IES SAN ANDRÉS

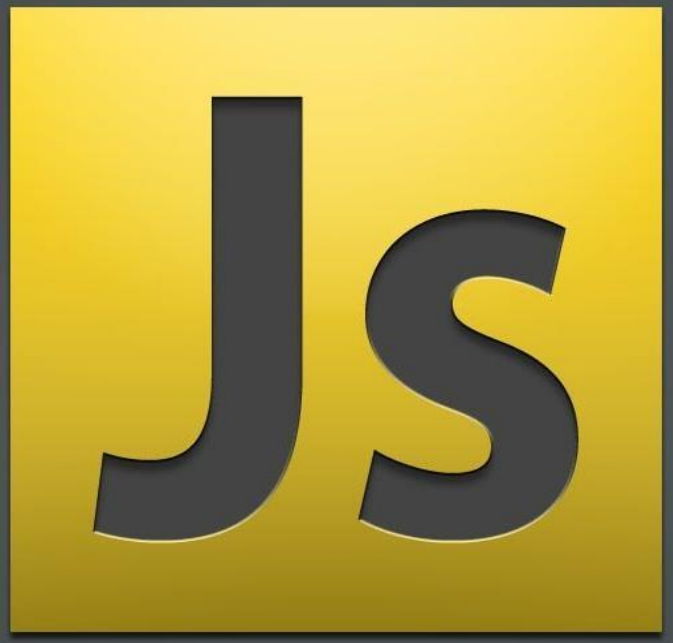
DESARROLLO DE APLICACIONES WEB

DESARROLLO WEB EN ENTORNO CLIENTE

UT04: MODELO DE OBJETOS DEL DOCUMENTO. DOM.

ÍNDICE

1

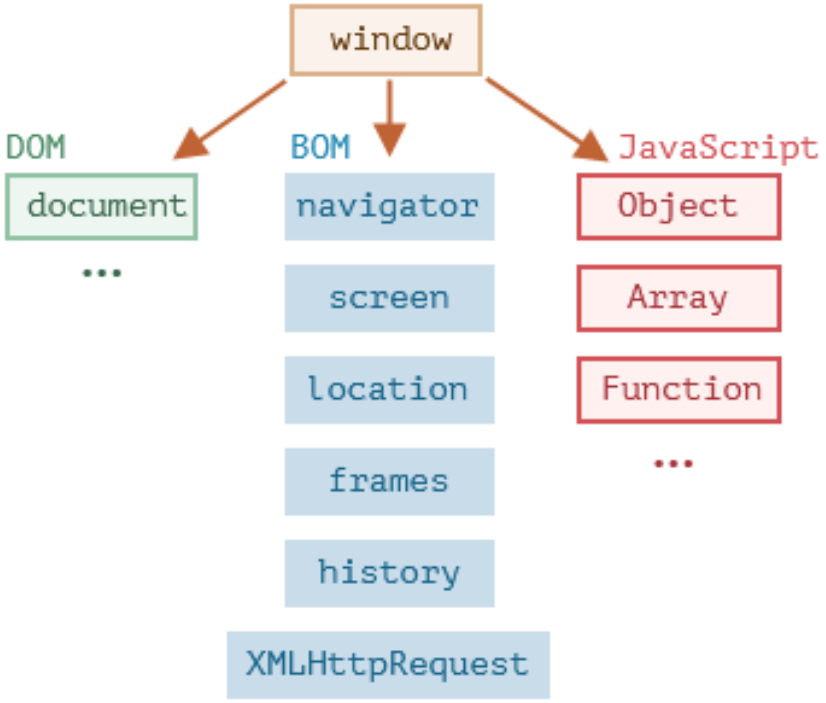


INTRODUCCIÓN AL DOM

JavaScript se puede ejecutar en diferentes plataformas (navegador, servidor web, …) y cada uno de ellos proporciona una funcionalidad específica denominada **entorno de host**.

El entorno de host añade objetos propios y funciones adicionales al núcleo de lenguaje.

En la imagen se puede ver el **entorno de host en un navegador**.



El objeto raíz se llama **window**, y tiene dos roles:

- Es el objeto global para el código JavaScript
- Representa la ventana del navegador y proporciona métodos para controlarla.

```
function saluda() {
    alert("Hola Mundo");
}

// Las funciones globales son métodos del objeto global
window.saluda();

alert(window.innerHeight); // Altura interior de la ventana
```

El **Document Object Model (DOM)** representa todo el contenido de la página como objetos que pueden ser modificados.

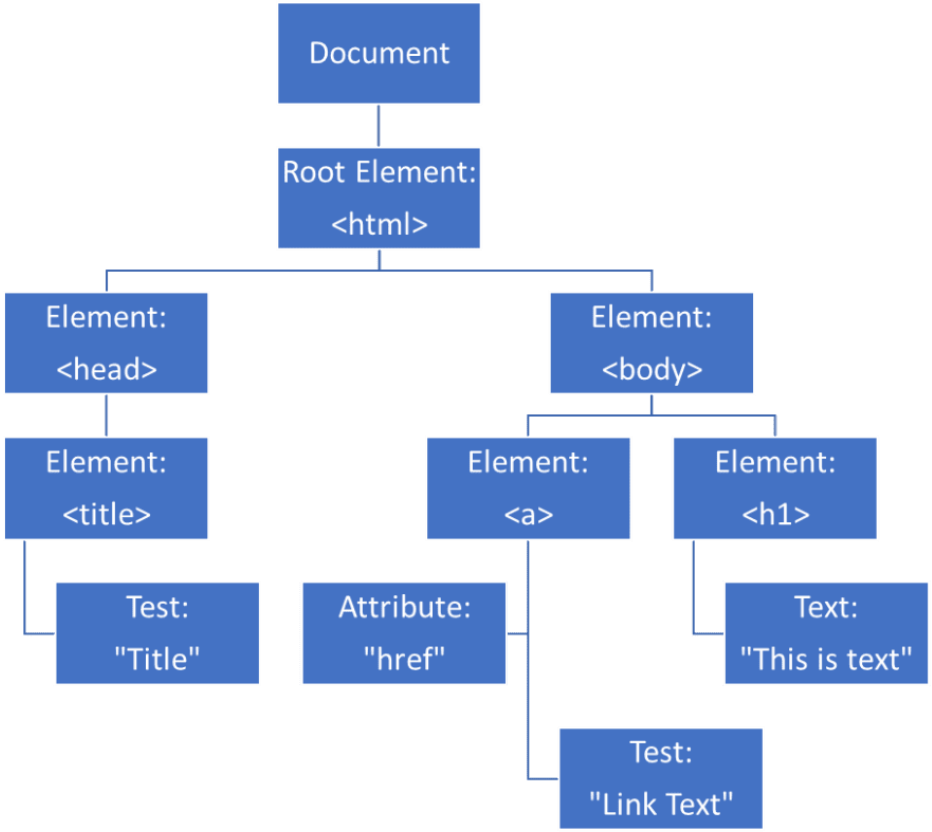
```
document.body.style.background = "red";
setTimeout( () => document.body.style.background="", 1000 );
```

El **Browser Object Model (BOM)** son objetos adicionales proporcionados por el navegador para trabajar con todo excepto el documento.

```
alert(location.href);
if ( confirm("Ir a Wikipedia?") ){
    location.href = "https://wikipedia.org";
}
```

La estructura de HTML son las etiquetas. Según DOM, **cada etiqueta HTML es un objeto**. El texto dentro de cada etiqueta también es un objeto.

