

JAVASCRIPT

CLASE 8

Material complementario

MODELO DE OBJETOS DEL DOCUMENTO (DOM)

CODER HOUSE

JavaScript: DOM

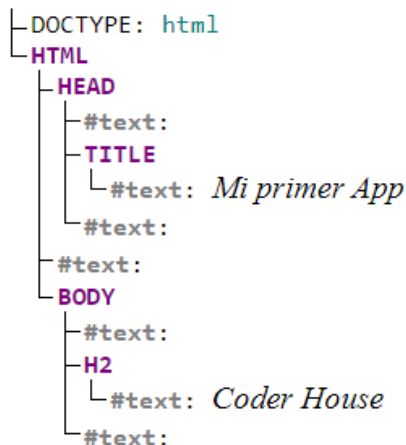
DOM: definición

El Modelo de Objetos del Documento (DOM) es una estructura que representa al documento HTML, y que podemos utilizar desde JavaScript para modificar la página actual.

Todos los navegadores construyen el DOM de forma automática, estableciendo un objeto por cada etiqueta del HTML, así como una relación de jerarquía en función de la disposición de las etiquetas anidadas. Supongamos ahora que tenemos el siguiente documento HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primer App</title>
  </head>
  <body>
    <h2>Coder House</h2>
  </body>
</html>
```

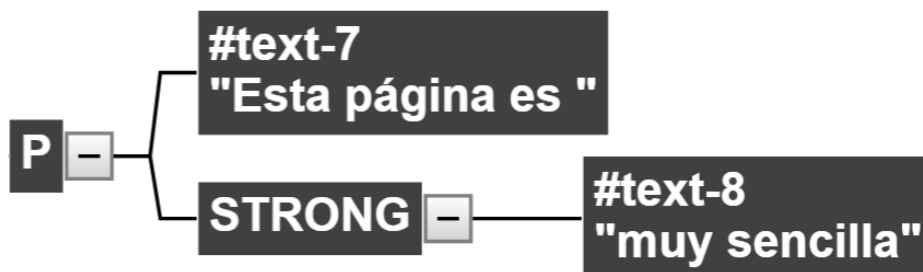
Cuando la página se carga en el navegador, se genera el DOM constituido por una jerarquía de objetos (comúnmente llamada **jerarquía de nodos**) que el programador/a puede emplear para explorar la estructura de la página web, realizar salidas y capturar entradas. Para la estructura anterior, el DOM tiene la siguiente distribución:



Como podemos observar en la estructura, existe una organización de los elementos del DOM teniendo en cuenta las etiquetas creadas dentro de otra etiqueta (anidamiento). Esta situación determina relaciones de jerarquía entre los elementos llamadas relaciones de **padre-hijo**, donde la etiqueta que contiene a otras se identifica como **padre**, y sus etiquetas anidadas como **hijo**. Analicemos la siguiente estructura para focalizar más en detalle el significado de esta relación:

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

Como podemos observar, la etiqueta párrafo (<p>) contiene dentro de sí una etiqueta ; al momento de crearse el DOM se instancia un objeto (nodo) por cada etiqueta y se determina que el nodo <p> es padre del nodo , porque está dentro de <p>. Así, la estructura resultante quedaría de la siguiente forma:



También podemos notar que en la estructura existen otros nodos cuya etiqueta no

podemos identificar en el HTML: hablamos de los [nodos de texto](#), los cuales son creados para contener las cadenas de caracteres que pueden tipearse dentro de cualquier etiqueta HTML.

Además, debemos tener presentes algunas particulares cuando hablamos del DOM:

- El DOM es independiente del lenguaje de programación: como tecnología empleada para representar documentos HTML en estructuras manipulables e integrada en todos los navegadores, cuenta con su propio estándar bien definido ([DOM Standard](#)), y puede utilizarse desde cualquier lenguaje de programación (JavaScript, Python, PHP, etcétera).
- Cada nodo del DOM tiene propiedades y métodos propios: siendo los nodos del DOM objetos, dispone de propiedades y métodos que podemos utilizar para modificarlos, eliminarlos o agregar nuevos nodos.
- El DOM se crea en el cliente: cuando el intérprete (navegador) realiza la lectura del documento HTML, genera el DOM. Dado que este último puede modificarse empleando instrucciones JavaScript, el tiempo de construcción de dicha estructura puede variar. Podemos analizar el DOM desde el cliente empleando herramientas como [Live DOM Viewer](#), [DOM node tree viewer](#), o en la propia consola para desarrolladores en la pestaña “Elements”.

Tipos de nodos

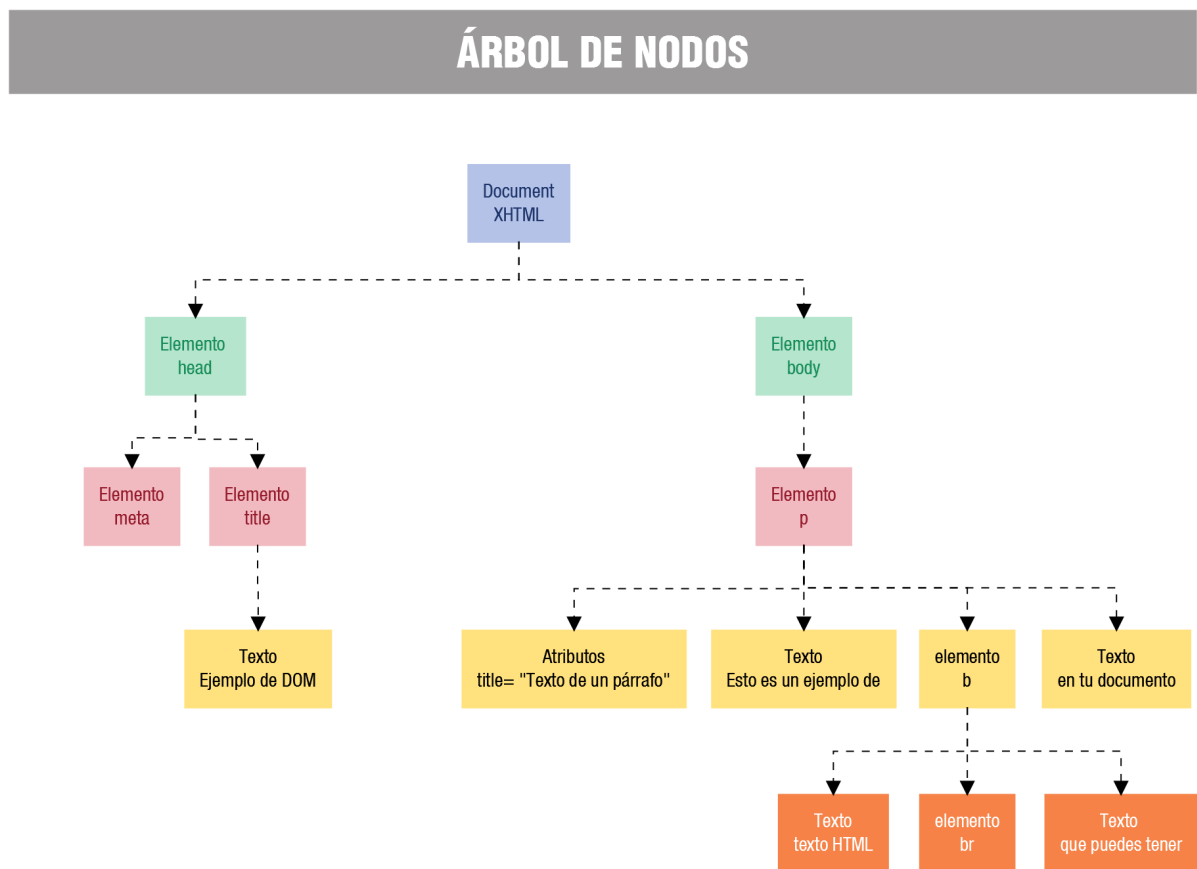
Dijimos que en el DOM se crea un objeto al que llamamos “nodo”, por cada elemento del HTML. Existen distintos tipos de nodos, entre los cuales identificamos los siguientes como esenciales:

- **Document:** nodo raíz, del cual derivan todos los demás nodos del árbol.
- **Elemento:** representa cada una de las etiquetas HTML. Estos tipos de nodos son los únicos que pueden contener atributos, y de los cuales pueden derivar otros nodos.
- **Atributo:** representa los atributos de las etiquetas HTML, es decir, un nodo por cada atributo=valor.

- Texto: es el que contiene el texto incluido en una etiqueta HTML.
- Comentario: representa los comentarios de la página.

Otros tipos de nodos pueden ser: [CdataSection](#), [DocumentFragment](#), [DocumentType](#), [Entity](#) y [ProcessingInstruction](#)

En la siguiente imagen podemos percibir la constitución de la jerarquía del DOM, también llamada **árbol de nodos**, identificando el tipo de nodo de cada objeto:



Cada nodo cuenta con la propiedad [nodeType](#) que podemos utilizar para reconocer su tipo.

Acceso al DOM

Como desarrolladores/as front-end, empleamos el DOM para modificar la interfaz del usuario, controlando las acciones que realiza en la página web con la intención de obtener entradas, y en consecuencia efectuar salidas apropiadas. Para operar sobre el DOM en JavaScript, empleamos el objeto de acceso global [document](#):

