 0;
 if (a < b) return -1;
 }
let arr = [ 1, 15, 2 ];
arr.sort(compareNumeric);
console.log(arr); // 1, 2, 15
```

### map()

El método map() crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. La sintaxis es:

```
let result = arr.map(function(elemento, indice, array) {
  // devuelve el nuevo valor en lugar de item
});
```

Por ejemplo, acá transformamos cada elemento en el valor de su respectivo largo (length):

```
let longitudes = ["Bilbo", "Gandalf", "Nazgul"].map(function(elemento){
elemento.length();
});
console.log(longitudes); // 5,7,6
```

### flat()

El método flat() crea una nueva matriz con todos los elementos de sub-array concatenados recursivamente hasta la profundidad especificada.

```
var nuevoArray = arr.flat([profundidad]);
```

El método tiene un solo parámetro que es, depth, este parámetro es opcional y especifica qué tan profunda debe aplanarse una estructura de matriz anidada. El valor predeterminado es 1.

Ejemplos:

```
var arr1 = [1, 2, [3, 4]];
arr1.flat();
// [1, 2, 3, 4]
var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();
// [1, 2, 3, 4, [5, 6]]
```

### flatMap()

El método flatMap() devuelve un nuevo array formado al aplicar una función de devolución de llamada determinada a cada elemento de la matriz y luego aplanar el resultado en un nivel. Es idéntico a un map() seguido de un flat() de profundidad 1, pero un poco más eficiente que llamar a esos dos métodos por separado.

```
var arrayNuevo = array.flatMap(function(elemento, indice, array){
// retorna elementos para el nuevo array
});
```

Ejemplos:

```
var arr1 = [1, 2, 3, 4];
var arr2 = arr1.map(function(x){
x = x * 2;
});
console.log(arr2) // [[2], [4], [6], [8]]
var arr2 = arr1.flatMap(function(x){
x = x * 2;
});
console.log(arr2) // [2, 4, 6, 8]
```

## METODOS EXTRAS DE ARRAY

Vamos a mostrar otros métodos más de la clase Array

| Método | Descripción | Ejemplo |
|---|---|---|
| concat() | Une dos o más arrays | `var arraytotal = array1.concat(array2);` |
| join() | Junta los elementos de un array en una cadena con un separador – opcional. | `var fruta = [“Kiwi”, “Limon”, “Otra”];`<br>`var ej = frutas.join();`<br>`// Kiwi, Limon, Otra` |
| pop() | Borra el último elemento del array y devuelve su contenido | `var a = frutas.pop();` |
| push() | Añade nuevos elementos al array y devuelve su nueva longitud | `var a = frutas.push(“Uva”);` |

| | | |
|---|---|---|
| shift() | Elimina el primer elemento del array y devuelve el elemento | `var a = frutas.shift();` |
| find() | El método find() devuelve el valor del primer elemento del array que cumple la función de prueba proporcionada. | `const array1 = [5, 12, 8, 130, 44];`<br>`const encontrado = array1.find(elemento => elemento > 10);`<br>`console.log(encontrado);`<br>`// resultado: 12` |
| unshift() | Añade elementos al inicio del array y devuelve la nueva longitud | `var frutas = ["Banana", "Naranja", "Manzana"];`<br>`frutas.unshift("Limon","Anana");`<br>`//Limon, Anana, Banana, "Naranja, Manzana` |

# OBJETOS MAP Y SET

Hasta este momento, hemos aprendido sobre las siguientes estructuras de datos:

- Objetos para almacenar colecciones de datos ordenadas mediante una clave.
- Arrays para almacenar colecciones ordenadas de datos.

Pero eso no es suficiente para la vida real. Por eso también existen Map y Set.

## MAP

Map es, al igual que Object, una colección de datos identificados por claves. Pero la principal diferencia es que Map, permite claves de cualquier tipo.

Los métodos y propiedades son:

- **new Map():** crea el mapa.
- **map.set(clave, valor):** almacena el valor asociado a la clave.
- **map.get(clave):** devuelve el valor de la clave. Será undefined si la clave no existe en map.
- **map.has(clave):** devuelve true si la clave existe en map, false si no existe.
- **map.delete(clave):** elimina el valor de la clave.
- **map.clear():** elimina todo de map.
- **map.size:** tamaño, devuelve la cantidad actual de elementos.

Por ejemplo:

```
let map = new Map();

map.set('1', 'str1');   // un string como clave

map.set(1, 'num1');     // un número como clave

map.set(true, 'bool1'); // un booleano como clave

// ¿recuerda el objeto regular? convertiría las claves a string.

// Map mantiene el tipo de dato en las claves, por lo que estas dos son
// diferentes:

console.log( map.get(1)   ); // 'num1'

console.log( map.get('1') ); // 'str1'

console.log( map.size ); // 3
```

Podemos ver que, a diferencia de los objetos, las claves no se convierten en strings. Cualquier tipo de clave es posible en un Map.

También podemos usar objetos como claves.

Por ejemplo:

```
let john = { name: "John" };

// para cada usuario, almacenemos el recuento de visitas

let contadorVisitasMap = new Map();

// john es la clave para el Map

contadorVisitas.set(john, 123);

console.log(contadorVisitas.get(john)); // 123
```

El uso de objetos como claves es una de las características de Map más notables e importantes. Esto no se aplica a los objetos: una clave de tipo String está bien en un Object, pero no podemos usar otro Object como clave.

## ITERACIÓN SOBRE MAP

Para recorrer un map, hay 3 métodos:

- map.keys(): devuelve un iterable para las claves.
- map.values(): devuelve un iterable para los valores.
- map.entries(): devuelve un iterable para las entradas [clave, valor]. Es el que usa por defecto en for..of.

Por ejemplo:

```
let recetaMap = new Map([
  ['pepino', 500],
  ['tomates', 350],
  ['cebollas', 50]
]);
// iterando sobre las claves (verduras)
for (let vegetales of recetaMap.keys()) {
  console.log(vegetales); // pepino, tomates, cebollas
}
// iterando sobre los valores (precios)
for (let cantidad of recetaMap.values()) {
  console.log(cantidad); // 500, 350, 50
}
// iterando sobre las entradas [clave, valor]
for (let entry of recetaMap) { // lo mismo que recipeMap.entries()
  console.log(entry); // pepino,500 (etc)
}
```

## SET

Un Set es una colección de tipo especial: "conjunto de valores" (sin claves), donde cada valor puede aparecer solo una vez.

Sus principales métodos son:

- new Set(iterable): crea el set. El argumento opcional es un objeto iterable (generalmente un array) con valores para inicializarlo.
- set.add(valor): agrega un valor, y devuelve el set en sí.
- set.delete(valor): elimina el valor, y devuelve true si el valor existía al momento de la llamada; si no, devuelve false.
- set.has(valor):devuelve true si el valor existe en el set, si no, devuelve false.
- set.clear(): elimina todo el contenido del set.
- set.size: es la cantidad de elementos.

La característica principal es que llamadas repetidas de set.add(valor) con el mismo valor no hacen nada. Esa es la razón por la cual cada valor aparece en Set solo una vez.

```
let setNombres = new Set();
let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };
// visitas, algunos usuarios lo hacen varias veces
setNombres.add(john);
setNombres.add(pete);
setNombres.add(mary);
setNombres.add(john);
setNombres.add(mary);
// set solo guarda valores únicos
console.log(set.size); // 3
for (let usuario of setNombres) {
  console.log(usuario.name); // John (luego Pete y Mary)
}
```

### ITERACIÓN SOBRE SET

Podemos recorrer Set con for..of o usando forEach:

```
let setFrutas = new Set(["naranjas", "manzanas", "uvas"]);
for (let valor of setFrutas){
 console.log(valor); // "naranjas", "manzanas", "uvas"
}
// lo mismo que forEach:
setFrutas.forEach((valor, valorDeNuevo, setFrutas) => {
  console.log(valor); // "naranjas", "manzanas", "uvas"
});
```

## JSON

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. JSON es un formato que almacena información estructurada y se utiliza principalmente para transferir datos entre un servidor y un cliente.

Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, Standard ECMA-262. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, Java, JavaScript, y muchos otros. Este formato funcionan bien para lograr la carga asincrónica de los datos almacenados, lo que significa que un sitio web puede actualizar su información sin actualizar la página.

## SINTAXIS JSON

Para crear correctamente un archivo .json, debes seguir la sintaxis correcta.

JSON está constituido por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o sequencias.

### Objetos

Hay dos elementos centrales en un objeto JSON: **claves** (Keys) y **valores** (Values).

- Las **Keys** deben ser cadenas de caracteres (strings). Como su nombre en español lo indica, estas contienen una secuencia de caracteres rodeados de comillas.
- Los **Values** son un tipo de datos JSON válido. Puede tener la forma de un arreglo (array), objeto, cadena (string), booleano, número o nulo.

Un **objeto JSON** comienza y termina con **llaves {}**. Puede tener **dos o más pares** de **claves/valor** dentro, con **una coma para separarlos**. Así mismo, cada **key es seguida por dos puntos** para distinguirla del valor.

Ejemplo:

```
{"ciudad":"Nueva York", "país":"Estados Unidos"}
```

Aquí tenemos dos pares de clave/valor: **ciudad y país son las claves**, **Nueva York y Estados Unidos son los valores**. Los valores en este ejemplo, son Strings. Por eso también están entre comillas, similares a las claves.

### Array

Un valor de un array puede contener objetos JSON, lo que significa que utiliza el mismo concepto de par clave/valor. Por ejemplo:

```
"estudiantes": [
{"primerNombre":"Tom", "Apellido":"Jackson"},
{"primerNombre":"Linda", "Apellido":"Garner"},
{"primerNombre":"Adam", "Apellido":"Cooper"}
]
```

En este caso, la información entre corchetes es un array, que tiene tres objetos.

## METODOS JSON

Si analizamos bien la sintaxis de un JSON, nos daremos cuenta que es muy similar a un objeto de Javascript y que no debería ser muy difícil pasar de JSON a JavaScript y viceversa.

En Javascript tenemos una serie de métodos que nos facilitan esa tarea, pudiendo trabajar con Strings que contengan JSON y objetos Javascript de forma indiferente:

### Convertir JSON a Objeto

La acción de convertir JSON a objeto JavaScript se le suele denominar parsear. Es una acción que analiza un **String que contiene un JSON** válido y devuelve **un objeto JavaScript** con dicha información correctamente estructurada. Para ello, utilizaremos el método **JSON.parse()**:

```
const str = '{ "name": "Manz", "life": 99 }';

const obj = JSON.parse(str);

obj.name; // 'Manz'

obj.life; // 99
```

Como se puede ver, **obj** es un **objeto** generado a partir del **JSON** recogido en la variable str y podemos consultar sus propiedades y trabajar con ellas sin problemas.

### Convertir Objeto a JSON

La acción inversa, convertir un **objeto Javascript a JSON** también se puede realizar fácilmente haciendo uso del método **JSON.stringify()**. Este método difícil de pronunciar viene a ser algo así como **"convertir a texto"**, y lo podemos utilizar para transformar un **objeto de Javascript a JSON** rápidamente;

```
const obj = {
  name: "Manz",
  life: 99,

  saludar: function () {
    return "Hola!";
  },
};
const str = JSON.stringify(obj);

console.log(str); // '{"name":"Manz","life":99}'
```

Las **funciones no están soportadas por JSON**, por lo que si intentamos convertir un objeto que contiene métodos o funciones, JSON.stringify() no fallará, pero simplemente devolverá un String **omitiendo las propiedades que contengan funciones**.

# MANEJO DE ERRORES

La sentencia try consiste en un bloque "try" que contiene una o más sentencias. Las llaves {} se deben utilizar siempre. incluso para una bloques de una sola sentencia. Al menos un bloque "catch" o un bloque "finally" debe estar presente. Esto nos da tres formas posibles para la sentencia "try":

1. try...catch
2. try...finally
3. try...catch...finally

Un bloque "catch" contiene sentencias que especifican que hacer si una excepción es lanzada en el bloque "try". Si cualquier sentencia dentro del bloque "try" (o en una función llamada desde dentro del bloque "try") lanza una excepción, el control cambia inmediatamente al bloque "catch". Si no se lanza ninguna excepción en el bloque "try", el bloque "catch" se omite.

La bloque "finally" se ejecuta despues del bloque "try" y el/los bloque(s) "catch" hayan finalizado su ejecución. Éste bloque siempre se ejecuta, independientemente de si una excepción fue lanzada o capturada.

Puede anidar una o más sentencias "try". Si una sentencia "try" interna no tiene una bloque "catch", se ejecuta el bloque "catch" de la sentencia "try" que la encierra.

Usted también puede usar la declaración "try" para manejar excepciones de JavaScript.

Cuando solo se utiliza un bloque "catch", el bloque "catch" es ejecutado cuando cualquier excepción es lanzada. Por ejemplo, cuando la excepción ocurre en el siguiente código, el control se transfiere a la cláusula "catch".

```
try {
   throw "miExcepcion"; // genera una excepción
}catch(e) {
   // sentencias para manejar cualquier excepción
   logMyErrors(e); // pasa el objeto de la excepción al manejador de
                   //errores
}
function isValidJSON(text) {
  try {
    JSON.parse(text);
    return true;
  } catch {
    return false; }}
```

También se pueden crear "bloques “catch” condicionales", combinando bloques try…catch con estructuras “if…else” como estas:

```
try {
    miRutina();  // puede lanzar tres tipos de excepciones
} catch (e) {
    if (e instanceof TypeError) {
        // sentencias para manejar excepciones TypeError
    } else if (e instanceof RangeError) {
        // sentencias para manejar excepciones RangeError
    } else if (e instanceof EvalError) {
        // sentencias para manejar excepciones EvalError
    } else {
       // sentencias para manejar cualquier excepción no especificada
       logMyErrors(e); // pasa el objeto de la excepción al manejador de
                       // errores
}
```

El siguiente ejemplo abre un archivo y después ejecuta sentencias que usan el archivo (JavaScript del lado del servidor permite acceder a archivos). Si una excepción es lanzada mientras el archivo está abierto, la cláusula “finally” cierra el archivo antes de que el script falle. El código en “finally” también se ejecuta después de un retorno explícito de los bloques “try” o “catch”.

```
openMyFile(){
    try {
       // retiene un recurso
       escribirMyArchivo(informacion);
    } catch (e) {
       // sentencias para manejar cualquier excepción
       logMyErrors(e); // pasa el objeto de la excepción al manejador de
                       // errores
    } finally {
       cerrarMiArchivo(); // siempre cierra el recurso
    }
}
```

# INTRODUCCIÓN A DOM

Cuando aprendimos HTML/CSS, nos dimos cuenta que, sólo podremos crear páginas "estáticas" (sin demasiada personalización por parte del usuario), pero si añadimos Javascript, podremos crear páginas "dinámicas". Cuando hablamos de páginas dinámicas, nos referimos a que podemos dotar de la potencia y flexibilidad que nos da un lenguaje de programación para crear documentos y páginas mucho más ricas, que brinden una experiencia más completa y con el que se puedan automatizar un gran abanico de tareas y acciones.

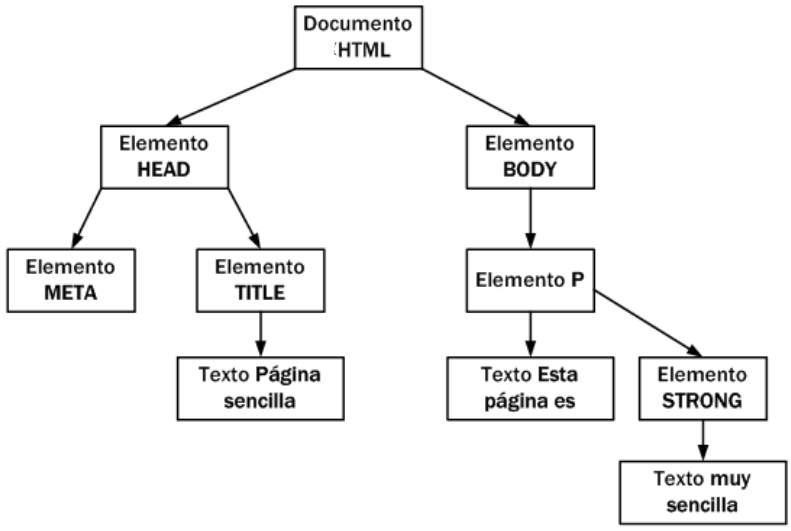
# ¿QUÉ ES EL DOM?

Las siglas DOM significan Document Object Model, o lo que es lo mismo, la estructura del documento HTML. Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol DOM (o simplemente DOM).

Supongamos que tenemos a siguiente pagina:

```
<!DOCTYPE>
<html>
  <head>
    <meta/>
    <title>Página sencilla</title>
  </head>
  <body>
    <p>Esta página es <strong>muy sencilla</strong></p>
  </body>
</html>
```

El árbol de etiquetas que nos mostraría el DOM, sería el siguiente:



En Javascript, cuando nos referimos al DOM nos referimos a esta estructura, que podemos modificar de forma dinámica desde Javascript, añadiendo nuevas etiquetas, modificando o eliminando otras, cambiando sus atributos HTML, añadiendo clases, cambiando el contenido de texto, etc..

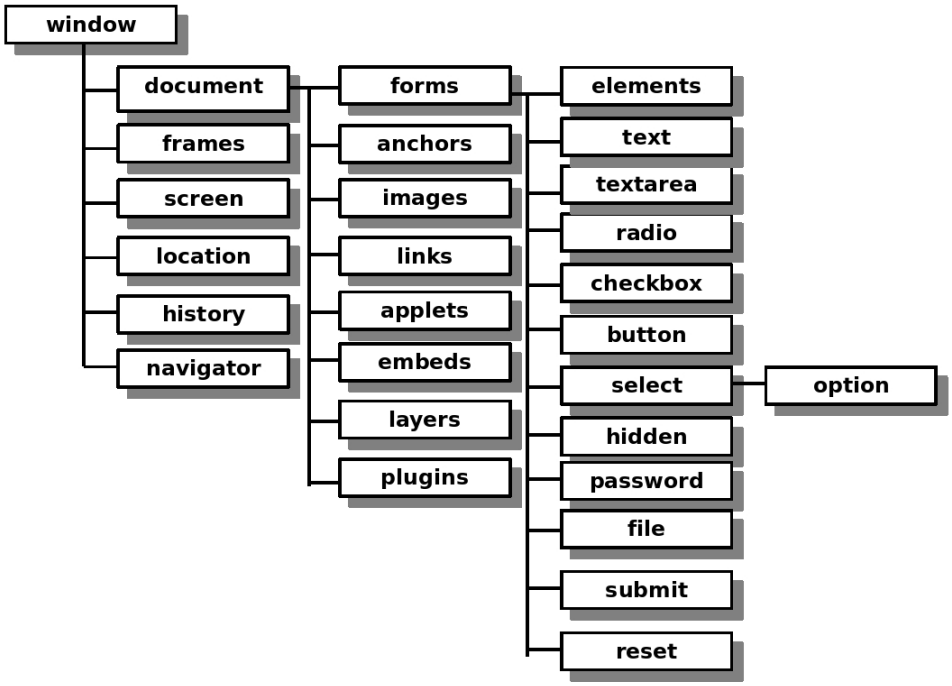
Al estar "amparado" por un lenguaje de programación, todas estas tareas se pueden automatizar, incluso indicando que se realicen cuando el usuario haga acciones determinadas, como por ejemplo: pulsar un botón, mover el ratón, hacer click en una parte del documento, escribir un texto, etc...

## ¿COMO PODEMOS ACCEDER A LAS ETIQUETAS?

Cuando se carga una página, el navegador crea una jerarquía de objetos en memoria que sirven para controlar los distintos elementos de dicha página. Con Javascript y la nomenclatura de objetos que hemos aprendido, podemos trabajar con esa jerarquía de objetos, acceder a sus propiedades e invocar sus métodos.

Cualquier elemento de la página se puede controlar de una manera u otra accediendo a esa jerarquía. Es crucial conocerla bien para poder controlar perfectamente las páginas web con Javascript o cualquier otro lenguaje de programación del lado del cliente.

Esta jerarquía de objetos está compuesta de la siguiente manera:




Como se puede apreciar, todos los objetos comienzan en un objeto que se llama window. Este objeto ofrece una serie de métodos y propiedades para controlar la ventana del navegador. Con ellos podemos controlar el aspecto de la ventana, la barra de estado, abrir ventanas secundarias y otras cosas

Además de ofrecer control, el objeto window da acceso a otros objetos como el **document** (La página web que se está visualizando), el historial de páginas visitadas o los distintos frames de la ventana. De modo que para acceder a cualquier otro objeto de la jerarquía deberíamos empezar por el objeto window. Tanto es así que JavaScript **entiende perfectamente que la jerarquía empieza en window aunque no lo señalemos.**

# OBJETO WINDOW

Vamos a ver algunos de los métodos de la clase window:

| Método | Descripción. |
|---|---|
| alert(texto) | Presenta una ventana de alerta donde se puede leer el texto que recibe por parámetro |
| back() | Ir una página atrás en el historial de páginas visitadas. |
| captureEvents(eventos) | Captura los eventos que se indiquen por parámetro |
| close() | Cierra la ventana |
| confirm(texto) | Muestra una ventana de confirmación y permite aceptar o rechazar. |
| find() | Muestra una ventanita de búsqueda. |
| home() | Ir a la página de inicio que haya configurada en el explorador. |
| prompt(pregunta,inicializacionDeLaRespuesta) | Muestra una caja de diálogo para pedir un dato. Devuelve el dato que se ha escrito. |
| setInterval() | Define un script para que sea ejecutado indefinidamente en cada intervalo de tiempo. |

| | |
|---|---|
| setTimeout(sentencia,milisegundos) | Define un script para que sea ejecutado una vez después de un tiempo de espera determinado. |
| clearInterval() | Elimina la ejecución de sentencias asociadas a un intervalo indicadas con el método setInterval(). |
| cleatTimeout() | Elimina la ejecución de sentencias asociadas a un tiempo de espera indicadas con el método setTimeout(). |

Estos son solo algunos de los métodos del objeto Window, para conocer más sobre este objeto, sus propiedades y sus métodos. **Les recomendamos entrar al siguiente link:** ObjetoWindow.

# EL OBJETO DOCUMENT

En Javascript, la forma de acceder al **DOM** es a través de un objeto llamado **document**, que representa el árbol DOM de la página o pestaña del navegador donde nos encontramos. En su interior pueden existir varios tipos de elementos, pero principalmente serán o **Element** o **Node**:

- **Element** no es más que la representación genérica de una **etiqueta: HTMLElement**.
- **Node** es una unidad más básica, la cuál puede ser **Element** o un **nodo de texto**.

## SELECCIONAR ELEMENTOS DEL DOM

Si nos encontramos en nuestro código Javascript y queremos hacer modificaciones en un elemento de la página HTML, lo primero que debemos hacer es buscar dicho elemento. Para ello, se suele intentar identificar el elemento a través de el **nombre de la etiqueta** o de alguno de sus atributos más utilizados, generalmente **el id** o la **clase**.

### MÉTODOS TRADICIONALES

Existen varios métodos, los más clásicos y tradicionales para realizar búsquedas de elementos en el documento. Observa que si lo que buscas es un elemento específico, lo mejor sería utilizar getElementById(), en caso contrario, si utilizamos uno de los 3 siguientes métodos, nos devolverá un Array donde tendremos que elegir el elemento en cuestión posteriormente:

| Método | Descripción. |
| --- | --- |
| .getElementById(id) | Busca el elemento HTML con el id id. Si no, devuelve null. |
| .getElementsByClassName(class) | Busca elementos con la clase class. Si no, devuelve []. |
| .getElementsByName(name) | Busca elementos con atributo name name. Si no, devuelve []. |
| .getElementsByTagName(tag) | Cierra la ventana |

Estos son los **4 métodos tradicionales** de Javascript para manipular el DOM. Se denominan tradicionales porque son los que existen en Javascript desde versiones más antiguas. Dichos métodos te permiten buscar elementos en la página dependiendo de los atributos **id, class, name** o **de la propia etiqueta**, respectivamente.

### getElementById()

El primer método, .getElementById(id) busca un elemento HTML con el **id** especificado en **id por parámetro**. En principio, un documento HTML bien construído **no debería** tener más de un elemento con el mismo id, por lo tanto, este método devolverá siempre un solo elemento:

```
const page = document.getElementById("page"); // <div id="page"></div>
```

### getElementsByClassName()

Por otro lado, el método **.getElementsByClassName(class)** permite buscar los elementos con **la clase especificada en class**. Es importante darse cuenta del matiz de que el método tiene getElements **en plural**, y esto es porque **al devolver clases** (al contrario que los id) **se pueden repetir**, y por lo tanto, **devolvernos varios elementos, no sólo uno.**

```
const items = document.getElementsByClassName("item");  // [div, div, div]
console.log(items[0]);      // Primer item encontrado:
                            // class="item"></div>

console.log(items.length);  // 3
```

### getElementsByTagName()

Esta función nos permite obtener todos los elementos **cuya etiqueta sea igual al parámetro** que le pasamos a la función. Por ejemplo, para seleccionar todos los `h1` de una página html.

```
const items = document.getElementsByTagName("h1");  // [h1, h1]
console.log(items[0]);       // Primer item encontrado:
                            // <h1></h1>

console.log(items.length);  // 2
```

Una manera menos tradicional pero que pueden llegar a ver es, seleccionar la etiqueta por su nombre sin usar ningún método, por ejemplo:

```
const items = document.h1;
```

Esto no puede traer muchas etiquetas o una sola como en el caso de body.

## ATRIBUTOS DE LOS ELEMENTOS

De la misma manera que podemos acceder a los elementos del HTML, también podemos acceder a sus atributos, podemos ponerle nuevos valores a sus atributos, removerlos o validar si están. Esto se hacer con los siguiente métodos:

| Método | Descripción. |
|---|---|
| .getAttribute(atributo) | Devuelve el valor del atributo especificado en el elemento |
| getAttributeNames() | Devuelve un Array con los atributos del elemento |
| .setAttribute(atributo, valor) | Establece el valor de un atributo en el elemento indicado. Si el atributo ya existe, el valor es actualizado, en caso contrario, el nuevo atributo es añadido con el nombre y valor indicado. |
| hasAttributes() | Indica si el elemento tiene atributos HTML. |
| .hasAttribute(atributo) | El método hasAttribute() devuelve un valor Booleano indicando si el elemento tiene el atributo especificado o no. |
| .removeAttribute(atributo) | removeAttribute elimina un atributo del elemento especificado. |

Estos métodos son bastante autoexplicativos y fáciles de entender, aún así, vamos a ver unos ejemplos de uso donde podemos ver como funcionan:

```
// Obtenemos <div id="page" class="info data dark" data-number="5"></div>

const div = document.getElementById("page");

div.hasAttribute("data-number");  // true (data-number existe)
div.hasAttributes();              // true (tiene 3 atributos)

div.getAttributeNames();          // ["id", "data-number", "class"]
div.getAttribute("id");           // "page"

div.removeAttribute("id");        // class="info data dark" y data-
                                 // number="5"

div.setAttribute("id", "page");   // Vuelve a añadir id="page"
```

### getAttribute()

Vamos a ver en mayor profundidad este método, recordemos que el metodo .getAttribute() devuelve el valor del atributo especificado en el elemento. Si el atributo especificado no existe, el valor retornado puede ser tanto null como "" (una cadena vacía).