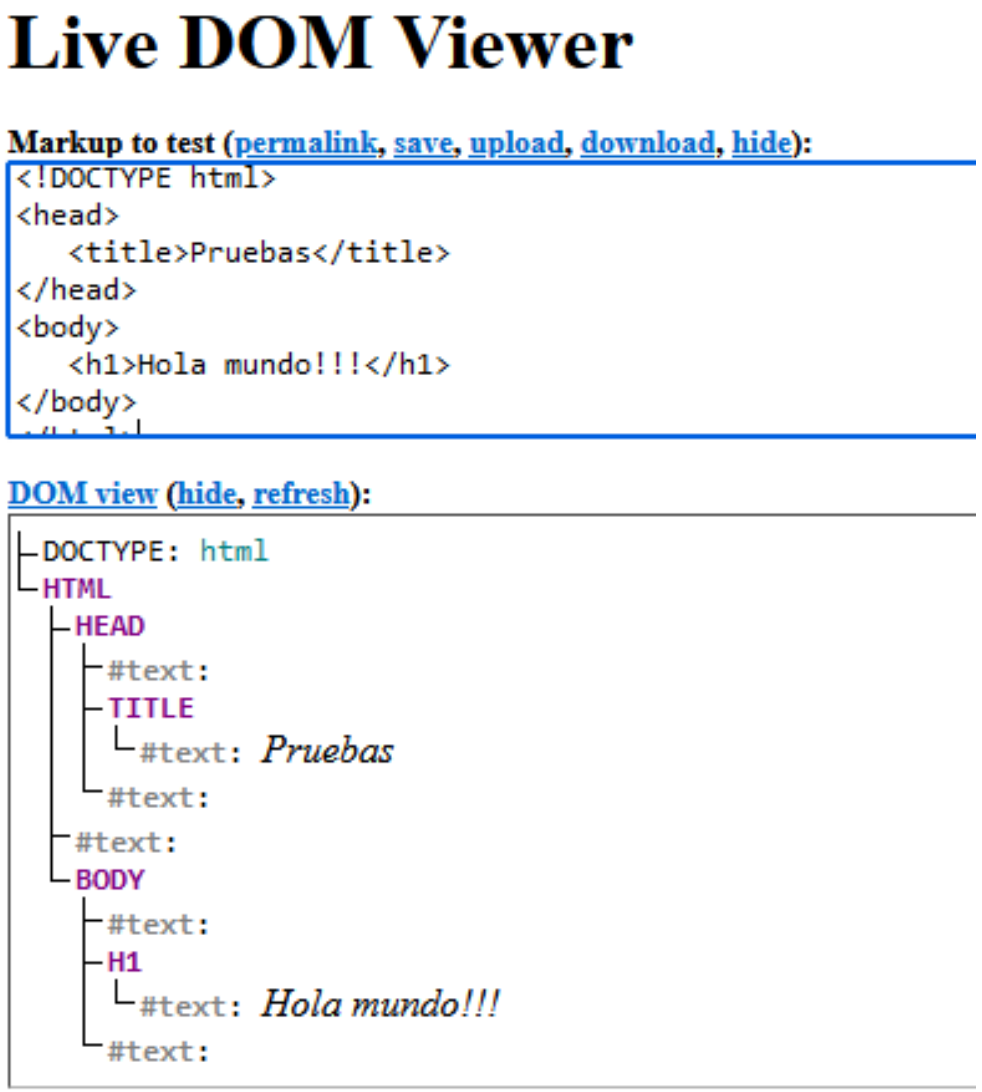
Esto hace que podamos ver una página Web como un árbol de objetos.



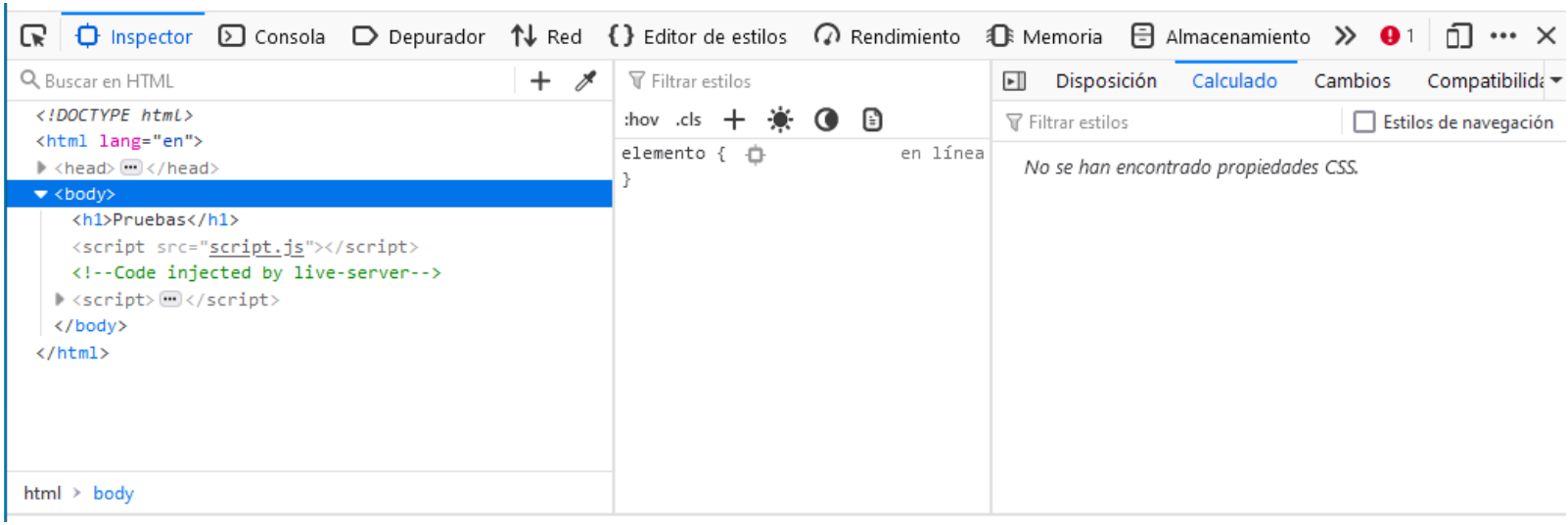
En este árbol podemos ver:

- **document:** es el punto de entrada al DOM
- **Nodos de elementos**: que corresponden a las etiquetas HTML y forman la estructura del árbol.
- **Nodos de texto**: que corresponden a texto dentro de los elementos y siempre son nodos hoja.
- **Nodos de comentario:** correspondientes a los comentarios en el HTML

Podemos ver el DOM en http://software.hixie.ch/utilities/js/live-dom-viewer/



Dentro del navegador, podemos ver el DOM en la pestaña **Inspector** de las Herramientas del desarrollador.

