Modelado de páginas web

JavaScript es un lenguaje que permite crear lo que se llama DHTML. Este término se refiere a las páginas web que modifican ellas mismas sus propios contenidos sin cargar una nueva página. Es este cambio en la estructura de una página Web lo que estudiaremos en esta sección.

Manipular el código HTML (Parte 1/2)

En este primer capítulo de la manipulación de código HTML, vamos a ver el concepto de DOM. Vamos a estudiar la primera forma de navegar entre los nodos que componen una página HTML y vamos a editar el contenido de una página al agregar, modificar y eliminar nodos.

Verás rápidamente que es bastante fácil de manipular el contenido de una página web y que esto se convierte en un procedimiento eventualmente necesario.

Modelo de Objetos de Documento

El Modelo de Objetos de Documento (DOM) es una interfaz de programación para los documentos con formato XML y HTML.

Una interfaz de programación, también conocida como una API (Interfaz de programación de aplicaciones para), es un un conjunto de herramientas para la comunicación entre varios programas o, en este caso, lenguajes diferentes. La API se verá a menudo, sin importar el lenguaje de programación que se aprendendiendo.

DOM es una API que se utiliza con documentos XML y HTML, lo que nos permitirá, a través de Javascript, acceder a documentos en XML y / o HTML. Es a través del DOM que podemos cambiar los elementos HTML (pantalla u ocultar un div por ejemplo), añadir, mover o incluso eliminar.

Pequeña nota de vocabulario: en un curso de HTML, hablamos de etiquetas HTML (en realidad par variable, son una etiqueta de apertura y otra de cierre). Aquí en Javascript, hablamos de elemento HTML, por la sencilla razón de que cada par de etiquetas (apertura y cierre) es visto como un objeto. Por conveniencia, y para no confundir, por lo tanto habla de elemento HTML.

Un poco de historia

Originalmente, cuando Javascript se encontraba integrado en los primeros navegadores (Internet Explorer y Netscape Navigator), DOM no estaba unificado, es decir que los navegadores tenían un diferente DOM. Y así, para acceder a un elemento HTML, como difieren de un navegador a otro, se obligaba a los desarrolladores web a codificar de manera diferente dependiendo del navegador. En resumen, era caótico.

W3C ha puesto todo en orden, y ha publicado una nueva especificación que llamamos "DOM-1" (nivel de DOM 1). Esta nueva especificación define claramente lo que es DOM, y sobre todo cómo un documento HTML o XML se esquematiza. Desde entonces, un documento HTML o XML se representa como un árbol, o más bien jerárquicamente. Así el elemento <html> contiene dos elementos hijos <head> y <body>, que a su vez contienen otros elementos secundarios.

Cuando fue publicada la especificación DOM-2, la novedad de esta versión era la introducción del método `getElementById()` para recuperar un archivo HTML o XML y conocer su id.

El objeto window

De hecho, antes de hablar sobre el documento, es decir, de la página web, vamos a hablar sobre el objeto `window`. El objeto `window` se llama a un objeto global que representa la ventana del navegador. Javascript se ejecuta a partir de este objeto.

Si tomamos nuestro "¡Hola Mundo!” ya visto al principio, tenemos:

Código: HTML

```
<! DOCTYPE html>

<head>
<title> Hola Mundo </ title>
</ head>

<body>
        <script>
        alert ('¡Hola mundo!');
        </ script>
</body>
</ html>
```

Al contrario de lo que se ha dicho en este curso, `alert()` no es realmente una función. Se trata en realidad de un método perteneciente al objeto `window`. Pero el objeto `window` se dice que es implícito, es decir que en general no hay necesidad de especificarlo, estas dos instrucciones producen el mismo efecto, es decir, abrir un cuadro de diálogo:

Código: JavaScript

```
window.alert ('¡Hola mundo!');
alert ('¡Hola mundo!');
```

Dado que no es necesario especificar el objeto `window`, por lo general no se indica salvo que sea necesario, por ejemplo si manipulamos marcos o *frames* (veremos más adelante de que se trata).

No hagas una generalización apresurada: si `alert()` es un método del objeto `window`, todas las funciones no son necesariamente parte de `window`. Así funciones como `isNaN()`, `parseInt()` o `parseFloat()` no dependen de un objeto. Estas son las funciones globales. Sin embargo, son extremadamente raras. Algunas funciones citadas en este apartado representan casi la mitad de las funciones conocidas como "globales", lo que muestra claramente que no son muy numerosas.

