¿Qué es una función?

Fuente: lenguajejs.com

Las **funciones** nos permiten agrupar líneas de código en tareas con un nombre, para que, posteriormente, podamos hacer referencia a ese nombre para realizar todo lo que se agrupe en dicha tarea. Para usar funciones hay que hacer 2 cosas:

- **Declarar la función**: Preparar la función, darle un nombre y decirle las tareas que realizará.
- **Ejecutar la función**: «Llamar» a la función para que realice las tareas de su contenido.

Declaración

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  console.log("Hola, soy una función");
}
```

El contenido de la función es una línea que mostrará por consola un saludo. Sin embargo, si escribimos estas 4-5 líneas de código en nuestro programa, no mostrará nada por pantalla. Esto ocurre así porque solo hemos declarado la función (*le hemos dicho que existe*), pero aún nos falta el segundo paso, **ejecutarla**, que es realmente cuando se realizan las tareas de su contenido.

Veamos, ahora sí, el ejemplo completo con declaración y ejecución:

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  console.log("Hola, soy una función");
}

// Ejecución de la función
saludar();
```

En este ejemplo hemos **declarado la función** y además, hemos **ejecutado** la función (*en la última línea*) llamándola por su nombre y seguida de ambos paréntesis, que nos indican que es una función. En este ejemplo, si se nos mostraría en la consola Javascript el mensaje de saludo.

Ejemplo

Veamos un primer ejemplo que muestre en la consola Javascript la **tabla de multiplicar del 1**:

```
function tablaDelUno() {
  for (let i = 0; i <= 10; i++)
        console.log("1 x", i, "=", 1 * i);
}

// Bucle que ejecuta 3 veces la función tablaDelUno()
for (let i = 0; i < 3; i++){
        tablaDelUno();
}
```

¿Qué son los parámetros?

Pero las funciones no sirven sólo para esto. Tienen mucha más flexibilidad de la que hemos visto hasta ahora. A las funciones se les pueden pasar **parámetros**, que no son más que variables que existirán sólo dentro de dicha función, con el valor pasado desde la ejecución.

Veamos el siguiente ejemplo, utilizando el parámetro **hasta**:

```
// Declaración
function tablaDelUno(hasta) {
        for (let i = 0; i <= hasta; i++) {
                console.log("1 x", i, "=", 1 * i);
        }
}

// Ejecución
tablaDelUno(10);
tablaDelUno(5);
```

Como podemos ver, en el interior de los paréntesis de la función se ha indicado una variable llamada **hasta**. Esa variable contiene el valor que se le da a la hora de ejecutar la función, que en este ejemplo, si nos fijamos bien, se ejecuta dos veces: una con valor **10** y otra con valor **5**.

Si analizamos el código de la declaración de la función, vemos que utilizamos la variable **hasta** en la condición del bucle, para que el bucle llegue hasta ese número (*de ahí su nombre*). Por lo tanto, en la primera ejecución se nos mostrará la tabla de multiplicar del 1 hasta llegar al **1 x 10** y en la segunda ejecución se nos mostrará la tabla de multiplicar del 1 hasta llegar al **1 x 5**.

> La idea de las funciones es enfocarnos en el código de la declaración, y una vez lo tengamos funcionando, nos podemos olvidar de él porque está **encapsulado** dentro de la función. Simplemente tendremos que recordar el nombre de la función y los parámetros que hay que pasarle. Esto hace que sea mucho más fácil trabajar con el código.

Parámetros múltiples

Hasta ahora sólo hemos creado una función con **1 parámetro**, pero una función de Javascript puede tener muchos más parámetros. Vamos a crear otro ejemplo, mucho más útil donde convertimos nuestra función en algo más práctico y útil:

```
// Declaración
function tablaMultiplicar(tabla, hasta) {
  for (let i = 0; i <= hasta; i++)
        console.log(tabla, "x", i, "=", tabla * i);
}

// Ejecución
tablaMultiplicar(1, 10); // Tabla del 1
tablaMultiplicar(5, 10); // Tabla del 5
```

En este ejemplo, hemos modificado nuestra función **tablaDelUno()** por esta nueva versión que hemos cambiado de nombre a **tablaMultiplicar()**. Esta función necesita que le pasemos dos parámetros: **tabla** (*la tabla de multiplicar en cuestión*) y **hasta** (*el número hasta donde llegará la tabla de multiplicar*).

> Podemos añadir **más parámetros** a la función según nuestras necesidades. Es importante recordar que el orden de los parámetros es importante y que los nombres de cada parámetro no se pueden repetir en una misma función.

Parámetros por defecto

Es posible que en algunos casos queramos que ciertos parámetros tengan un valor sin necesidad de escribirlos en la ejecución. Es lo que se llama un **valor por defecto**.