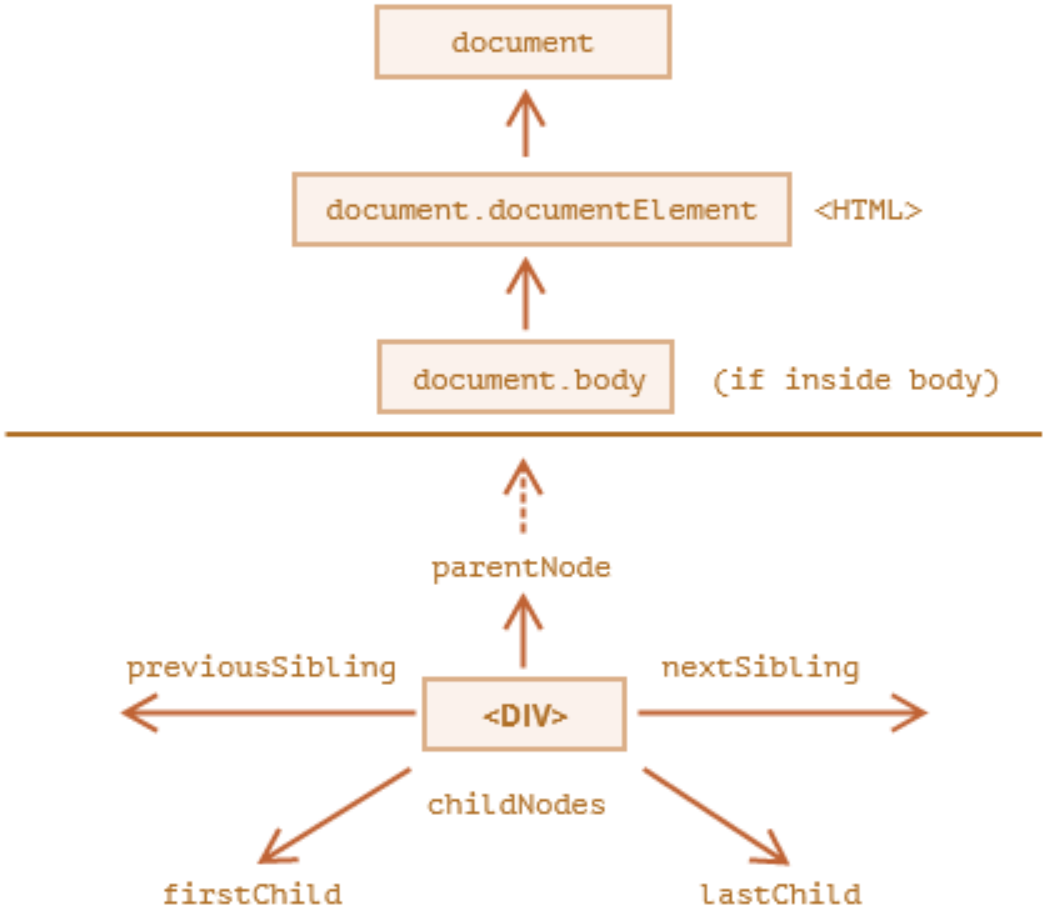


2

RECORRIENDO EL DOM

Todas las operaciones en el DOM comienzan con el objeto **document**, que es el punto de entrada principal al DOM.

A la derecha se muestran los diferentes enlaces que permiten navegar por el DOM.



document.documentElement

Es el nodo superior del documento y corresponde a la etiqueta <html>

document.body

Este nodo es el que corresponde a la etiqueta <body>

document.head

Este nodo es el que corresponde a la etiqueta <head>

Los elementos anteriores, comunes a todas las páginas Web, pueden ser accedidos directamente.

Para acceder al resto de **nodos** hay dos maneras posibles para hacerlo:

- Recorrer el árbol DOM hasta llegar al nodo deseado
- Acceder directamente al nodo

Para **recorrer el árbol de nodos** disponemos de los siguientes métodos y propiedades:

- Node.childNodes
- Node.firstChild
- Node.lastChild
- Node.parentNode
- Node.nextSibling
- Node.previousSibling

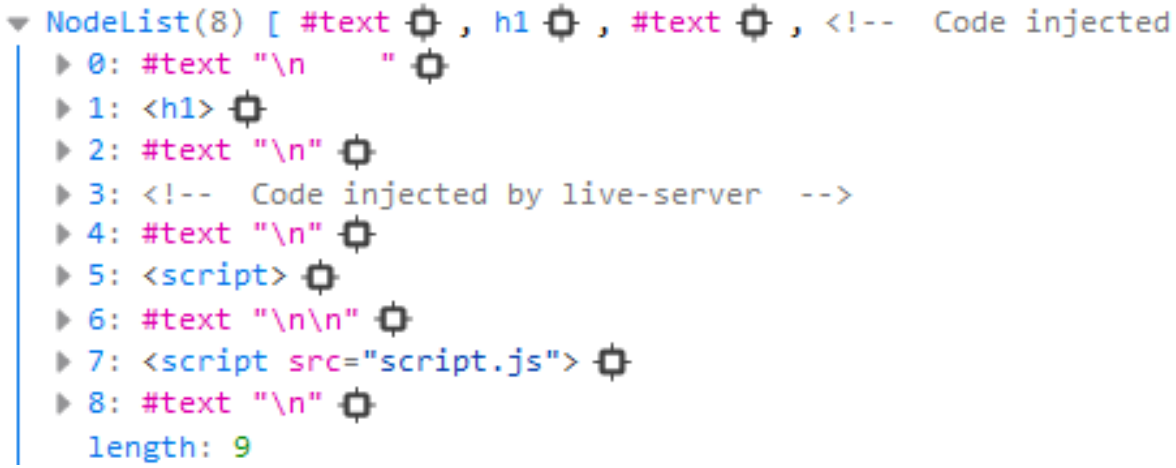
Para **acceder directamente un nodo** las opciones son:

- document.getElementById
- document.getElementsByClassName
- document.getElementsByName
- document.getElementsByTagName
- querySelector
- querySelectorAll

Node.childNodes

Es una propiedad de solo lectura que devuelve una colección de hijos de un elemento dado.

```
console.log( document.body.childNodes );
```



Hay que tener cuidado porque, aunque parece que devuelve un array, realmente es una colección, por lo que **los métodos que conocemos para trabajar con arrays no se pueden utilizar directamente**.

Si queremos utilizarlos debemos convertirlo primero a array con **Array.from.**

```
let body = document.body.childNodes
console.log( Array.isArray(body) );     // false

body = Array.from(body);
console.log( Array.isArray(body) );     // true
```

Sin embargo, si queremos acceder directamente a un elemento sí podemos utilizar la notación de corchetes.

```
console.log(document.body.childNodes[1]);    // <h1>
```

Node.firstChild y Node.lastChild

Permite un acceso rápido a la primera y a la última propiedad hijo de un elemento.

```
console.log( document.body.firstChild );
console.log( document.body.lastChild );
```

Node.parentNode

Cada nodo del árbol DOM tiene un único nodo padre. Se puede obtener utilizando la propidad **parentNode**.

Node.nextSibling y Node.previousSibling

Esta propiedad sirve para acceder a los nodos hermanos de un nodo determinado, considerándose como nodos hermanos aquellos que tienen un mismo padre.

Si un nodo no tiene un nodo hermano en la posición solicitada contendrá *null*

document.getElementById(*id*)

Las propiedades anteriores permiten acceder a un nodo desde otro nodo, pero normalmente es más cómodo acceder a un nodo directamente.

La función **getElementById()** permite obtener un nodo en función de su valor en el atributo de HTML id.

```
<body>
    <h1 id="titulo">Pruebas</h1>
    <h2 id="subtitulo">Este es el H2</h2>
</body>
```

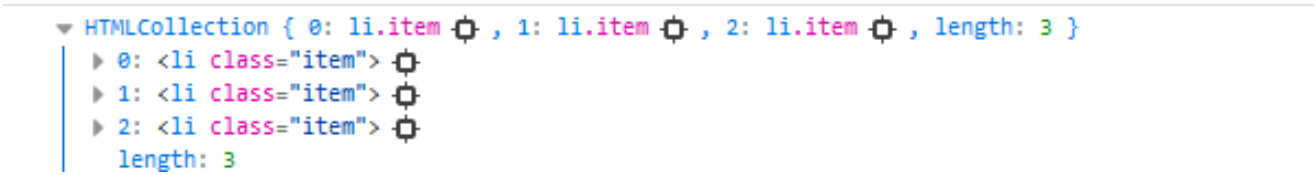
```
let titulo = document.getElementById('titulo');
console.log(titulo.textContent);                    // Pruebas
```

document.getElementsByClassName(*className*)

De forma análoga al anterior, este método devuelve una colección con todos los nodos del árbol DOM que tienen el valor indicado en la etiqueta *class* de HTML.

```
<h1 id="titulo">Pruebas</h1>
    <ul>
        <li class="item">Primero</li>
        <li class="item">Segundo</li>
        <li class="item">Tercero</li>
    </ul>
```

```
let a = document.getElementsByClassName('item');
console.log(a);
```



Accediendo directamente a nodos

Si en lugar de invocar el método desde *document* lo hacemos desde otro nodo nos devolverá los nodos que tengan dicha clase y estén contenidos en el subárbol que parte de dicho nodo.

document.getElementsByName(*name*)

Similar al anterior, pero utilizando el atributo *name* de HTML.

```
Cats:
<input name="animal" type="checkbox" value="Cats">
Dogs:
<input name="animal" type="checkbox" value="Dogs">
```

```
let num = document.getElementsByName("animal").length;
document.getElementById("demo").innerHTML = num;
```

document.getElementsByTagName(*tagName*)

En este caso el criterio que se utiliza es el nombre de la etiqueta HTML.

