JAVASCRIPT

# CLASE 8

# Material complementario

*MODELO DE OBJETOS DEL DOCUMENTO (DOM)*

CODER HOUSE

# JavaScript: DOM

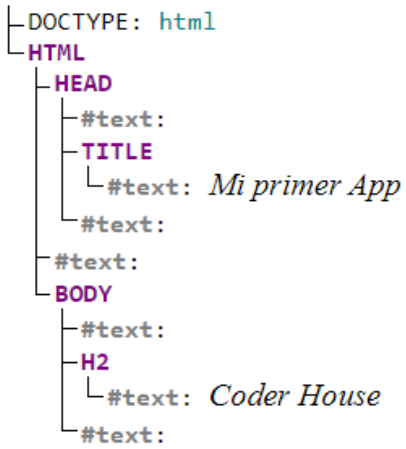
### DOM: definición

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>
```

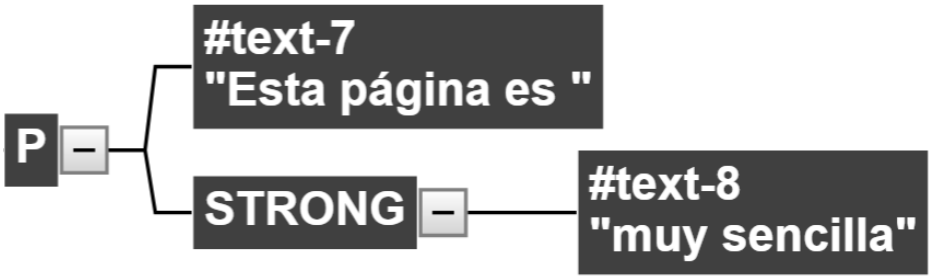
Cuando la página se carga en el navegador, se genera el DOM constituido por una jerarquía de objetos (comúnmente llamada **jerarquía de nodos**) que el programador/a puede emplear para explorar la estructura de la página web, realizar salidas y capturar entradas. Para la estructura anterior, el DOM tiene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

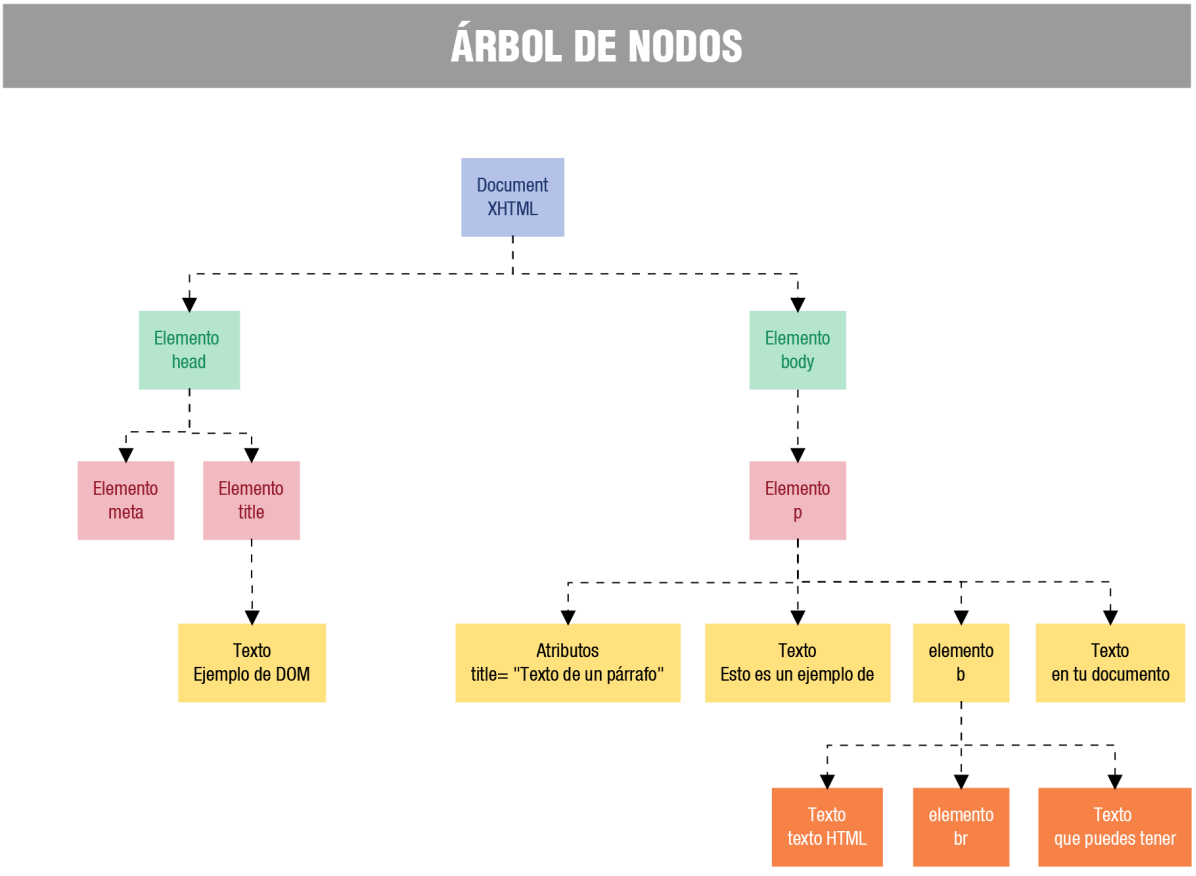
- **Document:** Nodo raíz, del cual derivan todos los demás nodos del árbol.
- **Element:** Representa cada una de las etiquetas HTML. Estos tipos de nodos son los únicos que pueden contener atributos, y de los cuales pueden derivar otros

nodos.

- **Attr:** Representa los atributos de las etiquetas HTML, es decir, un nodo por cada atributo=valor.
- **Text:** Es el que contiene el texto incluido en una etiqueta HTML.
- **Comment:** 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:




Fuente del gráfico: https://bit.ly/3DJoUuA

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>, , <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="paises">AR</li>
      <li class="paises">CL</li>
     <li class="paises">UY</li>
 </ul>
//CODIGO JS
let paises = document.getElementsByClassName("paises");
console.log(paises[0].innerHTML);
console.log(paises[1].innerHTML);
console.log(paises[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 intencion de acceder a todos los elemento 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);
}
```