Del mismo modo, cuando se declara una variable dentro del contexto general de su *script*, la variable se convierte de hecho en un objeto de la propiedad `window`. Para mostrarlo fácilmente, ver este ejemplo:

Código: JavaScript

```
var texto= "variable global ';
(Function () {// Usamos un IEF para "aislar" el código
    var texto = 'variable local';
alert (texto) // Obviamente, la variable local tiene prioridad
alert (window.texto) // Pero todavía es posible acceder a la variable global
gracias al objeto "window"
}) ();
```

Si intentas ejecutar este ejemplo a través de la página web jsfiddle.net entonces puede ser que consigas un resultado equivocado. Se puede decir que ese sitio no permite la ejecución de todo tipo de códigos, especialmente cuando se relacionan con `window`.

Un último aspecto importante que debes recordar: cualquier variable no declarada (es decir, nunca se utiliza la palabra clave `var`) inmediatamente se convierte en propiedad del objeto `window`, y con independencia de donde se utilice esta variable. Consideremos un ejemplo simple:

Código: JavaScript

```
(Function () {/ / Usamos IEF para "aislar" el código
    texto = 'Variable accesible' // Esta variable no fue todavía
    declarada, se le asigna un valor
}) ();
alert (texto) / / Muestra: "Variable accesible'
```

Nuestra variable se utilizó por primera vez en un IEF y por lo tanto, hemos accedido desde el espacio global. Entonces ¿por qué opera de esa manera? Simplemente JavaScript trata de resolver el problema que se le planteó: se le pide que asigne un valor a la variable `text`, así que busca esta variable, pero no la encuentra, la única solución para resolver el problema que se le ha planteado es utilizar el objeto `window`. Esto significa:

Código: JavaScript

```
text = 'Variable accesible';
```

el código se interpreta de esta manera si hay alguna variable accesible con este nombre:

Código: JavaScript

```
window.text = 'Variable accesible';
```

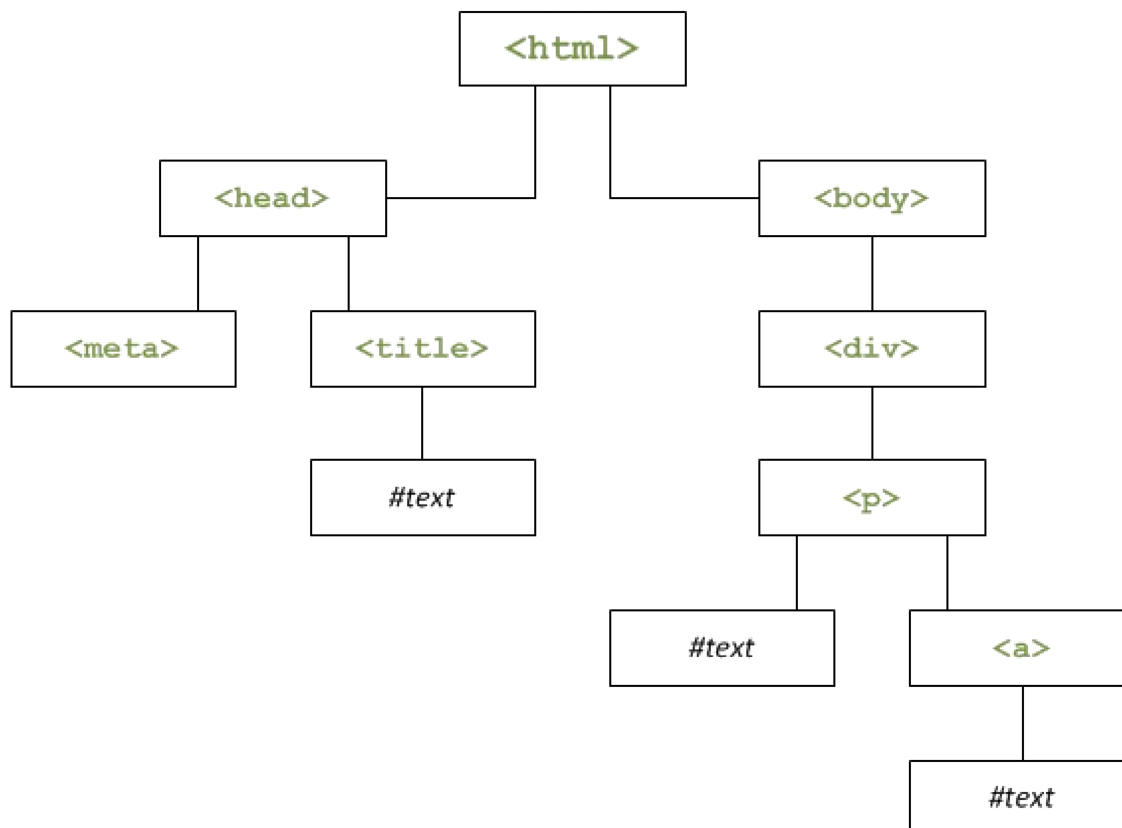
Si se muestra esta característica de JavaScript es para aconsejarte que no lo uses. Si no utilizas la palabra clave `var`, entonces pronto llegarás a una gran confusión en el código (y numerosos *bugs*). Si queremos declarar una variable en el espacio global mientras que te encuentras en otro espacio (un IEF por ejemplo), especifica explícitamente el objeto `windows`. El resto de veces, asegúrate de escribir la palabra clave var.