Se puede utilizar el asterisco (*) como comodín que representa todos los elementos

```html
<h1 id="titulo">Pruebas</h1>
    <ul>
        <li class="item">Primero</li>
        <li class="item">Segundo</li>
        <li class="item">Tercero</li>
    </ul>
```

```js
let a = document.getElementsByTagName('li');
console.log(a);
```

```
▶ HTMLCollection { 0: li.item , 1: li.item , 2: li.item , length: 3 }
GET http://127.0.0.1:5500/favicon.ico
```

Todos los elementos **getElementsBy*** devuelven una **colección viva**. Esto quiere decir que almacena el estado actual del documento, pero si este cambia se actualizan automáticamente.

```
<div>First div</div>

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length);          // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length);          // 2
</script>
```

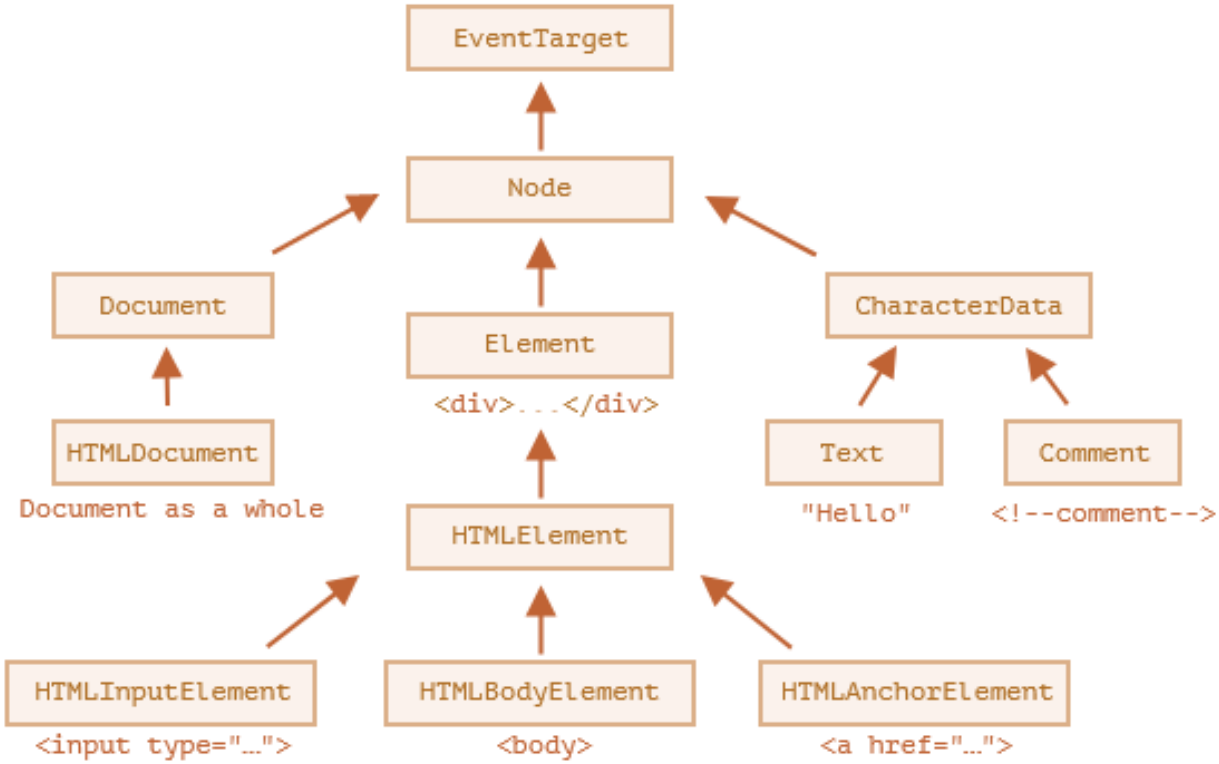
En cambio, **querySelectorAll** devuelve una **colección estática** que refleja siempre el estado del documento en el momento en que se ejecutó el método.

```
<div>First div</div>

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length);           // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length);           // 1
</script>
```

3

ATRIBUTOS Y PROPIEDADES

Todos los comandos anteriores devuelven uno o varios nodos de DOM, pero estos pueden ser de varias clases diferentes, organizadas en la siguiente jerarquía.



Propiedades del DOM

Víctor J. González

- **EventTarget**: clase abstracta que proporciona soporte para eventos al resto de clases.
- **Node**: otra clase abstracta que proporciona funcionalidad básica como las propiedades *parentNode*, *nextSibling*, …
- **Document**: contiene el documento completo. El objeto global **document** corresponde con esta clase.
- **CharacterData**: clase abstracta de la que heredan las clases:
  - **Text**: los textos que hay dentro de los elementos
  - **Comment**: comentarios