En nuestro ejemplo anterior, nos podría interesar que la tabla de multiplicar llegue siempre hasta el 10, ya que es el comportamiento por defecto. Si queremos que llegue hasta otro número, lo indicamos explícitamente, pero si lo omitimos, queremos que llegue hasta 10. Esto se haría de la siguiente forma:

```
function tablaMultiplicar(tabla, hasta = 10) {
	for (let i = 0; i <= hasta; i++) {
		console.log(tabla, "x", i, "=", tabla * i);
	}
}

// Ejecución
tablaMultiplicar(2); // Esta tabla llegará hasta el número 10
tablaMultiplicar(2, 15); // Esta tabla llegará hasta el número 15
```

> Hay que remarcar que esta característica se añade en **ECMAScript 6**, por lo que en navegadores sin soporte podría no funcionar correctamente.

Devolución de valores

Hasta ahora hemos utilizado funciones simples que realizan acciones o tareas (*en nuestro caso, mostrar por consola*), pero habitualmente, lo que buscamos es que esa función realice una tarea y nos devuelva la información al exterior de la función, para así utilizarla o guardarla en una variable, que utilizaremos posteriormente para nuestros objetivos.

Para ello, se utiliza la palabra clave **return**, que suele colocarse al final de la función, ya que con dicha devolución terminamos la ejecución de la función (*si existe código después, nunca será ejecutado*).

Veamos un ejemplo con una operación muy sencilla, para verlo claramente:

```
// Declaración
function sumar(a, b) {
  return a + b; // Devolvemos la suma de a y b al exterior de la función
  console.log("Ya he realizado la suma."); // Este código nunca se ejecutará
}

// Ejecución
let resultado = sumar(5, 5); // Se guarda 10 en la variable resultado
```

Como podemos ver, esto nos permite crear funciones más modulares y reutilizables que podremos utilizar en multitud de casos, ya que la información se puede enviar al exterior de la función y utilizarla junto a otras funciones o para otros objetivos.

Una vez conocemos las bases de las funciones que hemos explicado en el tema de introducción funciones básicas, podemos continuar avanzando dentro del apartado de las funciones. En Javascript, las **funciones** son uno de los tipos de datos más importantes, ya que estamos continuamente utilizándolas a lo largo de nuestro código.

Y no, no me he equivocado ni he escrito mal el texto anterior; a continuación veremos que las funciones también pueden ser tipos de datos:

```
typeof function () {}; // 'function'
```

Creación de funciones

Hay varias formas de crear funciones en Javascript: por **declaración** (*la más usada por principiantes*), por **expresión** (*la más habitual en programadores con experiencia*) o mediante constructor de **objeto** (*no recomendada*):

Constructor	Descripción
`function nombre(p1, p2...) { }`	Crea una función mediante **declaración**.
`let nombre = function(p1, p2...) { }`	Crea una función mediante **expresión**.
`new Function(p1, p2..., code);`	Crea una función mediante un constructor de **objeto**.

Funciones por declaración

Probablemente, la forma más popular de estas tres, y a la que estaremos acostumbrados si venimos de otros lenguajes de programación, es la primera, a la **creación de funciones por declaración**. Esta forma permite declarar una función que existirá a lo largo de todo el código:

```
function saludar() {
  return "Hola";
}

saludar(); // 'Hola'
typeof saludar; // 'function'
```

De hecho, podríamos ejecutar la función **saludar()** incluso antes de haberla creado y funcionaría correctamente, ya que Javascript primero busca las declaraciones de funciones y luego procesa el resto del código.

Funciones por expresión

Sin embargo, en Javascript es muy habitual encontrarse códigos donde los programadores «guardan funciones» dentro de variables, para posteriormente «ejecutar dichas variables». Se trata de un enfoque diferente, creación de funciones por **expresión**, que fundamentalmente, hacen lo mismo con algunas diferencias:

```
// El segundo "saludar" (nombre de la función) se suele omitir: es redundante
const saludo = function saludar() {
  return "Hola";
};

saludo(); // 'Hola'
```

Con este nuevo enfoque, estamos creando una función **en el interior de una variable**, lo que nos permitirá posteriormente ejecutar la variable (*como si fuera una función*). Observa que el nombre de la función (*en este ejemplo: saludar*) pasa a ser inútil, ya que si intentamos ejecutar **saludar()** nos dirá que no existe y si intentamos ejecutar **saludo()** funciona correctamente.

¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las **funciones anónimas** (*o funciones lambda*).

Funciones como objetos

Como curiosidad, debes saber que se pueden declarar funciones como si fueran **objetos**. Sin embargo, es un enfoque que no se suele utilizar en producción. Simplemente es interesante saberlo para darse cuenta que en Javascript todo pueden ser objetos:

```
const saludar = new Function("return 'Hola';");

saludar(); // 'Hola'
```

Funciones anónimas

Las **funciones anónimas** o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
  return "Hola";
};

saludo; // ƒ () { return 'Hola'; }
saludo(); // 'Hola'
```

Observa que en la última línea del ejemplo anterior, estamos **ejecutando** la variable, es decir, ejecutando la función que contiene la variable. Sin embargo, en la línea anterior hacemos referencia a la variable (*sin ejecutarla, no hay paréntesis*) y nos devuelve la función en sí.

La diferencia fundamental entre las funciones por declaración y las funciones por expresión es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si «ejecutamos la variable» antes de declararla, nos dará un error.

Callbacks

Ahora que conocemos las **funciones anónimas**, podremos comprender más fácilmente como utilizar **callbacks** (*también llamadas funciones callback o retrollamadas*). A grandes rasgos, un **callback** (*llamada hacia atrás*) es pasar una **función B por parámetro** a una **función A**, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función:

```
// fB = Función B
const fB = function () {
  console.log("Función B ejecutada.");
};

// fA = Función A
const fA = function (callback) {
  callback();
};

fA(fB);
```

Funciones autoejecutables

Pueden existir casos en los que necesites crear una función y ejecutarla sobre la marcha. En Javascript es muy sencillo crear **funciones autoejecutables**. Básicamente, sólo tenemos que envolver entre paréntesis la función anónima en cuestión (*no necesitamos que tenga nombre, puesto que no la vamos a guardar*) y luego, ejecutarla:

```
// Función autoejecutable
(function () {
  console.log("Hola!!");
})();

// Función autoejecutable con parámetros
(function (name) {
  console.log(`¡Hola, ${name}!`);
})("Manz");
```

De hecho, también podemos utilizar parámetros en dichas funciones autoejecutables. Observa que sólo hay que pasar dichos parámetros al final de la función autoejecutable.

Ten en cuenta, que si la función autoejecutable devuelve algún valor con **return**, a diferencia de las **funciones por expresión**, en este caso lo que se almacena en la variable es el valor que devuelve la función autoejecutada:

```
const f = (function (name) {
  return `¡Hola, ${name}!`;
})("Manz");

f; // '¡Hola, Manz!`
typeof f; // 'string'
```

Clausuras

Las **clausuras** o cierres, es un concepto relacionado con las funciones y los ámbitos que suele costar comprender cuando se empieza en Javascript. Es importante tener las bases de funciones claras hasta este punto, lo que permitirá entender las bases de una clausura.

A grandes rasgos, en Javascript, una clausura o cierre se define como una función que «encierra» variables en su propio ámbito (*y que continúan existiendo aún habiendo terminado la función*). Por ejemplo, veamos el siguiente ejemplo:

```
// Clausura: Función incr()
const incr = (function () {
  let num = 0;
  return function () {
    num++;
    return num;
  };
})();

typeof incr; // 'function'
incr(); // 1
incr(); // 2
incr(); // 3
```

Tenemos una **función anónima** que es también una función autoejecutable. Aunque parece una función por expresión, no lo es, ya que la variable **incr** está guardando lo que devuelve la función anónima autoejecutable, que a su vez, es otra función diferente.

La «magia» de las clausuras es que en el interior de la función autoejecutable estamos creando una variable **num** que se guardará en el ámbito de dicha función, por lo tanto existirá con el valor declarado: **0**.

Por lo tanto, en la variable **incr** tenemos una función por expresión que además conoce el valor de una variable **num**, que sólo existe dentro de **incr**. Si nos fijamos en la función que devolvemos, lo que hace es incrementar el valor de **num** y devolverlo. Como la variable **incr** es una clausura y mantiene la variable en su propio ámbito, veremos que a medida que ejecutamos **incr()**, los valores de **num** (*que estamos devolviendo*) conservan su valor y se van incrementando.

Arrow functions

Las **Arrow functions**, funciones flecha o «fat arrow» son una forma corta de escribir funciones que aparece en Javascript a partir de **ECMAScript 6**. Básicamente, se trata de reemplazar eliminar la palabra **function** y añadir **=>** antes de abrir las llaves:

```
const func = function () {
  return "Función tradicional.";
};

const func = () => {
  return "Función flecha.";
};
```

Sin embargo, las **funciones flechas** tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves (*{}*).
- Además, en ese caso, automáticamente se hace un **return** de esa única línea, por lo que podemos omitir también el **return**.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: **() =>**.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: **e =>**.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: **(a, b) =>**.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: **({name: 'Manz'})**.

Por lo tanto, el ejemplo anterior se puede simplificar aún más:

```
const func = () => "Función flecha."; // 0 parámetros: Devuelve "Función flecha"
const func = (e) => e + 1; // 1 parámetro: Devuelve el valor de e + 1
const func = (a, b) => a + b; // 2 parámetros: Devuelve el valor de a + b
```

Las **funciones flecha** hacen que el código sea mucho más legible y claro de escribir, mejorando la productividad y la claridad a la hora de escribir código.

¿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*).

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 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 donde tendremos que elegir el elemento en cuestión posteriormente:

Métodos de búsqueda	Descripción
.getElementById(id)	Busca el elemento HTML con el id **id**. Si no, devuelve .
.getElementsByClassName(class)	Busca elementos con la clase **class**. Si no, devuelve **[]**.
.getElementsByName(name)	Busca elementos con atributo name **name**. Si no, devuelve **[]**.
.getElementsByTagName(tag)	Busca elementos **tag**. Si no encuentra ninguno, devuelve **[]**.

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 metodo 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: <div class="item"></div>
console.log(items.length);  // 3
```

Exactamente igual funcionan los métodos **getElementsByName(name)** y **getElementsByTagName(tag)**, salvo que se encargan de buscar elementos HTML por su atributo **name** o por su propia **etiqueta** de elemento HTML, respectivamente:

```
// Obtiene los elementos con atributo name="nickname"
const nicknames = document.getElementsByName("nickname");

// Obtiene todos los elementos <div> de la página
const divs = document.getElementsByTagName("div");
```

> Recuerda que el primer método tiene **getElement** en singular y el resto **getElements** en plural. Ten en cuenta ese detalle para no olvidarte que uno devuelve un sólo elemento y el resto una lista de ellos.