```
var div1 = document.getElementById("div1");

var align = div1.getAttribute("align");

alert(align); // Muestra el valor de la alineación(align) del elemento con
              // id="div1"
```

Otro ejemplo sería si queremos validar que el input de un formulario no está vacío, usaríamos el atributo value:

```
<input type=”text” id=”inputNombre” value=” ”>

var nombre = document.getElementById(“inputNombre”).getAttribute(“value”);

if(nombre == “ ”){

alert(“El nombre está vacio”);

}
```

De la misma manera que vimos que podemos acceder a los elementos por los nombres, también podemos hacerlo con los atributos, por ejemplo:

```
var nombre = input.value;
```

Siempre es mejor hacer uso de los métodos que nos da el DOM, ya que facilita la lectura.

## MODIFICAR ELEMENTOS DEL DOM

En este artículo vamos a centrarnos en tres categorías:

- Reemplazar contenido de elementos en el DOM
- Insertar elementos en el DOM
- Eliminar elementos del DOM

## REEMPLAZAR CONTENIDO

Comenzaremos por la familia de propiedades siguientes, que enmarcamos dentro de la categoría de **reemplazar contenido** de elementos HTML. Se trata de una vía rápida con la cuál podemos añadir (*o más bien, reemplazar*) el contenido de una etiqueta HTML.

| Método | Descripción. |
|---|---|
| .nodeName | Devuelve el nombre del nodo (etiqueta si es un elemento HTML). Sólo lectura. |
| .textContent | Devuelve el contenido de texto del elemento. Se puede asignar para modificar. |
| .innerHTML | Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar. |
| .outerHTML | Idem a .innerHTML pero incluyendo el HTML del propio elemento HTML. |
| .innerText | Versión no estándar de .textContent de Internet Explorer con diferencias. Evitar. |
| .outerText | Versión no estándar de .textContent/.outerHTML de Internet Explorer. Evitar. |

## LA PROPIEDAD TEXTCONTENT

La propiedad .textContent nos devuelve el **contenido de texto** de un elemento HTML. Es útil para obtener *(o modificar)* **sólo el texto** dentro de un elemento, obviando el etiquetado HTML:

```
const div = document.getElementById("div");  // <div></div>
div.textContent = "Hola a todos"; // <div>Hola a todos</div>
div.textContent;  // "Hola a todos"
```

Observa que también podemos utilizarlo para **reemplazar el contenido de texto**, asignándolo como si fuera una variable o constante.

## LA PROPIEDAD INNERHTML

Por otro lado, la propiedad .innerHTML nos permite hacer lo mismo, pero interpretando el código HTML **indicado y renderizando sus elementos**:

```
const div = document.getElementById("info"); // <div class="info"></div>
div.innerHTML = "<strong>Importante</strong>"; // Interpreta el HTML
div.innerHTML;  // "<strong>Importante</strong>"
div.textContent;  // "Importante"
div.textContent = "<strong>Importante</strong>";  // No interpreta el HTML
```

Observa que la diferencia principal entre .innerHTML y .textContent es que el primero renderiza e interpreta el marcado HTML, mientras que el segundo lo inserta como contenido de texto literalmente.

## INSERTAR ELEMENTOS

Hemos visto, como reemplazar contenido y como seleccionar elementos, pero no hemos visto como añadir los elementos al documento HTML actual (conectarlos al DOM), operación que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

| Método | Descripción. |
|---|---|
| .appendChild(node) | Añade como hijo el nodo node. Devuelve el nodo insertado. |
| .insertAdjacentElement(pos, elem) | Inserta el elemento elem en la posición pos. Si falla, null. |
| .insertAdjacentHTML(pos, str) | Inserta el código HTML str en la posición pos. |
| .insertAdjacentText(pos, text) | Inserta el texto text en la posición pos. |
| .insertBefore(new, node) | Inserta el nodo new antes de node y como hijo del nodo actual. |
| .appendChild(node) | Añade como hijo el nodo node. Devuelve el nodo insertado. |

## EL MÉTODO APPENDCHILD()

Uno de los métodos más comunes para añadir un elemento HTML creado con Javascript es appendChild(). Como su propio nombre indica, este método realiza un **"append"**, es decir, **inserta el elemento como un hijo al final de todos los elementos hijos que existan**.

Es importante tener clara esta particularidad, porque aunque es lo más común, no siempre querremos insertar el elemento en esa posición:

```
const img = document.getElementByTagName("img");
img.src = "https://lenguajejs.com/assets/logo.svg";
img.alt = "Logo Javascript";
document.body.appendChild(img);
```

En este ejemplo podemos ver comoañadimos los atributos src y alt, obligatorios en una etiqueta de imagen. Por último, conectamos al DOM el elemento, utilizando el método .appendChild() sobre document.body que no es más que una referencia a la etiqueta <body> del documento HTML.

## ELIMINAR ELEMENTOS

Al igual que podemos insertar o reemplazar elementos, también podemos eliminarlos. Ten en cuenta que al «eliminar» un nodo o elemento HTML, lo que hacemos realmente no es borrarlo, sino **desconectarlo del DOM o documento HTML**, de modo que no están conectados, pero siguen existiendo. Estas operaciones que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

| Método | Descripción. |
|---|---|
| .remove() | Elimina el propio nodo de su elemento padre. |
| .removeChild(node) | Elimina y devuelve el nodo hijo node. |
| .replaceChild(new, old) | Reemplaza el nodo hijo old por new. Devuelve old. |

El método **.remove()** se encarga de desconectarse del DOM a sí mismo, mientras que el segundo método, **.removeChild()**, desconecta el nodo o elemento HTML proporcionado. Por último, con el método **.replaceChild()** se nos permite cambiar un nodo por otro.

## EL MÉTODO REMOVE()

Probablemente, la forma más sencilla de eliminar nodos o elementos HTML es utilizando el método **.remove()** sobre el nodo o etiqueta a eliminar:

```
const div = document.getElementById("deleteme");
div.isConnected;  // true
div.remove();
div.isConnected;  // false
```

# EVENTOS

En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio a fin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es el de la programación basada en eventos. Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JavaScript permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos. Los eventos de JavaScript permiten la interacción entre las aplicaciones JavaScript y los usuarios. Cada vez que se pulsa un botón, se produce un evento. Cada vez que se pulsa una tecla, también se produce un evento. No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que por ejemplo, cada vez que se carga una página, también se produce un evento.

## TIPOS DE EVENTOS

Cada **elemento HTML tiene definida su propia lista de posibles eventos** que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos HTML y un mismo elemento HTML puede tener asociados diferentes eventos.

El nombre de los eventos se construye mediante **el prefijo on**, seguido del **nombre** en inglés de la **acción asociada al evento**. Así, el evento de pinchar un elemento con el ratón se denomina **onclick** y el evento asociado a la acción de mover el ratón se denomina **onmousemove**.

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

| Evento | Descripción | Elementos para los que está definido |
|---|---|---|
| onblur | Un elemento pierde el foco | `<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>` |
| onchange | Un elemento ha sido modificado | `<input>`, `<select>`, `<textarea>` |
| onclick | Pulsar y soltar el ratón | Todos los elementos |
| ondblclick | Pulsar dos veces seguidas con el ratón | Todos los elementos |

| | | |
|---|---|---|
| onfocus | Un elemento obtiene el foco | <button>, <input>, <label>, <select>, <textarea>, <body> |
| onkeydown | Pulsar una tecla y no soltarla | Elementos de formulario y <body> |
| onkeypress | Pulsar una tecla | Elementos de formulario y <body> |
| onkeyup | Soltar una tecla pulsada | Elementos de formulario y <body> |
| onload | Página cargada completamente | <body> |
| onmousedown | Pulsar un botón del ratón y no soltarlo | Todos los elementos |
| onmousemove | Mover el ratón | Todos los elementos |
| onmouseout | El ratón "sale" del elemento | Todos los elementos |
| onmouseover | El ratón "entra" en el elemento | Todos los elementos |
| onmouseup | Soltar el botón del ratón | Todos los elementos |
| onreset | Inicializar el formulario | <form> |
| onresize | Modificar el tamaño de la ventana | <body> |
| onselect | Seleccionar un texto | <input>, <textarea> |
| onsubmit | Enviar el formulario | <form> |
| onunload | Se abandona la página, por ejemplo al cerrar el navegador | <body> |

Los eventos más utilizados en las aplicaciones web tradicionales son **onload** para esperar a que se cargue la página por completo, los eventos **onclick**, **onmouseover**, **onmouseout** para controlar el ratón y **onsubmit** para controlar el envío de los formularios.

# MANEJADORES DE EVENTOS

Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben **asociar funciones o código JavaScript a cada evento**. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede responder ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *manejador de eventos* (*event handlers* en inglés) y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:

- Manejadores como **atributos de los elementos XHTML**.
- Manejadores como **funciones JavaScript externas**.
- Manejadores **"semánticos"**.

## MANEJADORES COMO ATRIBUTOS HTML

Se trata del método más sencillo y a la vez *menos profesional* de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento. En este caso, el código se incluye en un atributo del propio elemento HTML. En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pinche con el ratón sobre un botón:

```
<input type="button" value="Pinchame y verás" onclick="console.log('Gracias
por pinchar');" />
```

El ejemplo anterior sólo quiere controlar el evento de pinchar con el ratón, cuyo nombre es onclick. Así, el elemento HTML para el que se quiere definir este evento, debe incluir un **atributo llamado onclick**.

El contenido del atributo es una cadena de texto que contiene todas las instrucciones JavaScript que se ejecutan cuando se produce el evento. En este caso, el código JavaScript es muy sencillo (console.log('Gracias por pinchar');), ya que solamente se trata de mostrar un mensaje.

## MANEJADORES DE EVENTOS COMO FUNCIONES EXTERNAS

La definición de manejadores de eventos en los atributos HTML es un método sencillo pero **poco aconsejable para tratar con los eventos en JavaScript**. El principal inconveniente es que se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para los casos más sencillos.

Cuando el código de la función manejadora es más complejo, como por ejemplo la validación de un formulario, **es aconsejable agrupar todo el código JavaScript en una función externa** que se invoca desde el código HTML cuando se produce el evento.

De esta forma, el siguiente ejemplo:

```
<input type="button" value="Pinchame y verás" onclick="console.log('Gracias por pinchar');" />
```

Se puede transformar en:

```
function muestraMensaje() {
  console.log('Gracias por pinchar');
}

<input type="button" value="Pinchame y verás" onclick="muestraMensaje()"/>
```

## MANEJADORES DE EVENTOS SEMÁNTICOS

Utilizar los atributos HTML o las funciones externas para añadir manejadores de eventos tiene un grave inconveniente: **"ensucian" el código** HTML de la página.

Como es conocido, al crear páginas web se recomienda separar los contenidos (HTML) de la presentación (CSS). En lo posible, también se recomienda separar los contenidos (HTML) de la programación (JavaScript). Mezclar JavaScript y HTML **complica excesivamente el código fuente de la página**, **dificulta su mantenimiento** y **reduce la semántica del documento final producido.**

Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica consiste en asignar las **funciones externas mediante las propiedades DOM de los elementos HTML**. Así, el siguiente ejemplo:

```
<input type="button" value="Pinchame y verás" onclick="console.log('Gracias por pinchar');" />
```

Se puede transformar en:

```
function muestraMensaje() {
  console.log('Gracias por pinchar');
}

document.getElementById("pinchable").onclick = muestraMensaje;

<input id="pinchable"  type="button" value="Pinchame y verás"/>
```

El código HTML resultante es **muy "limpio"**, ya que **no se mezcla con el código JavaScript**. La técnica de los manejadores semánticos consiste en:

1. Asignar un **identificador único** al elemento HTML mediante **el atributo id**.
2. Crear **una función** de JavaScript **encargada de manejar el evento**.
3. **Asignar la función a un evento** concreto del elemento XHTML mediante DOM.

Asignar la función manejadora mediante DOM es un proceso que requiere una explicación detallada. En primer lugar, se obtiene la referencia del elemento al que se va a asignar el manejador:

```
document.getElementById("pinchable");
```

A continuación, se asigna el evento deseado para el elemento:

```
document.getElementById("pinchable").onclick = ...
```

Por último, se asigna la función externa. Como ya se ha comentado en capítulos anteriores, lo más importante (y la causa más común de errores) es indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

```
document.getElementById("pinchable").onclick = muestraMensaje;
```

Si se añaden los paréntesis al final, en realidad se está invocando la función y asignando el valor devuelto por la función al evento onclick de elemento.

### ARCHIVO JAVASCRIPT

Para los ejemplo previamente vistos, habremos creado un archivo propio de JavaScript, que sería un archivo con extensión .js y llamaríamos este archivo mediante la etiqueta <script>. Recordemos que esto nos va a ayudar a separar el HTML de JavaScript y tener un HTML más claro.

El archivo js se vería así:

```
function muestraMensaje() {
  console.log('Gracias por pinchar');
}
document.getElementById("pinchable").onclick = muestraMensaje;
```

Y el html se vería así:

```
<!DOCTYPE html>
<html>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo</p>

<input id="pinchable" type="button" value="Pinchame y verás"/>

<script src="script1.js"></script>

</body>
</html>
```

## ASINCRONIA

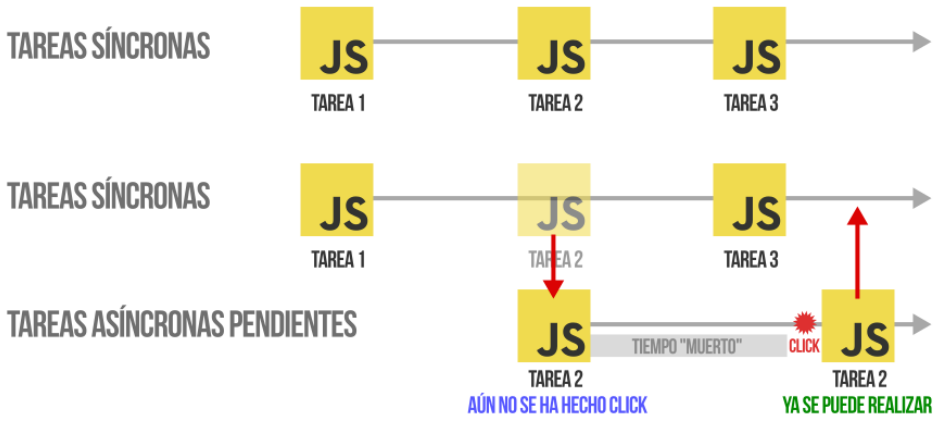
La asincronía es uno de los pilares fundamentales de Javascript, ya que es un lenguaje de programación de un sólo subproceso o hilo *(single thread)*, lo que significa que sólo puede ejecutar una cosa a la vez.

Si bien los idiomas de un sólo hilo simplifican la escritura de código porque no tiene que preocuparse por los problemas de concurrencia, esto también significa que no puede realizar operaciones largas como el acceso a la red sin bloquear el hilo principal.

Imagina que solicitas datos de una *API*. Dependiendo de la situación, el servidor puede tardar un tiempo en procesar la solicitud mientras bloquea el hilo principal y hace que la página web no responda.

Ahí es donde entra en juego la asincronía que permite realizar largas solicitudes de red sin bloquear el hilo principal.

Cuando hablamos de Javascript, habitualmente nos referimos a él como un lenguaje **no bloqueante**. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el resto de tareas.




Por lo que **Javascript** usa un **modelo asíncrono** y **no bloqueante**, con un *loop* de **eventos** implementado en un sólo hilo, *(single thread)* para operaciones de entrada y salida *(input/output)*.

¿Pero que significan todos estos conceptos? Ahora los vamos a explicar de manera más detallada.

## SINGLE THREAD Y MULTI THREAD

Un hilo la unidad básica de ejecución de un proceso, cada vez que abres un programa como el navegador o tu editor de código, se levanta un proceso en tu computadora e internamente este puede tener uno o varios hilos *(threads)* ejecutándose para que el proceso funcione.

## OPERACIONES DE CPU Y DE ENTRADA Y SALIDA

- **Operaciones CPU:** Aquellas que pasan el mayor tiempo consumiendo Procesos del *CPU*, por ejemplo, la escritura de ficheros.
- **Operaciones de Entrada y Salida:** Aquellas que pasan la mayor parte del tiempo esperando la respuesta de una petición o recurso, como la solicitud a una *API* o *DB*.

### CONCURRENCIA Y PARALELISMO

- Concurrencia: cuando dos o más tareas progresan simultáneamente.
- Paralelismo: cuando dos o más tareas se ejecutan, al mismo tiempo.

### BLOQUEANTE Y NO BLOQUEANTE

Se refiere a como la fase de espera de las operaciones afectan a nuestra aplicación:

- Bloqueante: Son operaciones que no devuelven el control a nuestra aplicación hasta que se ha completado. Por tanto el *thread* queda bloqueado en estado de espera.
- No Bloqueante: Son operaciones que devuelven inmediatamente el control a nuestra aplicación, independientemente del resultado de esta. En caso de que se haya completado, devolverá los datos solicitados. En caso contrario (si la operación no ha podido ser satisfecha) podría devolver un código de error.

### SÍNCRONO Y ASÍNCRONO

Se refiere a ¿cuándo tendrá lugar la respuesta?:

- Síncrono: La respuesta sucede en el presente, una operación síncrona esperará el resultado.
- Asíncrono: La respuesta sucede a futuro, una operación asíncrona no esperará el resultado.

## MECANISMOS ASÍNCRONOS EN JAVASCRIPT

Para controlar la asincronía, JavaScript cuenta con algunos mecanismos:

- Callback.
- Promises
- Async / Await.

## FUNCIONES CALLBACK

Los callbacks (*a veces denominados funciones de retrollamada o funciones callback*) no son más que un tipo de funciones que se pasan por parámetro a otras funciones. El objetivo de esto es tener una forma más legible de escribir funciones, más cómoda y flexible para reutilizarlas, y además entra bastante en consonancia con el concepto de asincronía de Javascript.

### CALLBACKS EN JAVASCRIPT

Vamos a ver un poco las funciones callbacks utilizadas para realizar tareas asíncronas. Probablemente, el caso más fácil de entender es utilizar un temporizador mediante la función setTimeout(callback, time).

Dicha función nos exige dos parámetros:

- La función **callback** a ejecutar
- El **tiempo time** que esperará antes de ejecutarla

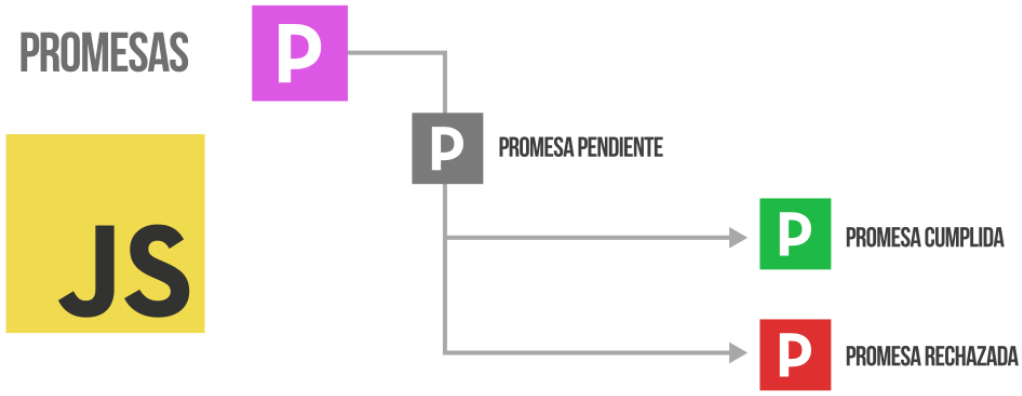
Así pues, el ejemplo sería el siguiente:

```
setTimeout(function() {
  console.log("He ejecutado la función");
}, 2000);
```

Simplemente, le decimos a **setTimeout()** que ejecute la **función callback** que le hemos pasado por primer parámetro cuando transcurran **2000 milisegundos** (*es decir, 2 segundos*).

## PROMESAS

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:




- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Con estas sencillas bases, podemos entender el funcionamiento de una promesa en JavaScript. Antes de empezar, también debemos tener claro que existen dos partes importantes de las promesas: **como consumirlas** *(utilizar promesas)* y **como crearlas** *(preparar una función para que use promesas y se puedan consumir).*

## PROMESAS EN JAVASCRIPT

Las **promesas** en Javascript se representan a través de un object, y cada **promesa** estará en un estado concreto: **pendiente, aceptada o rechazada.** Además, cada **promesa** tiene los siguientes métodos, que podremos utilizar para utilizarla:

| Método | Descripción. |
| --- | --- |
| .then(function resolve) | Ejecuta la función callback resolve cuando la promesa se cumple. |
| .catch(function reject) | Ejecuta la función callback reject cuando la promesa se rechaza. |
| .then(function resolve, function reject) | Método equivalente a las dos anteriores en el mismo .then(). |
| .finally(function end) | Ejecuta la función callback end tanto si se cumple como si se rechaza. |

### CONSUMIR UNA PROMESA

La forma general de consumir una promesa es utilizando el .then() con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla:

```
fetch("/robots.txt").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */
});
```

Lo que vemos en el ejemplo anterior es el uso de **la función fetch(), la cuál devuelve una promesa que se cumple cuando obtiene respuesta** de la petición realizada. De esta forma, estaríamos preparando (de una forma legible) la forma de actuar de nuestro código a la respuesta de la petición realizada, todo ello de forma asíncrona.

## API DE LAS PROMESAS

Ahora que sabemos ¿Qué son las promesas?, para qué y como se usan, podemos profundizar y aprender más sobre la **API Promise** nativa de Javascript, mediante la cuál podemos realizar operaciones con grupos de promesas, tanto independientes como dependientes entre sí.

### OBJETO PROMISE

El **objeto Promise** de Javascript tiene varios **métodos estáticos** que podemos utilizar en nuestro código. Todos devuelven una promesa y son los que veremos en la siguiente tabla:

| Método | Descripción. |
| --- | --- |
| Promise.all(Array list) | Acepta sólo si todas las promesas del Array se cumplen. |
| Promise.allSettled(Array list) | Acepta sólo si todas las promesas del Array se cumplen o rechazan. |
| Promise.any(Objet value) | Acepta con el valor de la primera promesa del Array que se cumpla. |
| Promise.race(Object value) | Acepta o rechaza dependiendo de la primera promesa que se procese. |
| Promise.resolve(Object value) | Devuelve un valor envuelto en una promesa que se cumple directamente. |
| Promise.reject(Object value) | Devuelve un valor envuelto en una promesa que se rechaza directamente. |

### PROMISE.RESOLVE() Y PROMISE.REJECT()

Mediante los métodos estáticos Promise.resolve() y Promise.reject() podemos devolver una promesa cumplida o rechazada.

```
//Promise
function resolverEn3seg() {
    return new Promise(function (resolve, reject) {
        // setTimeout(() => {
        //     resolve('2-Resuelto');
        // }, 3000);
        setTimeout(() => {
            reject(new Error("2-Oops i did it again"));
        }, 3000);
    });
}
```

## ASYNC / AWAIT

Las promesas fueron una gran mejora respecto a las callbacks para controlar la asincronía en JavaScript, sin embargo pueden llegar a ser muy verbosas a medida que se requieran más y más métodos .then().

Las funciones asíncronas (async / await) surgen para simplificar el manejo de las promesas.

La palabra **async** declara una función como asíncrona e indica que una promesa será automáticamente devuelta.

Podemos declarar como **async** funciones con nombre, anónimas o funciones flecha.

La palabra **await** debe ser usado siempre dentro de una función declarada como **async** y esperará de forma asíncrona y no bloqueante a que una promesa se resuelva o rechace.

## LA PALABRA CLAVE ASYNC

En primer lugar, tenemos la palabra clave **async**. Esta palabra clave se colocará previamente a function, para definirla así como una **función asíncrona**, el resto de la función no cambia:

```
async function funcion_asincrona() {
  return 42;
}
```

En el caso de que utilicemos **arrow function**, se definiría como vemos a continuación, colocando el async justo antes de los parámetros de la arrow function:

```
const funcion_asincrona = async () => 42;
```

Al ejecutar la función veremos que ya nos devuelve una **promise que ha sido cumplida**, con el valor devuelto en la función *(en este caso, 42)*. De hecho, podríamos utilizar un .then() para manejar la promesa:

```
funcion_asincrona().then(valor => {
  console.log("El resultado devuelto es: ", valor);
});
```

Sin embargo, veremos que lo que se suele hacer junto a async es utilizar la palabra clave await, que es donde reside lo interesante de utilizar este enfoque

## LA PALABRA CLAVE AWAIT

Cualquier función definida con async, o lo que es lo mismo, cualquier promise puede utilizarse junto a la palabra **clave await para manejarla**. Lo que hace await **es esperar a que se resuelva la promesa**, mientras permite **continuar ejecutando otras tareas** que puedan realizarse:

```
const funcion_asincrona = async () => 42;
const valor = funcion_asincrona();              // Promise { <fulfilled>:
                                                // 42 }
const asyncValue = await funcion_asincrona();   // 42
```

Observa que en el caso de valor, que se ejecuta sin await, lo que obtenemos es el valor devuelto por la función, **pero "envuelto" en una promesa que deberá utilizarse con .then() para manejarse**. Sin embargo, en asyncValue estamos **obteniendo un tipo de dato numérico**, guardando el valor directamente **ya procesado**, ya que await **espera a que se resuelva la promesa de forma asíncrona y guarda el valor**

# API

## ¿QUÉ ES UNA API?

El término API es una abreviatura de **Application Programming Interfaces**, que en español significa **interfaz de programación de aplicaciones**. Se trata de un **conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones**, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Así pues, podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores de terceros.

## PARA QUÉ SIRVE UNA API

Una de las principales funciones de las API es poder facilitarle el trabajo a los desarrolladores y ahorrarles tiempo y dinero. Por ejemplo, si estás creando una aplicación que es una tienda online, no necesitarás crear desde cero un sistema de pagos u otro para verificar si hay stock disponible de un producto. Podrás utilizar la API de un servicio de pago ya existente, por ejemplo PayPal, y pedirle a tu distribuidor una API que te permita saber el stock que ellos tienen.

- https://developers.mercadolibre.com.ar/
- https://developer.paypal.com/docs/api/overview/
- https://developers.facebook.com/docs/apis-and-sdks?locale=es_ES
- https://datosgobar.github.io/georef-ar-api/
- https://developers.google.com/maps/documentation/javascript/overview
- https://rickandmortyapi.com/
- https://dog.ceo/dog-api/
- https://developer.spotify.com/documentation/web-api/

Etc...

# FETCH

## ¿COMO COMUNICARME CON UNA API USANDO JAVASCRIPT?

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global fetch() (en-US) que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest. Fetch proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías como Service Workers (en-US). Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

La especificación fetch difiere de JQuery.ajax() en dos formas principales:

- El objeto Promise devuelto desde fetch() no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado ok configurado a false), y este solo sera rechazado ante un fallo de red o si algo impidió completar la solicitud.
- Por defecto, fetch no enviará ni recibirá cookies del servidor, resultando en peticiones no autenticadas si el sitio permite mantentener una sesión de usuario (para mandar cookies, credentials de la opción init deberan ser configuradas). Desde el 25 de agosto de 2017. La especificación cambió la politica por defecto de las credenciales a same-origin. Firefox cambió desde la versión 61.0b13.

Una petición básica de fetch es realmente simple de realizar. Eche un vistazo al siguiente código:

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

Aquí estamos recuperando un archivo JSON a través de red e imprimiendo en la consola. El uso de fetch() más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto Promise conteniendo la respuesta, un objeto Response.

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método json() (definido en el mixin de Body, el cual está implementado por los objetos Request y Response).

```
// Fetch GET
async function getAllCharacters() {
    let response = await fetch("https://rickandmortyapi.com/api/character");
    let data = await response.json();
    return data;
}
```

```
//Fetch POST
async function postData(url = '', data = {}) {
    const response = await fetch(url, {
        method: 'POST', // *GET, POST, PUT, DELETE, etc.
        mode: 'cors', // no-cors, *cors, same-origin
        cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-
cached
        credentials: 'same-origin', // include, *same-origin, omit
        headers: {
            'Content-Type': 'application/json'
            // 'Content-Type': 'application/x-www-form-urlencoded',
        },
        redirect: 'follow', // manual, *follow, error
        referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade,
origin, origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-
cross-origin, unsafe-url
        body: JSON.stringify(data) // el body debe ser igual al "Content-Type"
header
    });
    return response.json();
}

postData('https://localhost:8080', { mascota: "Chiquito" })
    .then(data => {
        console.log(data);
    });
```

# OBJETO STORAGE

El objeto Storage (API de almacenamiento web) nos permite almacenar datos de manera local en el navegador y sin necesidad de realizar alguna conexión a una base de datos.

# LOCALSTORAGE Y SESSIONSTORAGE: ¿QUÉ SON?

**localStorage y sessionStorage** son propiedades que acceden al objeto **Storage** y tienen la función de **almacenar datos de manera local**, la diferencia entre éstas dos es que localStorage almacena la información de **forma indefinida o hasta que se decida limpiar los datos del navegador** y sessionStorage almacena información **mientras la pestaña donde se esté utilizando siga abierta, una vez cerrada, la información se elimina**.

## GUARDAR DATOS EN STORAGE

Para guardar datos usamos el método **set(String "key", item).** En el primer parámetro "key" ingresamos el nombre de nuestro elemento, y en el parámetro "item" ingresamos el valor de éste.

```
localStorage.setItem('mascota', 'Chiquito');//guardo un elemento

//SessionStorage:

sessionStorage.setItem('mascota', 'Filomena');//guardo un elemento
```

### RECUPERAR DATOS DE STORAGE

Con éste método obtenemos desde el Local o SessionStorage el valor de nuestro elemento, donde "key" es el nombre de éste.

```
let miMascota1 = localStorage.getItem('mascota');//obtengo un elemento

//SessionStorage:

let miMascota2 = sessionStorage.getItem('mascota');//obtengo un elemento
```

### ELIMINAR DATOS DE STORAGE

Para eliminar un elemento dentro de Storage, usaremos el método remove("key"), que recibe el nombre del contenido a borrar.

```
localStorage.removeItem('mascota');//solo elimino este ítem

//SessionStorage:

sessionStorage.removeItem('mascota');//solo elimino este ítem
```

### NUMERO DE ELEMENTOS EN EL STORAGE

Si queremos averiguar la cantidad de elementos que tenemos guardados en el storage, vamos a usar el método length.

```
localStorage.length;//numero de elementos en local storage

//SessionStorage:

sessionStorage.length;//numero de elementos en local storage
```

### LIMPIAR TODO EL STORAGE

Ya para finalizar veremos la forma para eliminar todos los datos del Storage y dejarlo completamente limpio

```
localStorage.clear();//borro todos los items

//SessionStorage:

sessionStorage.clear();//borro todos los items
```

## ¿QUE SON LAS COOKIES?

Las cookies, de nombre más exacto HTTP cookies, es una tecnología que en su día inventó el navegador Netscape, y que consiste básicamente en **información enviada o recibida en las cabeceras HTTP** y que queda almacenada localmente **client-side durante un tiempo determinado**. En otras palabras, es información que queda almacenada en el dispositivo del usuario y que se envía hacia y desde el servidor web en las cabeceras HTTP.

Cuándo un usuario solicita una página web (o cualquier otro recurso), el servidor envía el documento, cierra la conexión y se olvida del usuario. Si el mismo usuario vuelve a solicitar la misma u otra página al servidor, será tratado como si fuera la primera solicitud que realiza. Esta situación puede suponer un problema en muchas situaciones y las cookies son una técnica que permite solucionarlo.

Con las cookies, el servidor puede enviar información al usuario en las cabeceras HTTP de respuesta y esta información queda almacenada en el dispositivo del usuario. En la siguiente solicitud que realice el usuario la cookie es enviada de vuelta al servidor en las cabeceras HTTP de solicitud. En el servidor podemos leer esta información y así "recordar" al usuario e información asociada a él.

## DOCUMENT.COOKIE

Mediante esta propiedad de pueden crear, modificar, eliminar y leer cookies en Javascript. Dentro de ella se incluyen diversos parámetros, como los que vemos a continuación.

## PARAMETROS

Una cookie consiste en una cadena de texto (string) con varios pares key=value cada uno separado por ";" :

```
<nombre>=<valor>; expires=<fecha>; max-age=<segundos>; path=<ruta>;
domain=<dominio>; secure; httponly;
```

Veamos cada uno de los parámetros de una cookie en más detalle.

### Nombre-Valor

Es un parámetro obligatorio a la hora de crear la cookies. "Nombre" se refiere al nombre que se adjudica a la cookies, mientras que "valor" representa su valor. Por ejemplo, "nombre" podría ser "color_favorito" y "valor" podría ser "azul".

### Expire date

Este parámetro es opcional y establece una fecha de final de validez de la cookie. La fecha se ha de establecer en formato UTC.

Otro parámetro temporal es «*max-age*«, que establece la duración en segundos de la cookie.

En caso de no asignar ningún valor para estos parámetros, se creará una cookie de sesión que espirará cuando el usuario finalice la sesión.

Si se establece una fecha de validez anterior a la fecha actual, o se asigna un valor negativo en «*max-age*«, lo que se conseguirá es eliminar la cookie.

### Domain & Path

Este parámetro **es opcional** y básicamente se trata de la URL para la cual la cookie es válida. En el caso de "Domain" se refiere al **dominio**, mientras que "Path" es el **subdominio.**

Debes tener en cuenta que la directiva same-origin policy no permite crear cookies para un dominio diferente al que crea la propia cookie. En el caso de los subdominios, se debe indicar para cuál se desea asignar la cookie. En caso de no asignar ninguna ruta, se creará automáticamente para la ruta de la página actual.

### secure

Parámetro **opcional**, sin valor. Si está presente la cookie sólo es válida para conexiones encriptadas (por ejemplo mediante protocolo HTTPS).

### HttpOnly

Parámetro **opcional,** no disponible en JavaScript ya que, crea cookies válidas sólo para protocolo HTTP/HTTPS y no para otras APIs, incluyendo JavaScript.

## ¿CÓMO CREAR UNA COOKIE CON JAVASCRIPT?

Para **guardar cookies en JavaScript** hay que definir el código y los parámetros de dicha cookie, y asignarlos a **document.cookie.**

```
document.cookie = "mascota=Malva";//guardo un elemento
```

Para crear más cookies es necesario seguir este mismo proceso para cada una de ellas. Ten en cuenta que si creas una cookie con el mismo nombre y para la misma ruta que una ya existente ésta sustituirá a la anterior.

## ¿CÓMO LEER UNA COOKIE CON JAVASCRIPT?

Uno de los puntos negativos de las cookies en JavaScript es que no hay una manera de leer o encontrar cookies de manera individual. Para **leer cookies en JavaScript** hay que crear un String que incluya todas las cookies válidas del documento, y manipularlo de manera que se encuentre el nombre y valor de la cookie que buscas.

El código para hacerlo es el siguiente

```
let cookies = document.cookie;//obtengo todas las cookies
```

Y los resultados se mostrarán de la siguiente manera:

```
"cookie1=valor1;cookie2=valor2;cookie3=valor3;cookie4=valor4;….
```

## ¿COMO ELIMINAR UN DATO ASOCIADO A UNA COOKIE?

Para esto, pisaremos la cookie ya creada y le dejaremos el dato como una cadena vacia.

