Unidad de Trabajo 6: Modelo de Objetos del Documento en JavaScript

1. Introducción

El **Document Object Model (DOM)**, conocido como el Modelo de Objetos del Documento, es fundamentalmente una interfaz de programación de aplicaciones (API) proporcionada por el W3C. Este modelo es esencial para representar documentos HTML y XML (Extensible Markup Language), ofreciendo un conjunto estándar de objetos y estableciendo un modelo común sobre cómo estos objetos pueden combinarse. Además, el DOM proporciona una interfaz **estandarizada** para acceder y manipular el contenido, la estructura y el estilo de los documentos HTML y XML.

La creación del DOM ha sido una innovación significativa que ha dejado una marca indeleble en el **desarrollo de páginas web dinámicas** y aplicaciones web más complejas. Este modelo permite a los programadores web acceder y manipular páginas XHTML como si fueran documentos XML, ya que originalmente se diseñó para simplificar la manipulación de estos últimos.

El DOM se divide en tres partes o niveles principales:

- **DOM Core**, que es un modelo estándar para cualquier documento estructurado.
- **DOM XML**, que proporciona un modelo estándar para documentos XML.
- **DOM HTML**, que establece un modelo estándar para documentos HTML.

A pesar de sus orígenes centrados en documentos XML, el DOM se ha convertido en una herramienta versátil disponible para la mayoría de los lenguajes de programación, como Java, PHP y JavaScript, con diferencias mínimas en la implementación según el lenguaje utilizado.

2. Bases del DOM HTML

El DOM HTML, como **estándar**, establece las pautas para acceder, modificar, añadir o eliminar elementos HTML en un documento. Dentro del DOM, se definen los **objetos, propiedades y métodos** necesarios para interactuar con todos los elementos de un documento HTML.

Es importante tener en cuenta que, a medida que evoluciona el panorama de los navegadores, las diferencias entre ellos se han reducido considerablemente. Aunque anteriormente los navegadores basados en Mozilla hacían grandes esfuerzos para implementar todos los niveles del DOM 1 y la mayoría del Nivel 2 del W3C, y Microsoft realizaba una implementación parcial del DOM en sus navegadores, en la actualidad, la adaptación al estándar del W3C es más uniforme.

Los navegadores modernos, como Chrome, Safari y Opera, ofrecen un amplio y extensivo soporte al DOM del W3C, lo que contribuye a una experiencia más consistente y compatible para los desarrolladores web. En este contexto actual, las diferencias en la implementación del DOM entre los navegadores han disminuido, facilitando el desarrollo de aplicaciones web más coherentes y compatibles.

Niveles del DOM

Al igual que muchas otras especificaciones del W3C, la evolución del DOM no se limita a una sola versión, sino que sigue un camino de desarrollo continuo. El DOM se encuentra en constante evolución, y las fechas propuestas por el W3C para las diversas versiones raramente coinciden con las versiones de los navegadores. Por lo tanto, es común que muchos navegadores incluyan solo algunos detalles de las versiones más recientes del W3C.

La primera especificación del DOM, conocida como nivel 1, se lanzó después de Netscape 4 e Internet Explorer 4. Esta especificación cubre la parte de HTML denominada DOM de nivel 0, aunque no existe un estándar publicado con ese nombre. Esta especificación es esencialmente el modelo de objetos implementado en Navigator 3 y en parte de Internet Explorer 3, incluyendo el objeto imagen. Una omisión notable en este modelo de nivel 1 fue la falta de una especificación del modelo de eventos.

La evolución continúa con el DOM de nivel 2, que se construye sobre los desarrollos del nivel 1. Se han introducido nuevas secciones, estilos y formas de inspección de la jerarquía del documento, las cuales se han publicado como módulos separados. Algunos módulos del nivel 3 del DOM han alcanzado el estado de "Recomendación". Aunque Internet Explorer sigue sin implementar la gran mayoría de opciones de los módulos, otros navegadores sí implementan algunos de ellos, incluso aquellos que están en estado de "Borrador". Esta diversidad en la implementación destaca la continua adaptación y la no siempre uniforme adopción de las últimas especificaciones del DOM por parte de los navegadores actuales.

3. Objetos del DOM. Propiedades y métodos

En unidades anteriores, has trabajado con varios de los objetos que se presentan en esta lista. A continuación, se proporciona la referencia completa de objetos que puedes encontrar en el Modelo de Objetos del Documento para HTML.