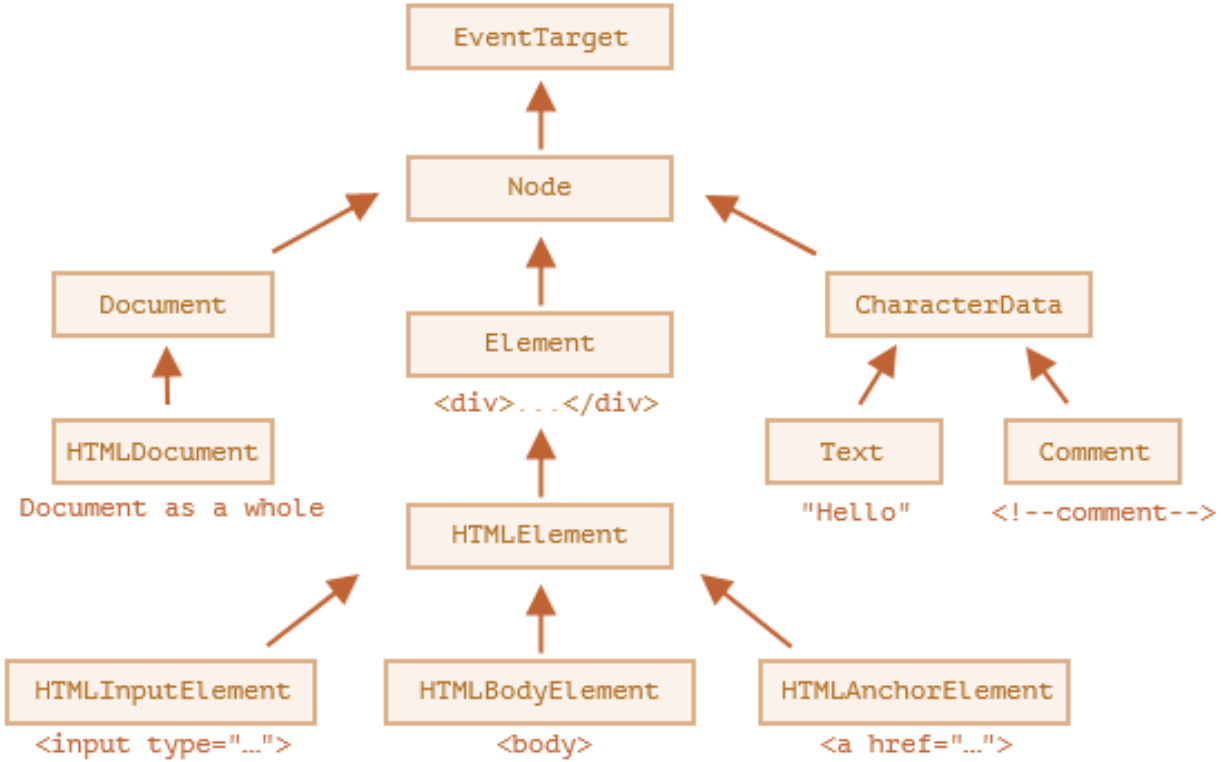
Propiedades del DOM

- **Element**: es la clase base para los elementos del DOM. Proporciona métodos de navegación como *nextElementSibling* o *children* o de búsqueda como *getElementsByTagName* o *querySelector*.
- **HTMLElement**: es la clase base para todos los elementos HTML. De esta clase heredan otras como:
  - **HTMLInputElement**: para los elementos <input>
  - **HTMLBodyElement**: para los elementos <body>
  - **HTMLAnchorElement**: para los elementos <a>

Propiedades del DOM

Víctor J. González

De esto se deduce que el conjunto de propiedades y métodos de un nodo **viene como resultado de la cadena de herencia**.



Propiedades del DOM

Víctor J. González

Podemos saber el nombre de la clase de un elemento usando su método **toString** o haciendo referencia a la propiedad **name** de su constructor.

```
let elem = document.getElementById('div');
console.log( elem.toString() );
console.log( elem.constructor.name );
```

```
[object HTMLDivElement]
HTMLDivElement
```

Alternativamente podemos utilizar **instanceOf** para comprobar la clase del objeto.

Además, ten en cuenta que, como hereda de múltiples clases, será verdadero para todas ellas.

```
let elem = document.getElementById('div');
console.dir( elem instanceof HTMLBodyElement);  // false
console.dir( elem instanceof HTMLDivElement);   // true
console.dir( elem instanceof HTMLElement);      // true
console.dir( elem instanceof Element);          // true
console.dir( elem instanceof Node);             // true
console.dir( elem instanceof EventTarget);      // true
```

Nombre de la etiqueta: nodeName y tagName

Podemos saber el nombre de la etiqueta que corresponde a un nodo con las propiedades **nodeName** y **tagName**.

Diferencias:

- **tagName** solo existe para los elementos.
- En nodos que no son elementos, **nodeName** contiene una cadena con el tipo de nodo

```
let elem = document.getElementById('div');
console.log( elem.tagName );          // DIV
```

Contenido interno: innerHTML

La propiedad **innerHTML** permite obtener el HTML dentro de la etiqueta.

```
<h1>Pruebas</h1>

    <div id="div">
        <p>Hola mundo</p>
    </div>
```

```
let elem = document.getElementById('div');
console.log( elem.innerHTML );          // <p>Hola mundo</p>
```

Esta propiedad es de **lectura y escritura**, por lo que nos **permite modificar el contenido de la página.**

```
<h1>Pruebas</h1>
    <div id="div">
        <p>Hola mundo</p>
    </div>
```

```
let elem = document.getElementById('div');
elem.innerHTML = '<b>Desarrollo de Aplicaciones Web</b>';
```

Pruebas

Desarrollo de Aplicaciones Web

HTML completo del elemento: outerHTML

Básicamente es como el innerHTML, pero añadiendo el elemento en sí.

```
<h1>Pruebas</h1>

    <div id="div">
        <p>Hola mundo</p>
    </div>
```

```
let elem = document.getElementById('div');
console.log( elem.outerHTML );  // <div id="div">
                                //<p>Hola mundo</p></div>
```

Contenido del nodo de texto: nodeValue/data

Para los nodos que no heredan de *Element* disponemos de las propiedades **nodeValue** y **data**.

```
<body>
    Hola
    <!-- Comentario -->
    <script>
      let text = document.body.firstChild;
      console.log(text.data);   // Hola

      let comment = text.nextSibling;
      console.log(comment.data); // Comentario
    </script>
</body>
```

Texto puro: textContent

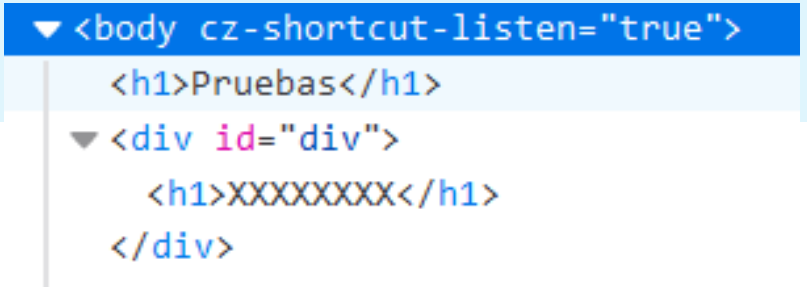
La propiedad **textContent** devuelve únicamente el texto dentro de un elemento (sin las etiquetas)

```
<body>
    <h1>Pruebas</h1>
    <div id="div">
        <p>Hola mundo</p>
    </div>
    <script src="script.js"></script>
</body>
```

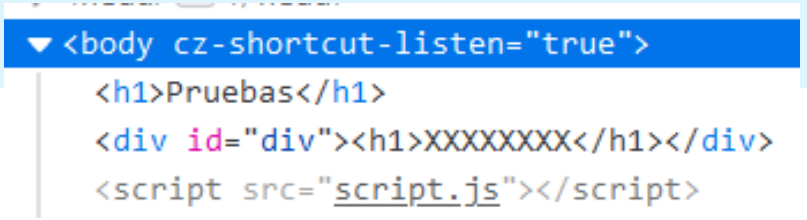
```
console.log( document.body.textContent );   // Pruebas
                                            // Hola mundo
```

Por norma general es preferible utilizar esta propiedad para modificar el contenido de un elemento ya que, si se incluyeran etiquetas HTML *maliciosamente* en el texto no las trataría como tales, sino simplemente como texto.

```
let elem = document.getElementById('div');
let str = '<h1>XXXXXXXX</h1>';
elem.innerHTML = str;
```




```
let elem = document.getElementById('div');
let str = '<h1>XXXXXXXX</h1>';
elem.textContent = str;
```



Ocultar etiquetas: propiedad hidden

El atributo **hidden** de HTML indica si un elemento está visible, por lo que podemos utilizar la propiedad homónima para ocultar un elemento en el DOM.

```
    <h1>Pruebas</h1>
    <div id="div">
        <p>Hola mundo</p>
    </div>
    <h1>Fin</h1>
```

```
let elem = document.getElementById('div');
elem.hidden = true;
```

Pruebas

Fin

El contenido de un campo: value

Esta propiedad pertenece a los elementos <input>, <select> y <textarea> y contiene el valor del mismo.

```
    <h1>Pruebas</h1>
    <label for="txt">Introduce un valor:</label>
    <input type="text" name="txt" id="input_text">
```

```
let txt = document.getElementById('input_text');
console.log(txt.value) // 1234
```

Enlaces: href

En el caso de la etiqueta <a> podemos obtener el valor del atributo href con la propiedad del mismo nombre.