# Modificar nodos

## innerText

La propiedad *innerText* de un nodo nos permite modificar su **nodo de texto**. Es decir, acceder y/o modificar el contenido textual de algún elemento del DOM.

```
//CODIGO HTML DE REFERENCIA
<h1 id="titulo">Hola Mundo!</h1>
//CODIGO JS
let titulo = document.getElementById("titulo")
console.log( titulo.innerText ) // "Hola Mundo!"

// cambio el contenido del elemento
titulo.innerText = "Hola Coder!"
console.log( titulo.innerText ) // "Hola Coder!"
```

Esto cambiaría el nodo de texto del elemento seleccionado con id "título" en el DOM. Significa que en la pantalla puedo ver reflejado el cambio en la etiqueta.

## innerHTML

Similar al innerText, **innerHTML** permite definir el **código html interno** del elemento seleccionado. La diferencia con lo anterior, es que al utilizar innerHTML el navegador lo interpreta como código HTML y no como contenido de texto, por lo que puedo asignar como string a la propiedad innerHTML de un elemento un bloque de código html que se reflejará en la forma de una nueva estructura de etiquetas y contenido interno:

```
//CODIGO HTML DE REFERENCIA
<div id=”contenedor”></div>
//CODIGO JS
let container = document.getElementById("contenedor")
// cambio el código HTML interno
container.innerHTML = “<h2>Hola mundo!</h2><p>Lorem ipsum</p>”

//Resultado en el DOM
<div id=”contenedor”>
    <h2>Hola mundo!</h2>
    <p>Lorem ipsum</p>
</div>
```

Al pasar un string con formato de etiquetas html y contenido a través de la propiedad innerHTML, el navegador la interpreta como código html y genera nuevos nodos con su contenido dentro del  seleccionado.

### className

A través de la propiedad className de algún nodo seleccionado podemos acceder al atributo class del mismo y definir cuáles van a ser sus clases. Esto es de utilidad cuando queremos generar cambios dinámicos en el estilado de elementos, combinando la asignación de clases desde JS con los estilos y clases prearmadas en CSS:

```
//CODIGO HTML DE REFERENCIA
<div id=”contenedor”></div>
//CODIGO JS
let container = document.getElementById("contenedor")
// cambio el código HTML interno
container.innerHTML = “<h2>Hola mundo!</h2><p>Lorem
ipsum</p>”
// cambio el atributo class
container.className = “container row”
```

```
//Resultado en el DOM
<div id="contenedor" class="container row">
    <h2>Hola mundo!</h2>
    <p>Lorem ipsum</p>
</div>
```

### Otras propiedades

De forma similar podemos acceder a otras propiedades y métodos de los nodos que nos permiten modificarlos. Si queremos podemos acceder y modificar los atributos **id, style, src, value, etc.,** de diversos elementos siguiendo un tratamiento similar. Para esto, es importante saber qué queremos cambiar de cada elemento y poder investigar cómo hacerlo. Recuerda que todo lo que se genera en el DOM es accesible y modificable desde JS.

# 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 un 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.append(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 ).
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 append. El método **append** inserta el nuevo elemento sobre el final del contenido del nodo padre seleccionado; si queremos insertarlo sobre el comienzo podemos utilizar el método **prepend** de forma similar.

Para eliminar un nodo del DOM basta con seleccionarlo de forma precisa a través de alguno de los métodos vistos y aplicar el método **remove()** sobre éste:

```
let parrafo      = document.getElementById("parrafo1");
//Elminando el propio elemento
parrafo.remove()

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

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

# Query Selector

Para muchos los métodos de selección de elementos vistos pueden ser incómodos y a veces confusos o imprecisos, ya que sólo podemos seleccionarlos por alguna característica específica. Por ello Javascript llegó a desarrollar el método **querySelector(),** que nos permite seleccionar nodos con la misma sintaxis que utilizamos en los **selectores de CSS**.

```
//CODIGO HTML DE REFERENCIA
<div id=”contenedor”>
    <p class=”texto”></p>
</div>
//CODIGO JS
// puedo seleccionar la etiqueta <p> siguiendo la sintaxis de
CSS para selectores:
let parrafo = document.querySelector("#contenedor p")
// o bien seleccionar sólo el contenedor por id con #
let contenedor = document.querySelector("#contenedor")
```

```
// o por clase:
parrafo = document.querySelector(".texto")
```

Lo interesante del querySelector es que también aplica a pseudo-clases de CSS, brindando un nivel más avanzado de precisión:

```
let radioChecked = document.querySelector(".radio:checked")
```

Suponiendo que tengo elementos html **radio button** y quiero seleccionar sólo aquel que esté en **checked**, ésto lo puedo lograr muy fácil con querySelector y la pseudo-clase **:checked** de CSS. Aplica también para todo tipo de selectores CSS.

QuerySelector me retorna el primer elemento que coincida con el parámetro de búsqueda, o sea un sólo elemento. Si quiero obtener una colección de elementos, es menester utilizar el método **querySelectorAll()** siguiendo el mismo comportamiento.