Es importante recordar la sintaxis para acceder a las propiedades o métodos de los objetos dentro de nuestro documento:

```
document.getElementById(objetoID).propiedad

document.getElementById(objetoID).metodo([parametros])
```

Recuerda que en el caso de los métodos, los parámetros son opcionales. A continuación, se muestra un ejemplo completo tal y como has venido trabajando a lo largo de las unidades anteriores:

```
// Supongamos que tienes un elemento en tu HTML con el id "miElemento"
var miElemento = document.getElementById("miElemento");
// Acceder a la propiedad "innerHTML" del elemento
var contenido = miElemento.innerHTML;
console.log(contenido);
// Llamar al método "addEventListener" para agregar un evento al elemento
miElemento.addEventListener("click", function() {
  alert("¡Hiciste clic en el elemento!");
});
```

El DOM HTML proporciona una amplia variedad de objetos que representan diferentes aspectos de un documento HTML. Aquí tienes una lista de algunos de los objetos más comunes y utilizados en el DOM HTML:

- **document:** Representa todo el documento HTML y proporciona métodos para acceder a elementos dentro de él.
- **element:** Representa un elemento HTML. Los elementos específicos tienen sus propios objetos, como HTMLDivElement para un .
- **node:** Representa un nodo en el árbol de nodos del DOM. Puede ser un elemento, atributo, texto, etc.
- **attribute:** Representa un atributo de un elemento HTML.
- **event:** Representa un evento que ocurre en un elemento, como un clic del mouse o una pulsación de tecla.
- **window:** Representa la ventana del navegador y proporciona métodos y propiedades globales.
- **location:** Representa la URL de la ventana y proporciona métodos para redirigir la página.

- **navigator:** Proporciona información sobre el navegador del usuario.
- **history:** Representa el historial de navegación del usuario.
- **console:** Proporciona métodos para imprimir mensajes en la consola del navegador.
- **form:** Representa un formulario HTML y proporciona métodos para acceder y manipular los elementos del formulario.
- **documentType**: Representa la declaración DOCTYPE del documento.
- **screen:** Proporciona información sobre la pantalla del usuario.
- **localStorage y sessionStorage:** Permiten almacenar datos en el navegador de forma persistente o de sesión, respectivamente.

Esta lista incluye sólo **algunos** de los objetos principales disponibles en el DOM HTML. Cada objeto tiene propiedades y métodos específicos que permiten interactuar y manipular la estructura y contenido de un documento HTML.