Métodos modernos

Aunque podemos utilizar los métodos tradicionales que acabamos de ver, actualmente tenemos a nuestra disposición dos nuevos métodos de búsqueda de elementos que son mucho más cómodos y prácticos si conocemos y dominamos los selectores CSS. Es el caso de los métodos **.querySelector()** y **.querySelectorAll()**:

Método de búsqueda	Descripción
.querySelector(sel)	Busca el primer elemento que coincide con el selector CSS **sel**. Si no, .
.querySelectorAll(sel)	Busca todos los elementos que coinciden con el selector CSS **sel**. Si no, **[]**.

Con estos dos métodos podemos realizar todo lo que hacíamos con los **métodos tradicionales** mencionados anteriormente e incluso muchas más cosas (*en menos código*), ya que son muy flexibles y potentes gracias a los **selectores CSS**.

querySelector()

El primero, **.querySelector(selector)** devuelve el primer elemento que encuentra que encaja con el selector CSS suministrado en **selector**. Al igual que su «equivalente» **.getElementById()**, devuelve un solo elemento y en caso de no coincidir con ninguno, devuelve :

```
const page = document.querySelector("#page");              // <div id="page"></div>
const info = document.querySelector(".main .info");        // <div class="info"></div>
```

Lo interesante de este método, es que al permitir suministrarle un selector CSS básico o incluso un selector CSS avanzado, se vuelve un sistema mucho más potente.

El primer ejemplo es equivalente a utilizar un **.getElementById()**, sólo que en la versión de **.querySelector()** indicamos por parámetro un , y en el primero le pasamos un simple . Observa que estamos indicando un **#** porque se trata de un **id**.

En el segundo ejemplo, estamos recuperando el primer elemento con clase **info** que se encuentre dentro de otro elemento con clase **main**. Eso podría realizarse con los métodos tradicionales, pero sería menos directo ya que tendríamos que realizar varias llamadas, con **.querySelector()** se hace directamente con sólo una.

querySelectorAll()

Por otro lado, el método **.querySelectorAll()** realiza una búsqueda de elementos como lo hace el anterior, sólo que como podremos intuir por ese **All()**, devuelve un con todos los elementos que coinciden con el CSS:

```
// Obtiene todos los elementos con clase "info"
const infos = document.querySelectorAll(".info");

// Obtiene todos los elementos con atributo name="nickname"
const nicknames = document.querySelectorAll('[name="nickname"]');

// Obtiene todos los elementos <div> de la página HTML
const divs = document.querySelectorAll("div");
```

En este caso, recuerda que querySelectorAll**()** siempre nos devolverá un grupo de elementos. Depende de los elementos que encuentre mediante nos devolverá un valor de **0** elementos o de **1**, **2** o más elementos.

> Al realizar una búsqueda de elementos y guardarlos en una variable, podemos realizar la búsqueda posteriormente sobre esa variable en lugar de hacerla sobre **document**. Esto permite realizar búsquedas acotadas por zonas, en lugar de realizarlo siempre sobre **document**, que buscará en todo el documento HTML.

Sobre todo si te encuentras en fase de aprendizaje, lo normal suele ser crear código HTML desde un fichero HTML. Sin embargo, y sobre todo con el auge de las páginas **SPA** (*Single Page Application**) y los frameworks Javascript, esto ha cambiado bastante y es bastante frecuente **crear código HTML desde Javascript** de forma dinámica.

Esto tiene sus ventajas y sus desventajas. Un fichero **.html** siempre será más sencillo, más «estático» y más directo, ya que lo primero que analiza un navegador web es un fichero de marcado HTML. Por otro lado, un fichero **.js** es más complejo y menos directo, pero mucho más potente, «dinámico» y flexible, con menos limitaciones.

En este artículo vamos a ver como podemos **crear elementos HTML desde Javascript** y aprovecharnos de la potencia de Javascript para hacer cosas que desde HTML, sin ayuda de Javascript, no podríamos realizar o costaría mucho más.

Crear elementos HTML

Existen una serie de métodos para **crear de forma eficiente** diferentes elementos HTML o nodos, y que nos pueden convertir en una tarea muy sencilla el crear estructuras dinámicas, mediante bucles o estructuras definidas:

Métodos	Descripción
.createElement(tag, options)	Crea y devuelve el elemento HTML definido por el **tag**.
.createComment(text)	Crea y devuelve un nodo de comentarios HTML **<!-- text -->**.
.createTextNode(text)	Crea y devuelve un nodo HTML con el texto **text**.

.cloneNode(deep)	Clona el nodo HTML y devuelve una copia. **deep** es **false** por defecto.
.isConnected	Indica si el nodo HTML está insertado en el documento HTML.