```
document.cookie = "mascota= ";//elimino el dato asociado al elemento
```

## DIFERENCIA ENTE COOKIES Y SESIONES

Principalmente, una de las mayores diferencias es que la información cuando la almacenas con una session se guarda en el lado del servidor y la información cuando la guardas con una cookie se guarda en el lado del cliente.

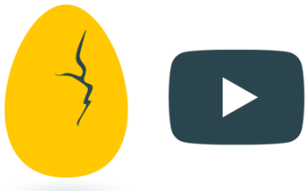
Además, las sesiones se destruyen cuando cierras el navegador (o cuando las destruyes manualmente) mientras que las cookies permanecen por un tiempo determinado en el navegador (que pueden ser varias semanas o incluso meses).

En relación con la seguridad, la cookie se guarda en el cliente, el usuario puede ver el archivo de la cookie y puede realizar operaciones similares de modificación y eliminación en el archivo de la cookie. La seguridad de los datos de la cookie es difícil de garantizar, mientras que los datos de la sesión se almacenan en el lado del servidor, tiene mejor seguridad. Si se usa junto con la base de datos, puede mantener los datos de la sesión durante mucho tiempo y obtener una buena seguridad.  Por lo tanto, se puede decir que las sesiones son más seguras que las cookies.

Esto hace, que no sea usar una o la otra, sino según que situaciones decidir que es lo mejor para utilizar en cada caso.

# EJERCICIOS DE APRENDIZAJE

¡¡Llegó el momento de poner nuestro conocimiento a prueba!! Vamos a trabajar todo lo que hemos visto en la guía de JavaScript y todo lo que veremos ahora con los videos de Youtube.



**VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.**

1. Escribir un algoritmo en el cual se consulte al usuario que ingrese ¿cómo está el día de hoy? (soleado, nublado, lloviendo). A continuación, mostrar por pantalla un mensaje que indique "El día de hoy está ...", completando el mensaje con el dato que ingresó el usuario.

2. Conocido el número en matemática PI ℗, pedir al usuario que ingrese el valor del radio de una circunferencia y calcular y mostrar por pantalla el área y perímetro. Recuerde que para calcular el área y el perímetro se utilizan las siguientes fórmulas:

   area = PI * radio$^2$

   perimetro = 2 * PI * radio

3. Escriba un programa en donde se pida la edad del usuario. Si el usuario es mayor de edad se debe mostrar un mensaje indicándolo.

4. Realiza un programa que sólo permita introducir los caracteres 'S' y 'N'. Si el usuario ingresa alguno de esos dos caracteres se deberá de imprimir un mensaje por pantalla que diga "CORRECTO", en caso contrario, se deberá imprimir "INCORRECTO".

5. Construir un programa que simule un menú de opciones para realizar las cuatro operaciones aritméticas básicas (suma, resta, multiplicación y división) con dos valores numéricos enteros. El usuario, además, debe especificar la operación con el primer carácter de la operación que desea realizar: 'S' o 's' para la suma, 'R' o 'r' para la resta, 'M' o 'm' para la multiplicación y 'D' o 'd' para la división.

6. Realizar un programa que, dado un número entero, visualice en pantalla si es par o impar. En caso de que el valor ingresado sea 0, se debe mostrar "el número no es par ni impar".

7. Escriba un programa en el cual se ingrese un valor límite positivo, y a continuación solicite números al usuario hasta que la suma de los números introducidos supere el límite inicial.

8. Escribir un programa que lea números enteros hasta teclear 0 (cero). Al finalizar el programa se debe mostrar el máximo número ingresado, el mínimo, y el promedio de todos ellos.

9. Realizar un programa que pida una frase y el programa deberá mostrar la frase con un espacio entre cada letra. La frase se mostrara así: H o l a. Nota: recordar el funcionamiento de la función Substring().

10. Escribir una función flecha que reciba una palabra y la devuelva al revés.

11. Escribir una función que reciba un String y devuelva la palabra más larga.

    String Ejemplo: “Guia de JavaScript”
    Resultado esperado : “JavaScript”

12. Escribir una función flecha de JavaScript que reciba un argumento y retorne el tipo de dato.

13. Crear un objeto persona, con las propiedades nombre, edad, sexo ('H' hombre, 'M' mujer, 'O' otro), peso y altura. A continuación, muestre las propiedades del objeto JavaScript.

14. Crear un objeto libro que contenga las siguientes propiedades: ISBN, Título, Autor, Número de páginas. Crear un método para cargar un libro pidiendo los datos al usuario y luego informar mediante otro método el número de ISBN, el título, el autor del libro y el numero de páginas.

15. Escribe un programa JavaScript para calcular el área y el perímetro de un objeto Círculo con la propiedad radio**.** **Nota:** Cree dos métodos para calcular el área y el perímetro. El radio del círculo lo proporcionará el usuario.

16. Realizar un programa que rellene dos vectores al mismo tiempo, con 5 valores aleatorios y los muestre por pantalla.

17. Realizar un programa que elimine los dos últimos elementos de un array. Mostrar el resultado

18. A partir del siguiente array: var valores = [true, 5, false, "hola", "adios", 2]:

    a) Determinar cual de los dos elementos de texto es mayor

    b) Utilizando exclusivamente los dos valores booleanos del array, determinar los operadores necesarios para obtener un resultado true y otro resultado false

    c) Determinar el resultado de las cinco operaciones matemáticas realizadas con los dos elementos numéricos

19. Realizar un programa en Java donde se creen dos arreglos: el primero será un arreglo A de 50 números reales, y el segundo B, un arreglo de 20 números, también reales. El programa deberá inicializar el arreglo A con números aleatorios y mostrarlo por pantalla. Luego, el arreglo A se debe ordenar de menor a mayor y copiar los primeros 10 números ordenados al arreglo B de 20 elementos, y rellenar los 10 últimos elementos con el valor 0.5. Mostrar los dos arreglos resultantes: el ordenado de 50 elementos y el combinado de 20.

20. Realizar un programa que obtenga la siguiente matriz [[3], [6], [9], [12], [15]] y devuelve y muestre el siguiente array [6, 9, 12, 15, 18].

21. Escribir un programa para obtener un array de las propiedades de un objeto Persona. Las propiedades son nombre, edad, sexo ('H' hombre, 'M' mujer, 'O' otro), peso y altura.

22. Escribir un programa de JavaScript que al clickear un botón muestre un mensaje a elección.

23. Resalte todas las palabras de más de 8 caracteres en el texto del párrafo (con un fondo amarillo, por ejemplo)

Hey, you're not permitted in there. It's restricted. You'll be deactivated for sure.. Don't call me a mindless philosopher, you overweight glob of grease! Now come out before somebody sees you. Secret mission? What plans? What are you talking about? I'm not getting in there! I'm going to regret this. There goes another one. Hold your fire. There are no life forms. It must have been short-circuited. That's funny, the damage doesn't look as bad from out here. Are you sure this things safe? Close up formation. You'd better let her loose. Almost there! I can't hold them! It's away! It's a hit! Negative. Negative! It didn't go in. It just impacted on the surface. Red Leader, we're right above you. Turn to point... oh-five, we'll cover for you. Stay there... I just lost my starboard engine. Get set to make your attack run. The Death Star plans are not in the main computer. Where are those transmissions you intercepted? What have you done with those plans? We intercepted no transmissions. Aaah....This is a consular ship. Were on a diplomatic mission. If this is a consular ship...were is the Ambassador? Commander, tear this ship apart until you've found those plans and bring me the Ambassador. I want her alive! There she is! Set for stun! She'll be all right. Inform Lord Vader we have a prisoner. What a piece of junk. She'll make point five beyond the speed of light. She may not look like much, but she's got it where it counts, kid. I've added some special modifications myself. We're a little rushed, so if you'll hurry aboard we'll get out of here. Hello, sir.

24. Escribir un programa de JavaScript que a través de un formulario calcule el radio de un circulo y lo muestre en el HTML.

25. Escriba una función de JavaScript para obtener los valores de Nombre y Apellido del siguiente formulario.

```
<!DOCTYPE html>
<html><head>
<meta charset=utf-8 />
<title>Obtener nombre y apellido de form </title>
</head><body>
<form id="form1" onsubmit="getFormValores()">
Nombre: <input type="text" name="nombre" value="David"><br>
Apellido: <input type="text" name="apellido" value="Beckham"><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```