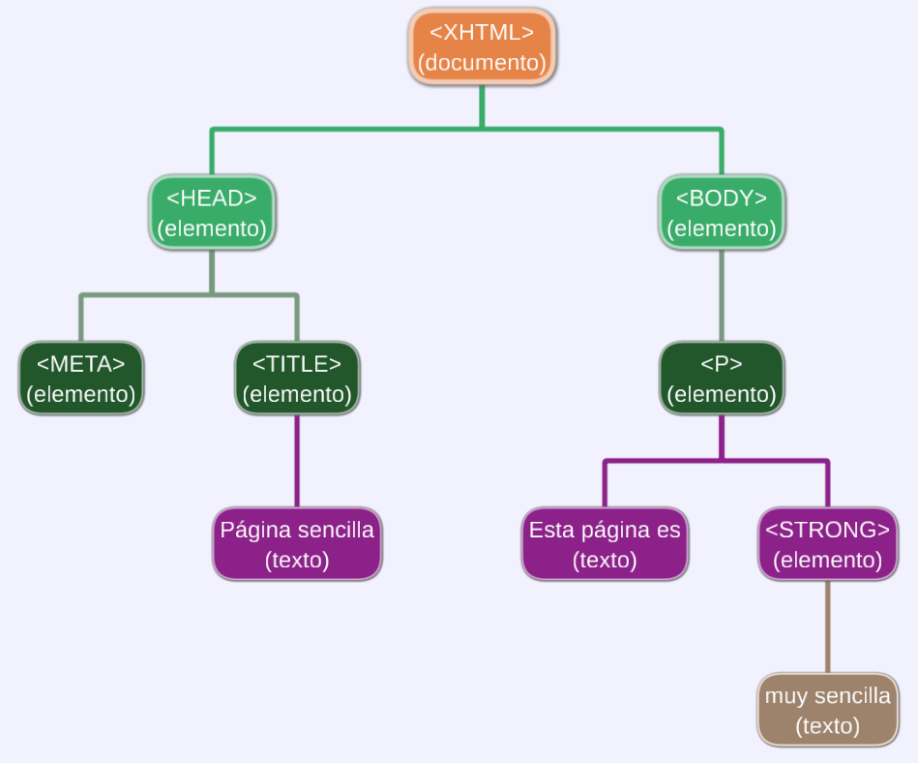
4. El árbol del DOM y tipos de nodos

En el ámbito de la programación web, una tarea común es la manipulación de páginas web, que implica acceder a su contenido, crear nuevos elementos, realizar animaciones, modificar valores, entre otras acciones. **El DOM facilita considerablemente todas estas operaciones.** Los navegadores web desempeñan un papel fundamental al transformar nuestros documentos en una estructura jerárquica de objetos, permitiéndonos acceder a su contenido de manera más organizada mediante métodos estructurados.