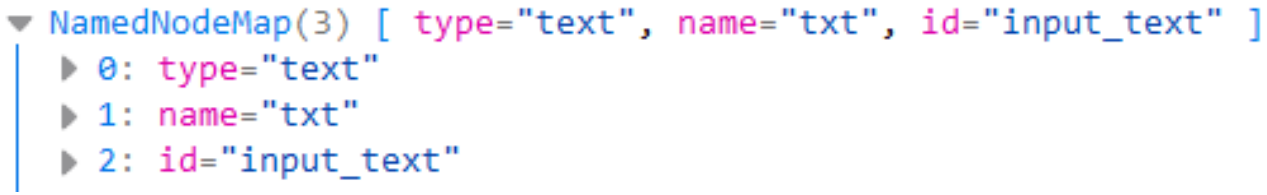
El atributo id

Otro atributo al que en ocasiones querremos acceder es el **id**, el cual se puede obtener con la propiedad del mismo nombre.

Esto se puede extrapolar a todos los atributos HTML estándar, de forma que siempre se corresponden con una propiedad homónima.

Aparte de una propiedad por atributo, todos los elementos disponen de la propiedad **attributes** que contienen un array con todos los atributos.

```
let txt = document.getElementById('input_text');
console.log(txt.attributes)
```



Propiedades de los elementos

Víctor J. González

La mayoría de las propiedades que corresponden a atributos contienen su valor como un *string*, pero hay excepciones:

- La propiedad *input.checked* para casillas de verificación es un booleano.
- El valor de la propiedad *style* se indica como cadena en el atributo, pero la propiedad almacena un objeto de tipo **CSSStyleDeclaration**.

Propiedades de los elementos

Víctor J. González

También se pueden manipular los atributos con los siguientes métodos:

`elem.hasAttribute(nombre)` – comprueba si existe.

`elem.getAttribute(nombre)` – obtiene el valor.

`elem.setAttribute(nombre, valor)` – establece el valor.

`elem.removeAttribute(nombre)` – elimina el atributo.

Estas funciones, aparte de servir para manipular los atributos HTML estándar, los podemos utilizar para trabajar con **atributos no estándar**.

Normalmente estos atributos se utilizan para pasar datos personalizados a HTML o para *marcar* elementos HTML para JavaScript.

Propiedades de los elementos

Víctor J. González

```
<!-- marque el div para mostrar "nombre" aquí -->
<div show-info="nombre"></div>
<!-- y "edad" aquí -->
<div show-info="edad"></div>

<script>
 // el código encuentra un elemento con la marca y muestra lo
que se solicita
  let user = {
    nombre: "Pete",
    edad: 25
  };

  for(let div of document.querySelectorAll('[show-info]')) {
    // inserta la información correspondiente en el campo
    let field = div.getAttribute('show-info');
    div.innerHTML = user[field]; // primero Pete en "nombre",
luego 25 en "edad"
  }
</script>
```

Propiedades de los elementos

Víctor J. González

```
<style>
     /* los estilos se basan en atributo "order-state" */
     .order[order-state="nuevo"] { color: green; }
     .order[order-state="pendiente"] { color: blue; }
     .order[order-state="cancelado"] { color: red; }
</style>

<div class="order" order-state="nuevo">
     Un nuevo pedido.
</div>

<div class="order" order-state="pendiente">
     Un pedido pendiente.
</div>

<div class="order" order-state="cancelado">
     Un pedido cancelado
</div>
```

¿Por qué usar atributos personalizados en lugar de clases? Porque los atributos son más sencillos de administrar, por ejemplo, se puede cambiar el estado del ejemplo anterior con la siguiente orden

```
// un poco más simple que eliminar/agregar clases
div.setAttribute('order-state', 'canceled')
```

¿Y qué pasa si creamos un atributo personalizado y luego el estándar introduce uno con el mismo nombre? Para evitar estos posibles conflictos el estándar contempla los **atributos data-***

Estos atributos están reservados para programadores y están disponibles a través de una propiedad del elemento llamada **dataset**.

En esta propiedad se les elimina el prefijo **data-** y si están formados por varias palabras se pasan a formato camelCase.

Propiedades de los elementos

Víctor J. González

```
<style>
.order[data-order-state="nuevo"] { color: green; }
.order[data-order-state="pendiente"] { color: blue; }
.order[data-order-state="cancelado"] { color: red; }
</style>

<div id="order" class="order" data-order-
state="nuevo">
Una nueva orden.
</div>

<script>
// leer
alert(order.dataset.orderState); // nuevo
// modificar
order.dataset.orderState = "pendiente"; // (*)
</script>
```

4

MODIFICANDO EL DOCUMENTO: ESTILOS Y CLASES

Modificando el DOM

Víctor J. González

Modificar el DOM es el elemento clave para crear páginas Web dinámicas.

Desde JavaScript podemos crear elementos, insertarlos, modificarlos o eliminarlo.

Creación de un elemento

Víctor J. González

document.createElement(*tag*)

Crea un **nodo elemento** HTML especificado por su nombre de etiqueta que se pasa como parámetro.

El elemento creado está vacío y además no se incluye en el árbol DOM, sino que habrá que hacerlo posteriormente.

Al nodo creado se le pueden modificar las propiedades:

```
// 1. Crear elemento <div>
let div = document.createElement('div');
// 2. Establecer su clase a "alert"
div.className = "alert";
// 3. Agregar el contenido
div.innerHTML = "<strong>¡Hola!</strong> Mundo";
```

Creación de un nodo de texto

Víctor J. González

document.createTextNode(*text*)

Crea un nuevo **nodo texto** con el texto que se le pasa como parámetro.

Métodos de inserción

Una vez creado un nodo, será necesario insertarlo en algún lugar del documento.

Para ello disponemos de las siguientes funciones:

- **node.append():** agrega nodos al final de *node*
- **node.prepend():** agrega nodos al principio de *node*
- **node.before():** agrega nodos antes de *node*
- **node.after():** agrega nodos después de *node*
- **node.replaceWith():** reemplaza *node* con los nodos dados

Métodos de inserción

Víctor J. González

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

```
<ol id="ol">
    <li>0</li>
    <li>1</li>
    <li>2</li>
</ol>

<script>
ol.before('before'); // inserta string "before" antes de <ol>
ol.after('after');  // inserta string "after" después de <ol>

let liFirst = document.createElement('li');
liFirst.innerHTML = 'prepend';
ol.prepend(liFirst); // inserta liFirst al principio de <ol>

let liLast = document.createElement('li');
liLast.innerHTML = 'append';
ol.append(liLast); // inserta liLast al final de <ol>
</script>
```

Eliminación de nodos

Para eliminar un nodo disponemos de la función **node.remove()**

Clonando nodos

Se puede clonar un nodo con **node.cloneNode(** *deep* **)**. El argumento es un booleano que puede ser:

- **true:** realiza una clonación profunda, clonando todo el árbol de nodos dentro de *node*
- **false:** únicamente clona el nodo, no sus nodos hijo.

Manipulación de clases

Víctor J. González

Uno de los usos más habituales de manipulación del DOM es para aplicar estilos de forma dinámica.

Hay dos formas de hacer esto:

- Crear clases css y agregarlas con JavaScript
- Escribir las propiedades directamente en *style*

Lo más aconsejable es utilizar clases salvo que queramos calcular los valores del CSS en tiempo de ejecución.

La propiedad que corresponde al atributo *class* es **elem.className**.

```
<h1>Desarrollo de Aplicaciones Web</h1>
<h2>Desarrollo Web en Entorno Cliente</h2>
<div id="main" class="container">
    DWEC
</div>
```

```
let container = document.getElementById('main');
console.log(container.className);        // container
container.className = "section";
console.log(container.className);        // section
```

El valor de *className* es una cadena, por lo que puede ser complicado añadir o quitar clases.