Para empezar, nos centraremos principalmente en la primera, que es la que utilizamos para **crear elementos HTML** en el DOM.

El método createElement()

Mediante el método **.createElement()** podemos crear un HTML **en memoria** (*¡no estará insertado aún en nuestro documento HTML!*). Con dicho elemento almacenado en una variable, podremos modificar sus características o contenido, para **posteriormente** insertarlo en una posición determinada del DOM o documento HTML.

Vamos a centrarnos en el proceso de **creación del elemento**, y en el próximo capítulo veremos el apartado de insertarlo en el DOM. El funcionamiento de **.createElement()** es muy sencillo: se trata de pasarle el nombre de la etiqueta **tag** a utilizar.

```
const div = document.createElement("div");      // Creamos un <div></div>
const span = document.createElement("span");    // Creamos un <span></span>
const img = document.createElement("img");      // Creamos un <img>
```

De la misma forma, podemos crear comentarios HTML con **createComment()** o nodos de texto sin etiqueta HTML con **createTextNode()**, pasándole a ambos un con el texto en cuestión. En ambos, se devuelve un que podremos utilizar luego para insertar en el documento HTML:

```
const comment = document.createComment("Comentario"); // <!--Comentario-->
const text = document.createTextNode("Hola");         // Nodo de texto: 'hola'
```

El método **createElement()** tiene un parámetro opcional denominado **options**. Si se indica, será un objeto con una propiedad **is** para definir un **elemento personalizado** en una modalidad menos utilizada. Se verá más adelante en el apartado de **Web Components**.

> Ten presente que en los ejemplos que hemos visto estamos creando los elementos en una constante, pero de momento **no están añadiéndose al documento HTML**, por lo que no aparecerían visualmente. Más adelante veremos como añadirlos.

El método cloneNode()

Hay que tener mucho cuidado al crear y **duplicar** elementos HTML. Un error muy común es asignar un elemento que tenemos en otra variable, pensando que estamos creando una copia (*cuando no es así*):

```
const div = document.createElement("div");
div.textContent = "Elemento 1";

const div2 = div;   // NO se está haciendo una copia
div2.textContent = "Elemento 2";

div.textContent;  // 'Elemento 2'
```

Con esto, quizás pueda parecer que estamos duplicando un elemento para crearlo a imagen y semejanza del original. Sin embargo, lo que se hace es una **referencia** al elemento original, de modo que si se modifica el **div2**, también se modifica el elemento original. Para evitar esto, lo ideal es utilizar el método **cloneNode()**:

```
const div = document.createElement("div");
div.textContent = "Elemento 1";

const div2 = div.cloneNode();   // Ahora SÍ estamos clonando
div2.textContent = "Elemento 2";

div.textContent;  // 'Elemento 1'
```

El método **cloneNode(deep)** acepta un parámetro **deep** opcional, a **false** por defecto, para indicar el tipo de clonación que se realizará:

- Si es **true**, clonará también sus hijos, conocido como una **clonación profunda** (*Deep Clone*).
- Si es **false**, no clonará sus hijos, conocido como una **clonación superficial** (*Shallow Clone*).

La propiedad isConnected

Por último, la propiedad **isConnected** nos indica si el nodo en cuestión está conectado al DOM, es decir, si está insertado en el documento HTML:

- Si es **true**, significa que el elemento está conectado al DOM.
- Si es **false**, significa que el elemento no está conectado al DOM.

Hasta ahora, hemos creado elementos que no lo están (*permanecen sólo en memoria*). En el capítulo Insertar elementos en el DOM veremos como insertarlos en el documento HTML para que aparezca visualmente en la página.

Atributos HTML de un elemento

Hasta ahora, hemos visto cómo crear elementos HTML con Javascript, pero no hemos visto cómo modificar los atributos HTML de dichas etiquetas creadas. En general, una vez tenemos un elemento sobre el que vamos a crear algunos atributos, lo más sencillo es **asignarle valores como propiedades** de objetos:

```
const div = document.createElement("div"); // <div></div>
div.id = "page";           // <div id="page"></div>
div.className = "data";    // <div id="page" class="data"></div>
div.style = "color: red"; // <div id="page" class="data" style="color: red"></div>
```

Sin embargo, en algunos casos esto se puede complicar (*como se ve en uno de los casos del ejemplo anterior*). Por ejemplo, la palabra **class** (*para crear clases*) o la palabra **for** (*para bucles*) son palabras reservadas de Javascript y no se podrían utilizar para crear

atributos. Por ejemplo, si queremos establecer una clase, se debe utilizar la propiedad **className**.

Aunque la forma anterior es la más rápida, tenemos algunos métodos para utilizar en un elemento HTML y añadir, modificar o eliminar sus atributos:unos ejemplos de uso donde podemos ver como funcionan:

```
// Obtenemos <div id="page" class="info data dark" data-number="5"></div>
const div = document.querySelector("#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");        // <div class="info data dark" data-number="5"></div>
div.setAttribute("id", "page");   // (Vuelve a añadirlo)
```

> Recuerda que hasta ahora hemos visto cómo crear elementos y cambiar sus atributos, pero **no los hemos insertado en el DOM** o documento HTML, por lo que no los veremos visualmente en la página.