El DOM lleva a cabo la **transformación** de todos los documentos XHTML en un conjunto de elementos denominados **nodos**. En el HTML DOM, cada nodo es tratado como un **objeto**. Estos nodos están interconectados y representan los diversos contenidos de la página web, así como las relaciones entre ellos. Al unir estos nodos de manera jerárquica, se forma una estructura similar a un árbol, a menudo referida como árbol DOM, **"árbol de nodos"**, entre otras denominaciones. Este enfoque jerárquico facilita la manipulación y navegación coherente a través de la estructura de la página web en el entorno de programación. Veamos un ejemplo que ilustra esta conversión de una página web sencilla en su árbol de nodos:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
   <title>Página sencilla</title>
</head>
<body>
   <p>Esta página es <strong>muy sencilla</strong></p>
</body>
</html>
```

Esta página web, que tal y como puedes ver es muy simple será convertida por el navegador en el siguiente árbol de nodos:



Cada cuadro en el gráfico representa un nodo en el DOM y las líneas indican las relaciones entre los mismos. La raíz del árbol de nodos siempre es un nodo especial llamado **"document"**. A partir de este nodo, cada etiqueta XHTML se transforma en nodos de tipo **"elemento" o "texto".** Los nodos de tipo "texto" contienen el texto contenido dentro de esa etiqueta XHTML (nodo de tipo "elemento"). Esta conversión se lleva a cabo de manera jerárquica, donde el nodo inmediatamente superior se convierte en el **nodo padre**, y todos los nodos debajo de él se convierten en **nodos hijos**.

Como puedes imaginar, las páginas XHTML habituales producen árboles con **miles de nodos**. Aun así, el proceso de transformación es rápido y automático, siendo las funciones proporcionadas por DOM (que se verán más adelante) las únicas que permiten acceder a cualquier nodo de la página de forma sencilla e inmediata.

Tipos de nodos

La especificación del DOM define 12 tipos de nodos, aunque en la práctica suele bastar con emplear cuatro o cinco de ellos:

- **Document:** Este nodo es la raíz del árbol y es de donde derivan todos los demás nodos.
- **Element:** Representa cada una de las etiquetas XHTML. Es el único nodo que puede contener atributos y del cual pueden derivar otros nodos.
- **Attr:** Con este tipo de nodo representamos los atributos de las etiquetas XHTML, es decir, un nodo por cada atributo=valor.
- **Text:** Este nodo contiene el texto encerrado por una etiqueta XHTML.
- **Comment:** Representa los comentarios incluidos en la página XHTML.

Aunque existen otros tipos de nodos definidos en la especificación, como **CdataSection, DocumentFragment, DocumentType, EntityReference, Entity, Notation y ProcessingInstruction**, generalmente en el desarrollo web cotidiano, nos enfocamos en el uso de los primeros cinco mencionados. Estos nodos son esenciales para manipular y acceder a la estructura y contenido de un documento HTML de manera efectiva.

5. Acceso a los nodos

Una vez que el árbol de nodos del DOM se ha generado automáticamente, podemos empezar a utilizar sus funciones para acceder a cualquier nodo del árbol. Acceder a un nodo del árbol es equiparable a acceder a una sección específica de la página de nuestro documento. Por lo tanto, una vez que hemos alcanzado esa parte del documento, tenemos la capacidad de modificar valores, crear y agregar nuevos elementos, reorganizarlos, entre otras acciones.

Existen dos métodos principales para acceder a un nodo específico (elemento XHTML): podemos seguir la jerarquía de nodos desde el nodo raíz, accediendo a los nodos hijos sucesivamente hasta llegar al elemento deseado, o podemos utilizar un método de acceso directo. El método de acceso directo, que es el más comúnmente empleado, utiliza funciones del DOM que nos permiten llegar directamente a un elemento sin tener que recorrer nodo por nodo.

Es esencial destacar que, para que podamos acceder a todos los nodos de un árbol, este debe estar completamente construido. En otras palabras, solo podemos acceder a cualquier elemento de la página XHTML una vez que la página ha sido cargada por completo.

A continuación vamos a ver una tabla que resume los principales métodos de acceso a los nodos del DOM:

Principales métodos de acceso a los nodos del DOM	Descripción
getElementsByName()	**Descripción:** Este método devuelve una colección de elementos que tienen un atributo name con el valor especificado. **Uso común:** Se utiliza cuando se desea seleccionar elementos por su atributo name en lugar de id o tag.
getElementsByTagName()	**Descripción:** Este método devuelve una colección de elementos con el nombre de la etiqueta especificada. **Uso común:** Útil para seleccionar múltiples elementos que comparten la misma etiqueta, como todos los elementos <p>.
getElementById()	**Descripción:** Este método devuelve el elemento que tiene el atributo id con el valor especificado. **Uso común:** Se utiliza cuando se desea seleccionar un elemento único por su atributo id, que debe ser único en toda la página.

querySelector()	**Descripción:** Este método devuelve el primer elemento que coincide con un selector CSS especificado. **Uso común:** Proporciona flexibilidad al seleccionar elementos utilizando selectores CSS, permitiendo una sintaxis más avanzada.
querySelectorAll()	**Descripción:** Este método devuelve todos los elementos que coinciden con un selector CSS especificado en un NodeList. **Uso común:** Similar a querySelector(), pero devuelve todos los elementos que coinciden en lugar de solo el primero. Permite operaciones en lotes sobre la colección resultante.

getElementsByName():

Este método devuelve una **colección** que contiene todos los elementos de la página XHTML cuyo atributo **name** coincida con el indicado como parámetro. Cabe destacar que la colección **no es un array**, pero se puede recorrer y referenciar similar a un array, aunque no admite los métodos propios de un array tales como, por ejemplo, push() o pop(). A continuación te muestro un ejemplo de su uso:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejemplo getElementsByName</title>
</head>
<body>
  <!-- Tres elementos con el atributo name="apellidos" -->
  <input type="text" name="apellidos" value="Apellido1">
  <input type="text" name="apellidos" value="Apellido2">
  <input type="text" name="apellidos" value="Apellido3">
```

```
  <script>
    // Obtener una colección de elementos con el atributo name="apellidos"
    var elementos = document.getElementsByName("apellidos");
    // Acceder al primer elemento de la colección
    var primerElemento = elementos[0];
    console.log("Primer Elemento:", primerElemento.value);
    // Acceder al segundo elemento de la colección
    var segundoElemento = elementos[1];
    console.log("Segundo Elemento:", segundoElemento.value);
    // Recorrer la colección mediante un bucle
    for (var j = 0; j < elementos.length; j++) {
      var elemento = elementos[j];
      console.log("Elemento", j + 1 + ":", elemento.value);
      // Realizar operaciones con cada elemento
    }
  </script>
</body>
</html>
```

getElementsByTagName():

Esta función es similar a la anterior y devuelve una colección de elementos cuya etiqueta XHTML coincide con la que se pasa como parámetro.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
```

```
  <title>Ejemplo getElementsByTagName</title>
</head>
<body>
  <!-- Cuatro elementos input en el documento -->
  <input type="text" value="Input 1">
  <input type="text" value="Input 2">
  <input type="text" value="Input 3">
  <input type="text" value="Input 4">

  <script>
    // Obtener una colección de elementos input en el documento
    var elementos = document.getElementsByTagName("input");

    // Acceder al cuarto elemento de la colección
    var cuartoElemento = elementos[3];
    console.log("Cuarto Elemento:", cuartoElemento.value);