Para ello hay otra propiedad llamada **elem.classList** que dispone de una serie de funciones para manipular las clases del elemento.

- **elem.classList.add/remove():** agrega o elimina una clase
- **elem.classList.toggle():** agrega la clase si no existe, si no la elimina.
- **elem.classList.contains():** devuelve *true* si el elemento tiene la clase indicada.

Asignación directa de estilos

Víctor J. González

La otra posibilidad para aplicar estilos es manipular directamente la propiedad **elem.style**.

```
let container = document.getElementById('main');
main.style.color = 'blue';
```

Hay una propiedad dentro de *style* para cada estilo. En el caso de **propiedades de varias palabras** se utiliza la nomenclatura camelCase.

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Asignación directa de estilos

Víctor J. González

Si queremos eliminar una propiedad podemos asignarle la cadena vacía.

```
document.body.style.backgroundColor = "blue";

setTimeout(() => document.body.style.backgroundColor = "", 1000);
```

Asignación directa de estilos

Víctor J. González

Con las propiedades anteriores podemos modificar propiedades una a una, pero no se pueden asignar todas directamente ya que *style* es de solo lectura.

Para añadir múltiples propiedades simultáneamente hay que usar la propiedad especial **style.cssText**.

```
<div id="div">Button</div>

<script>
  div.style.cssText=`color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center;
  `;
</script>
```

5

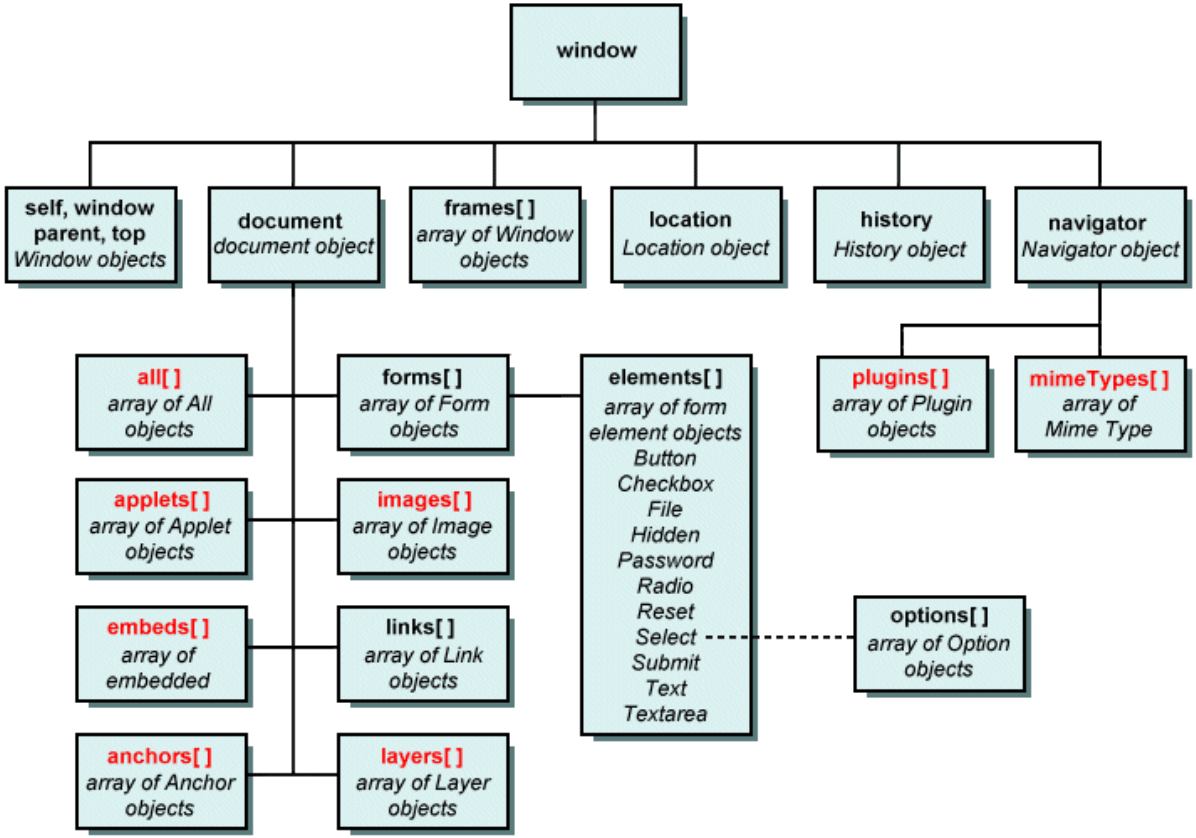
BROWSER OBJECT MODEL

El **BOM (Browser Object Model)** permite acceder y modificar las propiedades de las ventanas del propio navegador.

Algunas de las funciones que permite son:

- Crear, mover, redimensionar y cerrar ventanas del navegador.
- Obtener información sobre el propio navegador
- Propiedades de la página actual
- Gestión de cookies

El BOM está compuesto por varios objetos organizados según la siguiente jerarquía.



El objeto window

El objeto **window** es la raíz del resto de objetos y, además, proporciona métodos para manipular el tamaño y posición de la ventana:

- **moveBy(x, y):** desplaza la ventana la distancia indicada
- **moveTo(x, y):** mueve la ventana a las coordenadas indicadas
- **resizeBy(x, y):** incrementa o decrementa el tamaño de la ventana en los valores indicados por los parámetros.
- **resizeTo(x, y):** redimensiona la ventana al tamaño indicado

El objeto window

Víctor J. González

Dado que todos los métodos anteriores se consideran bastante intrusivos la mayoría de **los navegadores impiden su uso**, por lo que tenemos que usarlas con precaución.

El objeto document

Este objeto pertenece tanto al DOM como al BOM y permite acceder a información sobre la propia página.

Algunas propiedades que tiene son:

- **lastModified**: fecha de la última modificación de la página
- **referrer**: URL desde la que se accedió a la página
- **title**: texto de la etiqueta <title>
- **URL**: la dirección de la página actual del navegador

El objeto window

Víctor J. González

Tanto **title** como **URL** son propiedades de lectura y escritura, por lo que desde JavaScript podemos modificar el título de la página o llevar al usuario a otra página diferente.

El objeto window

Víctor J. González

El objeto window también dispone de varios arrays con información relevante de la página:

- **anchors**: todas las anclas de la página
- **applets**: todos los applets de la página
- **embeds**: todos los objetos embebidos mediante la etiqueta <embed>
- **forms**: todos los formularios de la página
- **images**: todas las imágenes de la página
- **links**: todos los enlaces

El objeto location

Representa la URL de la página HTML, con propiedades útiles para acceder a la misma:

- **hash**: el contenido de la URL que se encuentra a continuación del signo # (enlaces de las anclas)
- **host**: nombre del servidor
- **hostname**: suele coincidir con host
- **href**: URL completa de la página actual

El objeto location

Víctor J. González

- **pathname**: todo el texto que hay después del host
- **port**: si se especifica en la URL, puerto por el que se ha accedido.
- **protocol**: texto que hay antes de las dobles barras (//)
- **search**: todo lo que se encuentra tras la interrogación, es decir, la *query string*

El objeto location

Además, proporciona algunos métodos, como:

- **location.assign(**url**)**: cambia la URL del navegador
- **location.replace(**url**):** similar al anterior, salvo que se borra la página actual del array *history* del navegador
- **location.reload(**bool**)**: recarga la página. Si el parámetro pasado es *true* se carga la página del navegador mientras que si es *false* lo hace desde la caché