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.

Las propiedades son las siguientes:

Propiedades	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.

La propiedad **nodeName** nos devuelve el nombre del todo, que en elementos HTML es interesante puesto que nos devuelve el nombre de la etiqueta **en mayúsculas**. Se trata de una propiedad de sólo lectura, por lo cual no podemos modificarla, sólo acceder a ella.

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.querySelector("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. En el caso de que el elemento tenga anidadas varias etiquetas HTML una dentro de otra, la propiedad **.textContent** se quedará sólo con el contenido textual completo, como se puede ver en el siguiente ejemplo:

```
// Obtenemos <div class="info">Hola <strong>amigos</strong></div>
const div = document.querySelector(".info");

div.textContent;  // "Hola amigos"
```

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.querySelector(".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.

> Ten en cuenta que la propiedad **.innerHTML** comprueba y parsea el marcado HTML escrito (*corrigiendo si hay errores*) antes de realizar la asignación. Por ejemplo, si en el ejemplo anterior nos olvidamos de escribir el cierre **** de la etiqueta, **.innerHTML** automáticamente lo cerrará. Esto puede provocar algunas incongruencias si el código es

> incorrecto o una disminución de rendimiento en textos muy grandes que hay que preprocesar.

Por otro lado, la propiedad **.outerHTML** es muy similar a **.innerHTML**. Mientras que esta última devuelve el código HTML del interior de un elemento HTML, **.outerHTML** devuelve también el código HTML del propio elemento en cuestión. Esto puede ser muy útil para reemplazar un elemento HTML combinándolo con **.innerHTML**:

```
const data = document.querySelector(".data");
data.innerHTML = "<h1>Tema 1</h1>";

data.textContent;   // "Tema 1"
data.innerHTML;     // "<h1>Tema 1</h1>"
data.outerHTML;     // "<div class="data"><h1>Tema 1</h1></div>"
```

En este ejemplo se pueden observar las diferencias entre las propiedades **.textContent** (*contenido de texto*), **.innerHTML** (*contenido HTML*) y **.outerHTML** (*contenido y contenedor HTML*).

Insertar elementos

A pesar de que los métodos anteriores son suficientes para crear elementos y estructuras HTML complejas, sólo son aconsejables para pequeños fragmentos de código o texto, ya que en estructuras muy complejas (*con muchos elementos HTML*) la **legibilidad del código** sería menor y además, el rendimiento podría resentirse.

Hemos aprendido a crear elementos HTML y sus atributos, pero aún no hemos visto como añadirlos al documento HTML actual (*conectarlos al DOM*), operación que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

Métodos	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, .
.insertAdjacentHTML(pos, str)	Inserta el código HTML **str** en la posición **pos**.
.insertAdjacentText(pos, text)	Inserta el texto **text** en la posición **pos**.

De ellos, probablemente el más extendido es **.appendChild()**, no obstante, la familia de métodos **.insertAdjacent*()** también tiene buen soporte en navegadores y puede usarse de forma segura en la actualidad.

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.createElement("img");
img.src = "https://lenguajejs.com/assets/logo.svg";
img.alt = "Logo Javascript";

document.body.appendChild(img);
```

En este ejemplo podemos ver como creamos un elemento **<img>** que aún no está conectado al DOM. Posteriormente, añadimos 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.

Veamos otro ejemplo:

```
const div = document.createElement("div");
div.textContent = "Esto es un div insertado con JS.";

const app = document.createElement("div"); // <div></div>
app.id = "app"; // <div id="app"></div>
app.appendChild(div); // <div id="app"><div>Esto es un div insertado con JS</div></div>
```

En este ejemplo, estamos creando dos elementos, e insertando uno dentro de otro. Sin embargo, a diferencia del anterior, el elemento **app** no está conectado aún al DOM, sino que lo tenemos aislado en esa variable, sin insertar en el documento. Esto ocurre porque **app** lo acabamos de crear, y en el ejemplo anterior usabamos **document.body** que es una referencia a un elemento que ya existe en el documento.

Los métodos insertAdjacent*()

Los métodos de la familia **insertAdjacent** son bastante más versátiles que **.appendChild()**, ya que permiten muchas más posibilidades. Tenemos tres versiones diferentes:

- **.insertAdjacentElement()** donde insertamos un objeto

- **.insertAdjacentHTML()** donde insertamos **código HTML** directamente (*similar a innerHTML*)
- **.insertAdjacentText()** donde no insertamos elementos HTML, sino un con texto

En las tres versiones, debemos indicar por parámetro un **pos** como primer parámetro para indicar en que posición vamos a insertar el contenido. Hay 4 opciones posibles:

- **beforebegin**: El elemento se inserta **antes** de la etiqueta HTML de apertura.
- **afterbegin**: El elemento se inserta **dentro** de la etiqueta HTML, **antes de su primer hijo**.
- **beforeend**: El elemento se inserta **dentro** de la etiqueta HTML, **después de su último hijo**. Es el equivalente a usar el método **.appendChild()**.
- **afterend**: El elemento se inserta **después** de la etiqueta HTML de cierre.

Veamos algunos ejemplo aplicando cada uno de ellos con el método **.insertAdjacentElement()**:

```
const div = document.createElement("div");  // <div></div>
div.textContent = "Ejemplo";                // <div>Ejemplo</div>

const app = document.querySelector("#app"); // <div id="app">App</div>

app.insertAdjacentElement("beforebegin", div);
// Opción 1: <div>Ejemplo</div> <div id="app">App</div>

app.insertAdjacentElement("afterbegin", div);
// Opción 2: <div id="app"> <div>Ejemplo</div> App</div>

app.insertAdjacentElement("beforeend", div);
// Opción 3: <div id="app">App <div>Ejemplo</div> </div>

app.insertAdjacentElement("afterend", div);
// Opción 4: <div id="app">App</div> <div>Ejemplo</div>
```

Ten en cuenta que en el ejemplo muestro **varias opciones alternativas**, no lo que ocurriría tras ejecutar las cuatro opciones una detrás de otra.

Por otro lado, notar que tenemos **tres versiones** en esta familia de métodos, una que actua sobre elementos HTML (*la que hemos visto*), pero otras dos que actuan sobre código HTML y sobre nodos de texto. Veamos un ejemplo de cada una:

```
app.insertAdjacentElement("beforebegin", div);
// Opción 1: <div>Ejemplo</div> <div id="app">App</div>

app.insertAdjacentHTML("beforebegin", '<p>Hola</p>');
// Opción 2: <p>Hola</p> <div id="app">App</div>

app.insertAdjacentText("beforebegin", "Hola a todos");
// Opción 3: Hola a todos <div id="app">App</div>
```

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.

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.querySelector(".deleteme");

div.isConnected;    // true
div.remove();
div.isConnected;    // false
```

En este caso, lo que hemos hecho es buscar el elemento HTML ** ** en el documento HTML y desconectarlo de su elemento padre, de forma que dicho elemento pasa a no pertenecer al documento HTML.

Sin embargo, existen algunos métodos más para eliminar o reemplazar elementos:

Métodos	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 removeChild()

En algunos casos, nos puede interesar eliminar un nodo hijo de un elemento. Para esas situaciones, podemos utilizar el método **.removeChild(node)** donde **node** es el nodo hijo que queremos eliminar:

```
const div = document.querySelector(".item:nth-child(2)");   // <div class="item">2</div>

document.body.removeChild(div); // Desconecta el segundo .item
```

El método replaceChild()

De la misma forma, el método **replaceChild(new, old)** nos permite cambiar un nodo hijo **old** por un nuevo nodo hijo **new**. En ambos casos, el método nos devuelve el nodo reemplazado:

```
const div = document.querySelector(".item:nth-child(2)");

const newnode = document.createElement("div");
newnode.textContent = "DOS";

document.body.replaceChild(newnode, div);
```

La propiedad className

Javascript tiene a nuestra disposición una propiedad **.className** en todos los elementos HTML. Dicha propiedad contiene el valor del atributo HTML **class**, y puede tanto leerse como reemplazarse:

Propiedad	Descripción
.className	Acceso directo al valor del atributo HTML **class**. También se puede asignar.
.classList	Objeto especial para manejar clases CSS. Contiene métodos y propiedades de ayuda.

La propiedad **.className** viene a ser la modalidad directa y rápida de utilizar el getter **.getAttribute("class")** y el setter **.setAttribute("class", v)**. Veamos un ejemplo utilizando estas propiedades y métodos y su equivalencia:

```
const div = document.querySelector(".element");

// Obtener clases CSS
div.className;             // "element shine dark-theme"
div.getAttribute("class");  // "element shine dark-theme"
```

```
// Modificar clases CSS
div.className = "elemento brillo tema-oscuro";
div.setAttribute("class", "elemento brillo tema-oscuro");
```

Trabajar con **.className** tiene una limitación cuando trabajamos con **múltiples clases CSS**, y es que puedes querer realizar una manipulación sólo en una clase CSS concreta, dejando las demás intactas. En ese caso, modificar clases CSS mediante una asignación **.className** se vuelve poco práctico. Probablemente, la forma más interesante de manipular clases desde Javascript es mediante el objeto **.classList**.

El objeto classList

Para trabajar más cómodamente, existe un sistema muy interesante para trabajar con clases: el objeto **classList**. Se trata de un objeto especial (*lista de clases*) que contiene una serie de ayudantes que permiten trabajar con las clases de forma más intuitiva y lógica.

Si accedemos a **.classList**, nos devolverá un (*lista*) de clases CSS de dicho elemento. Pero además, incorpora una serie de métodos ayudantes que nos harán muy sencillo trabajar con clases CSS:

Método	Descripción
.classList	Devuelve la lista de clases del elemento HTML.
.classList.item(n)	Devuelve la clase número **n** del elemento HTML.
.classList.add(c1, c2, ...)	Añade las clases **c1**, **c2**... al elemento HTML.