    // Recorrer la colección mediante un bucle
    for (var i = 0; i < elementos.length; i++) {
      var elemento = elementos[i];
      console.log("Elemento", i + 1 + ":", elemento.value);
      // Realizar operaciones con cada elemento
    }
  </script>
</body>
</html>
```

getElementsById():

Esta función es la más utilizada, ya que nos permite acceder directamente al elemento por el ID. Tal y como hemos venido haciendo durante todo el curso, entre paréntesis se escribe la cadena de texto con el ID. **Es muy importante que el ID sea único** para cada elemento de una misma página. Así, la función nos devolverá únicamente el nodo buscado.

querySelector()

La función querySelector() acepta como parámetro un selector que identifica el elemento (o elementos) a seleccionar. En el caso de esta función, únicamente es devuelto el primer elemento que cumple la condición. Si no existe el elemento, el valor retornado es null.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejemplo querySelector()</title>
</head>
<body>
  <!-- Lista de elementos con diferentes clases y etiquetas -->
  <div class="seccion" id="seccion1">Sección 1</div>
  <div class="seccion" id="seccion2">Sección 2</div>
  <p class="parrafo" id="parrafo1">Este es un párrafo.</p>
  <p class="parrafo" id="parrafo2">Otro párrafo aquí.</p>

  <script>
        // Utilizar querySelector para seleccionar el primer elemento con la clase "seccion"
    var primeraSeccion = document.querySelector(".seccion");
    console.log("Contenido de la primera sección:", primeraSeccion.textContent);
```

```
    // Utilizar querySelector para seleccionar el párrafo con el ID "parrafo2"
    var parrafoDos = document.querySelector("#parrafo2");
    console.log("Contenido del segundo párrafo:", parrafoDos.textContent);
  </script>
</body>
</html>
```

querySelectorAll()

La función querySelectorAll() acepta como parámetro un selector que identifica el elemento (o elementos) a seleccionar. Esta función devuelve un objeto de tipo NodeList con los elementos que coincidan con el selector.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejemplo querySelectorAll()</title>
</head>
<body>
  <!-- Lista de elementos con diferentes clases y etiquetas -->
  <div class="seccion" id="seccion1">Sección 1</div>
  <div class="seccion" id="seccion2">Sección 2</div>
  <p class="parrafo" id="parrafo1">Este es un párrafo.</p>
  <p class="parrafo" id="parrafo2">Otro párrafo aquí.</p>
```

```
  <script>
      // Utilizar querySelectorAll para seleccionar todos los elementos con la clase "seccion"
    var secciones = document.querySelectorAll(".seccion");

    // Acceder al NodeList resultante e imprimir el contenido de cada elemento
    secciones.forEach(function(seccion, indice) {
                console.log("Contenido de la sección " + (indice + 1) + ":", seccion.textContent);
    });

    // Utilizar querySelectorAll para seleccionar todos los párrafos
    var parrafos = document.querySelectorAll("p");

    // Acceder al NodeList resultante e imprimir el contenido de cada párrafo
    parrafos.forEach(function(parrafo, indice) {
                console.log("Contenido del párrafo " + (indice + 1) + ":", parrafo.textContent);
    });
  </script>
</body>
</html>
```

5.1. Acceso a los nodos de tipo atributo

Ahora que ya sabes cómo acceder a los nodos en un documento, vamos a ver a continuación cómo tener acceso concreto a los nodos de tipo **atributo**. Para referenciar un atributo, como por ejemplo el atributo type="text" del campo "apellidos", emplearemos la colección **attributes**. Dependiendo del navegador, esta colección se podrá cubrir de diferentes maneras y podrán existir muchos pares en la colección, tantos como atributos tenga el elemento. Para buscar el par correcto emplearemos la propiedad **nodeName** y para acceder a su valor usaremos **nodeValue**. Por ejemplo:

```
<input type="text" id="apellidos" name="apellidos" />
<script>
  let apellidos = document.getElementById("apellidos");
  for(let i = 0; i < apellidos.attributes.length; i++ ){
     let atributo = apellidos.attributes[i];
     console.log(atributo.nodeName, atributo.nodeValue);
  }
</script>
```

Vemos que se accede tanto al nombre como al valor de cada atributo iterando sobre dicha colección. Igualmente, se puede modificar los valores de los atributos manualmente mediante **asignación**. Por ejemplo:

```
document.getElementById("apellidos").attributes[1].nodeValue="password";
// En este caso hemos modificado el type del campo apellidos y lo hemos puesto de tipo “password”.
//También se podría realizar como:
document.getElementById("apellidos").attributes["type"].nodeValue="password";
//Y como:
document.getElementById("apellidos").type="password";
```

Asimismo, el método **setAttribute()** permite crear o modificar los atributos de cada elemento del DOM. Por ejemplo, en el caso del elemento anterior, en el que hemos modificado el type por "password" podemos escribir la siguiente instrucción:

```
document.getElementById("apellidos").setAttribute('type','text');
```

Y para **crear** un nuevo atributo para ese elemento, por ejemplo:

```
document.getElementById("apellidos").setAttribute('value','Cid Blanco');
```

Igualmente, existen los métodos getAttribute() y removeAttribute() para inspeccionar el valor de un determinado atributo o eliminarlo, respectivamente:

```
document.getElementById("apellidos").getAttribute('type');

document.getElementById("apellidos").removeAttribute('type');
```

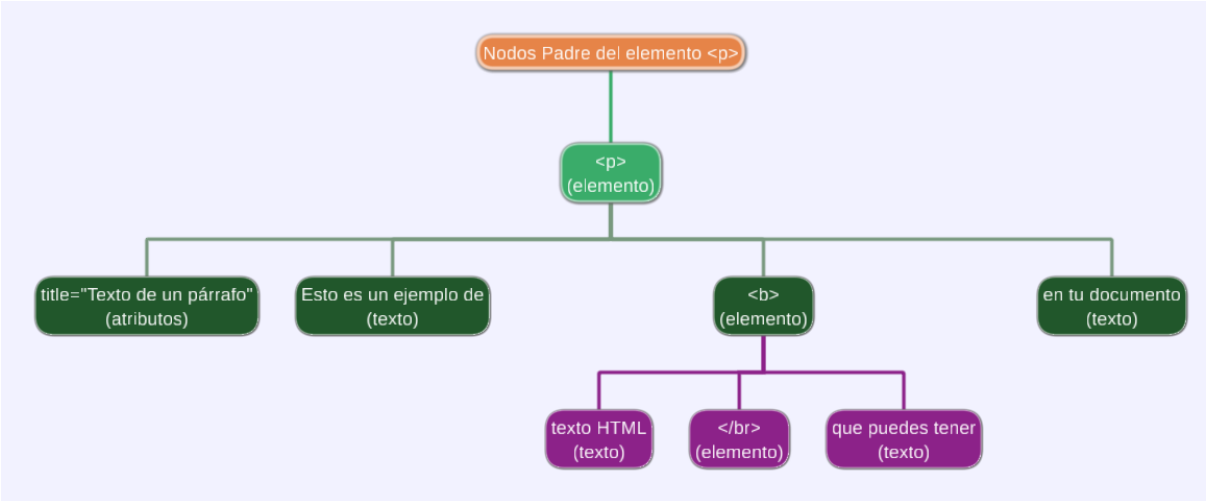
5.2. Acceso a los nodos de tipo texto

Para ver cómo acceder a la **información textual** de un nodo, nos basaremos en el siguiente ejemplo:

```
<p title="Texto de un párrafo">Esto es un ejemplo de <b>texto HTML<br />

que puedes tener</b> en tu documento.</p>
```

Ese elemento HTML tendrá el siguiente árbol del DOM asociado:



Para poder referenciar el fragmento **"texto HTML"** del nodo P, lo que haremos será utilizar la colección **childNodes**. Con la colección childNodes accederemos a los

nodos hijo de un elemento, ya sean **de tipo elemento o texto (fíjate que los nodos de tipo atributo no son accesibles con este método)**. Y el código de JavaScript para mostrar una alerta, con el contenido "texto HTML", sería:

```
window.alert(document.getElementsByTagName("p")[0].childNodes[1].childNodes[0].nodeValue);
```

Tal y como puedes ver, **childNodes[1]** selecciona el segundo hijo de <p> que sería el elemento <b> (el primer hijo es un nodo de tipo Texto "Esto es un..."). A continuación, childNodes[0] selecciona el primer hijo del elemento <b> que es el nodo de texto "texto HTML". En lugar de **childNodes[0]** también podríamos haber utilizado **firstChild**:

```
window.alert(document.getElementsByTagName("p")[0].childNodes[1].firstChild.nodeValue);
```

Cabe destacar que el tamaño máximo de lo que se puede almacenar en un nodo de texto depende del navegador, por lo que muchas veces, si el texto es muy largo, tendremos que consultar varios nodos para ver todo el contenido. Asimismo, en el DOM de HTML para acceder al valor de un nodo de texto, o modificarlo, es muy común ver la propiedad **innerHTML**. No obstante, esta propiedad aunque soportada a nivel actual por la mayoría de navegadores debe manejarse con especial cuidado y asegurar su compatibilidad. En casos de incompatibilidad, los métodos del DOM descritos en esta sección serán los más recomendables. Para modificar el contenido del nodo, modificaremos la propiedad **nodeValue** y le asignaremos otro valor, tal y como hemos visto hasta ahora. Igualmente, mediante este método de asignación se pueden mover nodos de sitio.

6. Creación y borrado de elementos mediante los métodos del DOM

La creación y eliminación de nodos constituyeron uno de los propósitos originales del DOM. Es posible generar elementos y luego incorporarlos en el DOM, reflejándose la actualización automáticamente en el navegador. También es viable trasladar nodos existentes, como ya se indicó anteriormente simplemente insertándolos en cualquier otra ubicación dentro del árbol del DOM.

Es importante tener en cuenta que al crear nodos de elementos, **el nombre del elemento debe estar en minúsculas**. Aunque en HTML esto no suponga un problema, en XHTML es sensible a mayúsculas y minúsculas, por lo que es necesario que esté escrito en minúsculas.

Para llevar a cabo estas operaciones, utilizaremos los métodos **createElement()**, **createTextNode()** y **appendChild()**. Estos nos permitirán crear un nuevo elemento, generar un nodo de texto y agregar un nuevo nodo hijo, respectivamente.

Supongamos, por ejemplo:

```
<p title="Texto de un párrafo" id="parrafito">Esto es un ejemplo de <b>texto HTML<br/> que puedes tener</b> en tu documento.</p>
```

Para crear un nuevo párrafo puedes proceder como:

```
let nuevoParrafo = document.createElement('p');

let nuevoTexto = document.createTextNode('Contenido añadido al párrafo.');

nuevoParrafo.appendChild(nuevoTexto);

document.getElementById('parrafito').appendChild(nuevoParrafo);
```

Que nos dará como resultado:

```
<p id="parrafito" title="Texto de un párrafo"> Esto es un ejemplo de <b>texto HTML<br>que puedes tener </b> en tu documento.<p>Contenido añadido al párrafo.</p> </p>
```

Asimismo, podríamos haber utilizado **insertBefore()** en lugar de **appendChild()** o, incluso, añadir manualmente el nuevo elemento al final de la colección de nodos childNodes. Si usamos **replaceChild()**, se podrían sobrescribir nodos ya existentes. También es posible copiar un nodo usando **cloneNode(true)**. Ésto devolverá una copia del nodo, pero **no lo añade automáticamente a la colección childNodes.**

Ejemplo:

```
<!DOCTYPE html>

<html lang="">

<head>

  <meta charset="utf-8">

</head>
```

```
<body>
    <p title="Texto de un párrafo" id="parrafito">Esto es un ejemplo de <b>texto HTML<br/> que puedes tener</b> en tu documento.</p>
  <script>
  //Creamos tres elementos nuevos: p, b, br
  let elementoP = document.createElement('p');
  let elementoB = document.createElement('b');
  let elementoBR = document.createElement('br');
  //Le asignamos un nuevo atributo title al elementoP que hemos creado.
  elementoP.setAttribute('title','Parrafo creado desde JavaScript');
  //Preparamos los nodos de texto
  let texto1 = document.createTextNode('Con JavaScript se ');
  let texto2 = document.createTextNode('pueden realizar ');
  let texto3 = document.createTextNode('un monton');
  let texto4 = document.createTextNode(' de cosas sobre el documento.');
  //Añadimos al elemento B los nodos de texto2, elemento BR y texto3.
  elementoB.appendChild(texto2);
  elementoB.appendChild(elementoBR);
  elementoB.appendChild(texto3);
  //Añadimos al elemento P los nodos de texto1, elemento B y texto 4.
  elementoP.appendChild(texto1);
  elementoP.appendChild(elementoB);
  elementoP.appendChild(texto4);
  //insertamos el nuevo párrafo como un nuevo hijo de nuestro parrafo
  document.getElementById('parrafito').appendChild(elementoP);
</script>
```