```
console.dir(document);  
console.dir(document.head)  
console.dir(document.body);
```

Podemos emplear el método [console.dir\(\)](#) para obtener un detalle por consola de las propiedades y métodos que componen dicho objeto. Mediante la propiedad **head** podemos acceder al nodo <head> del HTML, y mediante **body** al correspondiente nodo <body>. Esta última referencia será posible sólo si el script utilizado se encuentra referenciado en la página web antes de la cláusula de la etiqueta **body**, como observamos a continuación:

```
<body>  
  <h2>Coder House</h2>  
  <script src="js/main.js"></script>  
</body>
```

Utilizaremos algunos métodos de **document** con la intención de obtener, y posiblemente modificar, los nodos y sus propiedades. Inicialmente, identificamos tres formas de acceso a los elementos del DOM:

- Por identificador único: acceder a un elemento de la página empleando el valor del atributo **id**, el cual se utiliza para especificar un identificador único asociado a una sola etiqueta del documento HTML.
- Por clase: acceder a uno o más elementos de la página empleando el valor del atributo **class**, utilizado para especificar un identificador asociado a un grupo de

etiquetas del documento HTML.

- Por etiqueta: acceder a uno o más elementos de la página, usando el nombre de la etiqueta empleada (<p>, <div>, <h2>, etcétera).

Por cada forma de acceso, se cuenta con un método que podemos utilizar.

Obtener elemento por identificador

El método ***getElementById()*** sirve para acceder a un elemento del HTML, utilizando el valor del atributo ***id***. Por ejemplo:

```
//CODIGO HTML DE REFERENCIA
<div id = "app">
  <p id = "parrafo1" >Hola Mundo</p>
</div>

//CODIGO JS
let div      = document.getElementById("app");
let parrafo = document.getElementById("parrafo1");
console.log(div.innerHTML);
console.log(parrafo.innerHTML);
```

El único parámetro de método ***getElementById*** es un string que debe ser igual al valor del identificador a obtener. El método nos permite obtener un único elemento, que coincide con el notificador enviado. En caso de no existir un elemento que cumpla la igualdad, se obtiene ***null***.

Podemos obtener la estructura interna (es decir, las etiquetas anidadas) del elemento obtenido empleando la propiedad [innerHTML](#)

Obtener elementos por clase

El método ***getElementsByClassName()*** sirve para acceder a un conjunto de elementos del HTML, utilizando el valor del atributo ***class***. Porejemplo:

```
//CODIGO HTML DE REFERENCIA
<ul>
  <li class="países">AR</li>
  <li class="países">CL</li>
  <li class="países">UY</li>
</ul>

//CODIGO JS
let países = document.getElementsByClassName("países");
console.log(países[0].innerHTML);
console.log(países[1].innerHTML);
console.log(países[2].innerHTML);
```

La llamada al método retornará un conjunto de elementos con todas las coincidencias. Para acceder a cada uno de los nodos encontrados, debemos manejar el acceso de forma similar a un array.

Obtener elementos por etiqueta

El método *getElementsByTagName()* sirve para acceder a un conjunto de elementos del HTML, utilizando el nombre de etiqueta como criterio de búsqueda, por ejemplo:

```
//CODIGO HTML DE REFERENCIA
<div>
  <div>CONTENEDOR 2</div>
  <div>CONTENEDOR 3</div>
</div>

//CODIGO JS
let contenedores = document.getElementsByTagName("div");
console.log(contenedores[0].innerHTML);
console.log(contenedores[1].innerHTML);
console.log(contenedores[2].innerHTML);
```


Esta opción es la menos específica de todas, ya que es muy probable que las etiquetas se repitan en el código HTML, siendo difícil establecer un criterio adicional para diferenciarlas entre sí al no emplear *id* o *class*.

Por último, cabe aclarar que los elementos obtenidos mediante *getElementsByTagName* y *getElementsByClassName* se encuentran contenidos en un array especial identificado como [HTMLCollection](#), que podemos iterar con [for...of](#) con la intención de acceder a todos los elementos obtenidos con un recorrido:

```
let paises = document.getElementsByClassName("paises");
let contenedores = document.getElementsByTagName("div");

for (const pais of paises) {
    console.log(pais.innerHTML);
}
for (const div of contenedores) {
    console.log(div.innerHTML);
}
```

Crear y eliminar nodos

Si bien en la sección anterior reconocimos distintas formas de acceso para obtener elementos del DOM desde JavaScript, quedan por definir mecanismos para cambiar el contenido de una estructura HTML. Para ello, es necesario analizar una serie de pasos a realizar, que podemos abordar usando el siguiente ejemplo como referencia:

```
// Crear nodo de tipo Elemento, etiqueta p
let parrafo = document.createElement("p");
// Insertar HTML interno
parrafo.innerHTML = "<h2>¡Hola Coder!</h2>";
```

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

1. Crear un nodo nuevo con el método ***createElement***: el cual nos permite crear un nuevo nodo, especificando por parámetro el nombre de la etiqueta deseada (en el ejemplo optamos por un párrafo <p></p>).
2. Definir la estructura del nodo creado: ahora que tenemos un nuevo nodo, es necesario determinar cómo estará compuesto el interior del elemento. Esto podemos hacerlo empleando la propiedad ***innerHTML*** del nuevo elemento.
3. Añadir el nodo al DOM: para agregar el elemento creado, es necesario introducirlo como hijo de un elemento existente en el DOM. En el ejemplo analizado, el nodo párrafo se introduce como hijo del nodo body, usando el método [appendChild](#). Además, existe otro método alternativo que podemos emplear para agregar un nodo o una cadena a un elemento, denominado [append](#).

Así como es necesario definir un nodo padre al que agregar el nuevo nodo hijo, para eliminar un nodo es necesario hacerlo desde el nodo padre. Podemos emplear la propiedad [parentNode](#) para obtener el nodo padre, y llamar al método [removeChild](#), recibiendo por parámetro el propio nodo para eliminarlo. A continuación, un ejemplo de dicha implementación:

```
let parrafo = document.getElementById("parrafo1");
//Eliminando el propio elemento, referenciando al padre
parrafo.parentNode.removeChild(parrafo);

let paises = document.getElementsByClassName("paises");
//Eliminando el primer elemento de clase paises
paises[0].parentNode.removeChild(paises[0])
```

Y ¿por qué necesito crear nodos si podría definir las etiquetas directamente sobre el HTML? La creación del HTML con JavaScript es necesaria cuando la información de la interfaz depende de una estructura de datos, es decir que los valores en variables y objetos determinan cómo el usuario va a utilizar la interfaz, pudiendo actualizar la página web si los datos asociados se modifican. Veamos ahora un ejemplo aplicado:

```
//Obtenemos el nodo donde vamos a agregar los nuevos elementos
let padre = document.getElementById("personas");
//Array con la información a agregar
let personas = ["HOMERO", "MARGE", "BART", "LISA", "MAGGIE"];
//Iteramos el array con for...of
for (const persona of personas) {
    //Creamos un nodo <li> y agregamos al padre en cada ciclo
    let li = document.createElement("li");
    li.innerHTML = persona
    padre.appendChild(li);
}
```

Gracias al código anterior, las opciones del listado se crean teniendo en cuenta la información del array. Si se agrega o elimina un elemento de la colección del identificador personas, las opciones generadas también cambiarán. A esta forma de construir el documento HTML se la suele llamar ***generación dinámica de páginas web o HTML dinámico***, y es empleada para construir aplicaciones web interactivas, ya que se supone que la información y la estructura del sitio deben cambiar en respuesta a las acciones del usuario, y la información que este u otros usuario produzcan.

Plantillas de texto

Las plantillas de cadena, también llamadas plantillas literales, son elementos que nos permiten simplificar la concatenación de strings y variables. Sirven para agrupar los valores de variables, así como las propiedades de objetos con strings de referencia, con la intención de generar salidas más legibles, tanto para los usuarios como para los programadores/as que construyen dichos elementos. Veamos ahora un ejemplo de referencia, donde se efectúe la misma salida de dos formas diferentes:

```
let producto = { id: 1, nombre: "Arroz", precio: 125 };
let concatenado = "ID : " + producto.id + " - Producto: " + producto.nombre + "$ "+producto.precio;
let plantilla = `ID: ${producto.id} - Producto ${producto.nombre} $ ${producto.precio}`;
//El valor es idéntico pero la construcción de la plantilla es más sencilla
console.log(concatenado);
console.log(plantilla);
```

La cadena asignada a la variable concatenada se crea mediante una concatenación normal, mientras que la asignada a plantilla emplea la notación de plantilla literal, delimitada con el carácter de comillas o tildes invertidas (` `). Como podemos observar, la segunda es más acotada que la primera, e implica un reconocimiento sencillo de los datos de las propiedades añadidas al texto, identificados por el signo de dólar y envueltos en llaves (\${variable}).

Otra ventaja es que al emplear las plantillas podemos establecer saltos de línea e indentados, acción restringida con la concatenación. En el siguiente ejemplo podemos divisar cómo emplear este elemento para definir el innerHTML de un contenedor:

```
let producto = { id: 1, nombre: "Arroz", precio: 125 };
let contenedor = document.createElement("div");
//Definimos el innerHTML del elemento con una plantilla de texto
contenedor.innerHTML = `

### ID: ${producto.id}</h3> <p> Producto: ${producto.nombre}</p> <b> $ ${producto.precio}</b>`; //Agregamos el contenedor creado al body document.body.appendChild(contenedor);


```

En conclusión, las plantillas de literales nos permiten nos permiten construir un código más legible, más si nos encontramos ante la necesidad de construir HTML dinámico:

```
const productos = [{ id: 1, nombre: "Arroz", precio: 125 },
                    { id: 2, nombre: "Fideo", precio: 70 },
                    { id: 3, nombre: "Pan" , precio: 50},
                    { id: 4, nombre: "Flan" , precio: 100}];

for (const producto of productos) {
  let contenedor = document.createElement("div");
  //Definimos el innerHTML del elemento con una plantilla de texto
  contenedor.innerHTML = `<h3> ID: ${producto.id}</h3>
                          <p> Producto: ${producto.nombre}</p>
                          <b> $ ${producto.precio}</b>`;
  document.body.appendChild(contenedor);
}
```