.classList.remove(c1, c2, ...)	Elimina las clases **c1**, **c2**... del elemento HTML.
.classList.contains(clase)	Indica si la **clase** existe en el elemento HTML.
.classList.toggle(clase)	Si la **clase** no existe, la añade. Si no, la elimina.
.classList.toggle(clase, expr)	Si **expr** es **true**, añade **clase**. Si no, la elimina.
.classList.replace(old, new)	Reemplaza la clase **old** por la clase **new**.

Veamos un ejemplo de uso de cada método de ayuda. Supongamos que tenemos el siguiente elemento HTML en nuestro documento. Vamos a acceder a él y a utilizar el objeto **.classList** con dicho elemento:

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

Observa que dicho elemento HTML tiene:

- Un atributo **id**
- Tres clases CSS: **info**, **data** y **dark**
- Un metadato HTML **data-number**

Añadir y eliminar clases CSS

Los métodos **classList.add()** y **classList.remove()** permiten indicar una o múltiples clases CSS a añadir o eliminar. Observa el siguiente código donde se ilustra un ejemplo:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]

div.classList.add("uno", "dos"); // No devuelve nada.
div.classList; // ["info", "data", "dark", "uno", "dos"]

div.classList.remove("uno", "dos"); // No devuelve nada.
div.classList; // ["info", "data", "dark"]
```

En el caso de que se añada una clase CSS que ya existía previamente, o que se elimine una clase CSS que no existía, simplemente no ocurrirá nada.

Conmutar o alternar clases CSS

Un ayudante muy interesante es el del método **classList.toggle()**, que lo que hace es **añadir o eliminar la clase CSS** dependiendo de si ya existía previamente. Es decir, añade la clase si no existía previamente o elimina la clase si existía previamente:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]

div.classList.toggle("info"); // Como "info" existe, lo elimina. Devuelve "false"
div.classList; // ["data", "dark"]

div.classList.toggle("info"); // Como "info" no existe, lo añade. Devuelve "true"
div.classList; // ["info", "data", "dark"]
```

Observa que **.toggle()** devuelve un que será **true** o **false** dependiendo de si, tras la operación, la clase sigue existiendo o no. Ten en cuenta que en **.toggle()**, al contrario que **.add()** o **.remove()**, sólo se puede indicar una clase CSS por parámetro.

Otros métodos de clases CSS

Por otro lado, tenemos otros métodos menos utilizados, pero también muy interesantes:

- El método **.classList.item(n)** nos devuelve la clase CSS ubicada en la posición **n**.
- El método **.classList.contains(name)** nos devuelve si la clase CSS **name** existe o no.
- El método **.classList.replace(old, current)** cambia la clase **old** por la clase **current**.

Veamos un ejemplo:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]

div.classList.item(1); // 'data'
div.classList.contains("info"); // Devuelve `true` (existe la clase)
div.classList.replace("dark", "light"); // Devuelve `true` (se hizo el cambio)
```

Con todos estos métodos de ayuda, nos resultará mucho más sencillo manipular clases CSS desde Javascript en nuestro código.

Navegar a través de elementos

Las propiedades que veremos a continuación devuelven información de otros elementos relacionados con el elemento en cuestión.

Propiedades de elementos HTML	Descripción
children	Devuelve una lista de elementos HTML hijos.

parentElement	Devuelve el padre del elemento o si no tiene.
firstElementChild	Devuelve el primer elemento hijo.
lastElementChild	Devuelve el último elemento hijo.
previousElementSibling	Devuelve el elemento hermano anterior o si no tiene.
nextElementSibling	Devuelve el elemento hermano siguiente o si no tiene.

En primer lugar, tenemos la propiedad **children** que nos ofrece un con una lista de elementos HTML hijos. Podríamos acceder a cualquier hijo utilizando los corchetes de array y seguir utilizando otras propiedades en el hijo seleccionado.

- La propiedad **firstElementChild** sería un acceso rápido a **children[0]**
- La propiedad **lastElementChild** sería un acceso rápido al último elemento hijo.

Por último, tenemos las propiedades **previousElementSibling** y **nextElementSibling** que nos devuelven los elementos hermanos anteriores o posteriores, respectivamente. La propiedad **parentElement** nos devolvería el padre del elemento en cuestión. En el caso de no existir alguno de estos elementos, nos devolvería .

Consideremos el siguiente documento HTML:

```
<html>
  <body>
    <div id="app">
      <div class="header">
        <h1>Titular</h1>
      </div>
      <p>Párrafo de descripción</p>
      <a href="/">Enlace</a>
    </div>
  </body>
</html>
```

Si trabajamos bajo este documento HTML, y utilizamos el siguiente código Javascript, podremos «navegar» por la jerarquía de elementos, **moviéndonos entre elementos** padre, hijo o hermanos:

```
document.body.children.length;  // 1
document.body.children;         // <div id="app">
document.body.parentElement;    // <html>

const app = document.querySelector("#app");

app.children;                   // [div.header, p, a]
app.firstElementChild;          // <div class="header">
app.lastElementChild;           // <a href="/">

const a = app.querySelector("a");

a.previousElementSibling;       // <p>
a.nextElementSibling;           // null
```