```
</body>

</html>
```

7. Propiedades y métodos de los objetos nodos. DOM nivel 2

Propiedad	Descripción
nodeName	Varía según el tipo de nodo.
nodeValue	Varía según el tipo de nodo.
nodeType	Constante que representa cada tipo.
parentNode	Referencia al siguiente contenedor más externo.
childNodes	Todos los nodos hijos en orden.
firstChild	Referencia al primer nodo hijo.
lastChild	Referencia al último nodo hijo.
previousSibling	Referencia al hermano anterior.
nextSibling	Referencia al hermano siguiente.
attributes	Colección de atributos de todos los nodos.
ownerDocument	Contiene el objeto document.
namespaceURI	URI a la definición de namespace
prefix	Prefijo del namespace.
localName	Aplicable a los nodos afectados en el namespace.

Método	Descripción
appendChild(newchild)	Añade un hijo al final del nodo actual.
cloneNode(deep)	Realiza una copia del nodo actual (opcionalmente con todos sus hijos).
hasChildNodes()	Determina si el nodo actual tiene o no hijos (valor boolean).
insertBefore(new, ref)	Inserta un nuevo hijo antes de otro hijo.
removeChild(old)	Borra un hijo.
replaceChild(new, old)	Reemplaza un hijo viejo con el nuevo viejo.
isSupported(feature,version)	Determina cuando el nodo soporta una característica especial.

8. Desarrollo de aplicaciones Cross-Browser

En el ámbito del desarrollo de aplicaciones Cross-Browser, donde la diversidad de navegadores y plataformas es más evidente que nunca, garantizar la coherencia y eficiencia de las aplicaciones en todos los entornos se vuelve esencial. A tal respecto, existen enfoques tradicionales como la escritura manual de código adaptado a las peculiaridades de cada navegador, lo cual puede resultar laborioso y propenso a errores. Asimismo, el uso de bibliotecas y frameworks como Bootstrap o Foundation ha sido común para proporcionar una base estilizada y funcional, aunque a veces no abordan completamente los desafíos de compatibilidad.

Sin embargo, en este curso, nos centraremos en jQuery como una herramienta integral para facilitar el desarrollo Cross-Browser. Con una interfaz de programación amigable, jQuery simplifica la complejidad asociada con las diferencias entre navegadores. Asimismo, facilita tareas comunes como la manipulación del DOM, el manejo de eventos y las llamadas AJAX, independientemente del navegador utilizado.

La versatilidad de jQuery se evidencia al abordar desafíos específicos de compatibilidad. Por ejemplo, la normalización automática de funciones relacionadas con el Modelo de Objetos del Documento (DOM) asegura un comportamiento consistente en diferentes entornos.

En un entorno tecnológico en constante evolución, jQuery se beneficia de actualizaciones continuas proporcionadas por su activa comunidad. Las versiones más recientes ofrecen soluciones para desafíos emergentes y optimizan el rendimiento, proporcionando a los desarrolladores una base sólida para construir aplicaciones modernas y compatibles.

Aunque existen otras herramientas y enfoques en el desarrollo web, jQuery mantiene su popularidad debido a su accesibilidad y a la amplia variedad de complementos disponibles. Su capacidad para simplificar el desarrollo Cross-Browser optimiza el proceso y permite a los desarrolladores concentrarse en la lógica de la aplicación en lugar de las particularidades de cada navegador.

En resumen, el desarrollo de aplicaciones Cross-Browser continúa siendo fundamental en el panorama tecnológico actual, y en este curso, aprovecharemos jQuery como una herramienta clave para facilitar esta tarea. Su enfoque simplificado y su comunidad activa lo posicionan como una valiosa herramienta para garantizar la compatibilidad y la consistencia en una amplia gama de entornos de navegadores.

9. Bibliografía y Webgrafía

[1] A. Garro, "JavaScript", *Arkaitzgarro.com*, 01-ago-2014. [En línea]. Disponible en: https://www.arkaitzgarro.com/javascript/index.html [Consultado: 12-nov-2023].

[2] "Introducción a JavaScript", Uniwebsidad.com. [En línea]. Disponible en: https://uniwebsidad.com/libros/javascript [Consultado: 14-nov-2023].