El objeto location

Víctor J. González

```
▼ Location http://127.0.0.1:5500/index.html
  ▶ assign: function assign()
    hash: ""
    host: "127.0.0.1:5500"
    hostname: "127.0.0.1"
    href: "http://127.0.0.1:5500/index.html"
    origin: "http://127.0.0.1:5500"
    pathname: "/index.html"
    port: "5500"
    protocol: "http:"
  ▶ reload: function reload()
  ▶ replace: function replace()
    search: ""
  ▶ toString: function toString()
  ▶ valueOf: function valueOf()
    Symbol(Symbol.toPrimitive): undefined
```

El objeto navigator

Este objeto incluye información sobre el propio navegador:

- **appCodeName**: cadena que representa el nombre del navegador
- **appName**: nombre oficial del navegador
- **appVersion**: versión del navegador
- **browserLanguaje**: idioma del navegador
- **cookieEnabled**: indica si las cookies están habilitadas

El objeto navigator

- **platform:** plataforma sobre la que se ejecuta el navegador
- **plugins**: lista de plugins instalados en el navegador

El objeto navigator

Víctor J. González

```
▼ Navigator { permissions: Permissions, mimeTypes: MimeTypeArray, plugins: PluginArray, pdfViewerEnabled: true, doNotTrack: "unsp
vendorSub: "" }
    appCodeName: "Mozilla"
    appName: "Netscape"
    appVersion: "5.0 (Windows)"
    buildID: "20181001000000"
  ▶ clipboard: Clipboard {  }
    cookieEnabled: true
  ▶ credentials: CredentialsContainer {  }
    doNotTrack: "unspecified"
  ▶ geolocation: Geolocation {  }
    hardwareConcurrency: 12
    language: "es-ES"
  ▶ languages: Array(4) [ "es-ES", "es", "en-US", … ]
  ▶ locks: LockManager {  }
    maxTouchPoints: 0
  ▶ mediaCapabilities: MediaCapabilities {  }
  ▶ mediaDevices: MediaDevices { ondevicechange: null }
  ▶ mediaSession: MediaSession { metadata: null, playbackState: "none" }
  ▶ mimeTypes: MimeTypeArray { 0: MimeType, 1: MimeType, length: 2, … }
    onLine: true
    oscpu: "Windows NT 10.0; Win64; x64"
    pdfViewerEnabled: true
  ▶ permissions: Permissions {  }
    platform: "Win32"
  ▶ plugins: PluginArray { 0: Plugin, 1: Plugin, length: 5, … }
    product: "Gecko"
    productSub: "20100101"
  ▶ serviceWorker: ServiceWorkerContainer { controller: null, ready: Promise { "pending" }, oncontrollerchange: null, … }
  ▶ storage: StorageManager {  }
    userAgent: "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/119.0"
    vendor: ""
    vendorSub: ""
    webdriver: false
```

El objeto screen

Información sobre la pantalla del usuario:

- **availHeight**: altura de la pantalla disponible para las ventanas
- **availWidth**: ancho de pantalla disponible para las ventanas
- **colorDepth**: profundidad de color de la pantalla
- **height**: altura total de la pantalla en píxeles
- **width**: ancho total de la pantalla en píxeles

El objeto screen

Víctor J. González

```
▼ Screen { availWidth: 1920, availHeight: 1040, width: 1920, height: 1080, colorDepth: 24, pi
    availHeight: 1040
    availLeft: -1920
    availTop: 0
    availWidth: 1920
    colorDepth: 24
    height: 1080
    left: -1920
    mozOrientation: "landscape-primary"
    onmozorientationchange: null
  ▶ orientation: ScreenOrientation { type: "landscape-primary", angle: 0, onchange: null }
    pixelDepth: 24
    top: 0
    width: 1920
  ▶ <prototype>: ScreenPrototype { mozLockOrientation: mozLockOrientation(), mozUnlockOrient
```

Fuente: https://www.arkaitzgarro.com/javascript/capitulo-14.html

6

CONTROL DE TIEMPOS

Hay ocasiones en que queremos esperar un tiempo para realizar alguna acción o bien programar alguna acción para que se realice periódicamente.

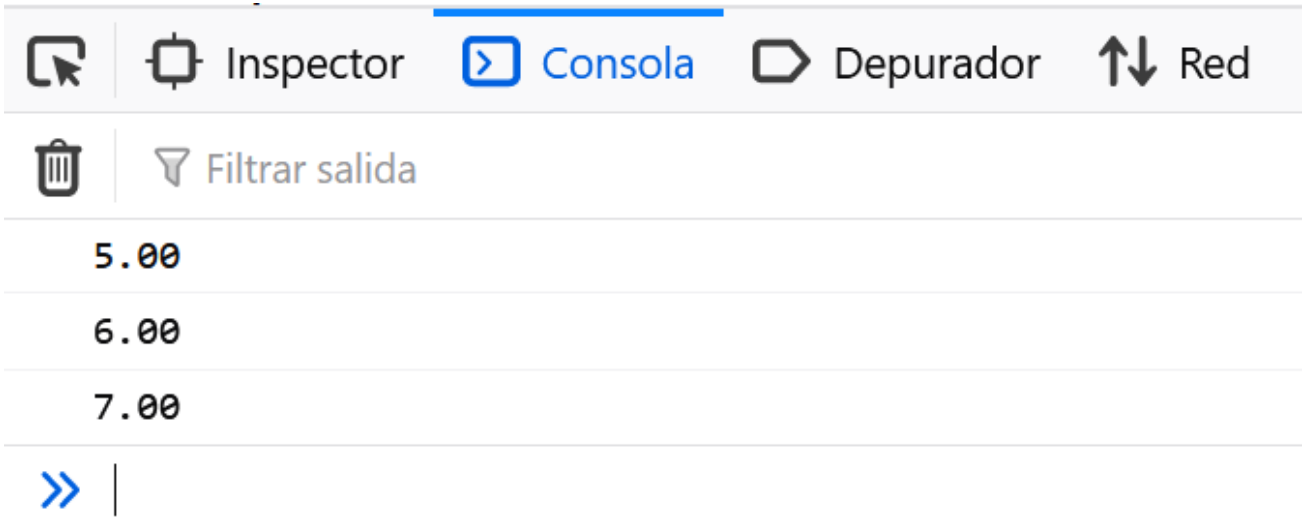
Javascript se ejecuta de forma asíncrona, lo cual quiere decir que no se puede congelar la ejecución del programa.

Como alternativa, proporciona los métodos **setTimeout** y **setInterval**

El método **setInterval** recibe dos parámetros:

- El primero es una función que se ejecutará cuando haya transcurrido el tiempo especificado.
- El segundo indicará el tiempo que hay que esperar, medido en milisegundos

```
let segs = 0;
setInterval( () => {
    console.log( (segs++).toFixed(2) );
}, 1000 )
```



El método **setTimeout** ejecuta la función que se le pase como primer parámetro una vez transcurrido el tiempo que se indique como segundo parámetro (en milisegundos).

Es decir, la función se ejecuta una única vez.

```
setTimeout( () => {
    console.log( "Hola mundo!!!" );
}, 1000 );
```

Hay ocasiones en que hemos programado una tarea con setTimeout() y queremos cancelarla antes de que transcurra el tiempo indicado.

También puede ocurrir que hayamos planificado una tarea para que se ejecute recurrentemente con setInterval() y queremos que deje de hacerlo.

Para estas situaciones disponemos de las funciones **clearTimeout()** y **clearInterval()**.

En ambas funciones necesito hacer referencia al *timeout* o al *intervalo* que quiero cancelar mediante su identificador.

El **identificador** lo devuelve la propia función setInterval() o setTimeout(), por lo que debo guardarlo en alguna variable si posteriormente querré cancelarlo.

```
let id = setInterval( () => console.log("Hola!!") );
//...
clearInterval(id);
```