Document

El objeto `document` es un subobjeto de `window`, es uno de los más utilizados. Y debido a que representa la página web y más precisamente la etiqueta <html>. Es gracias a este elemento que podemos acceder y editar los elementos HTML. Vamos, en el siguiente apartado, cómo navegar por el documento.

Navegar por el documento. Estructura DOM

Como se ha indicado anteriormente, el concepto DOM considera la página web como un árbol, como una jerarquía elementos. Podemos esbozar una página web como esto:



Aquí está el código fuente de la página:

Código: HTML

```
<! DOCTYPE html>
<html>
<head>
        <meta charset="utf-8" />
        <title> Título de la página </title>
</head>
<body>
        <div>
        <p>un poco de texto y <a> un enlace </ a></ p>
        </ div>
</ body>
</ html>
```

El esquema es bastante simple: el elemento <html> contiene dos elementos llamados hijos <head> y <body>. Para estos dos hijos <html> es el elemento padre. Cada elemento se denomina nodo (node en Inglés). El elemento <head> también contiene dos hijos: <meta> y <title>. <meta> no contiene ningún hijo mientras <title> contiene uno, que se llama #texto.

Como se indica, `#text` es un elemento que contiene texto.

Es importante entender bien este concepto: el texto de una página Web es visto por el DOM como un nodo de tipo `#text`. En el diagrama anterior, el ejemplo del párrafo que contiene texto y un enlace ilustra esto:

Código: HTML

```
<p>
Parte del texto
<a>y un enlace </ a>
</p>
```

Si vamos a la línea de después de cada nodo, se ve claramente que el elemento <p> contiene dos hijos: #text que contiene "Parte del texto" y <a>, que el mismo contiene un hijo #text representando "y un enlace".

Acceso a los elementos

El acceso a HTML a través de DOM es bastante simple, pero en la actualidad sigue siendo bastante limitado. El objeto `document` tiene tres métodos principales: `getElementById()`, `getElementsByTagName()` y `getElementsByName()`.

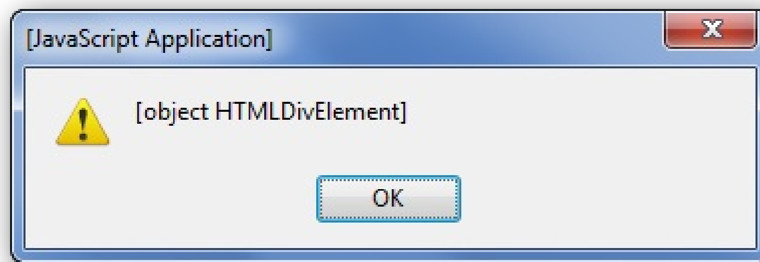
- getElementById ()

Este método proporciona acceso a un elemento conociendo su identidad, que es simplemente el atributo `id` del elemento. Funciona de la siguiente manera:

Código: HTML

```
<div id="miDivision">
	<p> un poco de texto <a> y un enlace</ a> </p>
</div>
<script>
	var div = document.getElementById ('miDiv'ision);
	alert (div);
</script>
```

Mediante la ejecución de este código, el navegador muestra lo siguiente:



Nos dice que div es un objeto de tipo `HTMLDivElement`. Claramente, este es un elemento HTML que se encuentra en un <div> este lo que muestra que la secuencia de comandos funciona correctamente.

- getElementsByTagName ()

Ten mucho cuidado con el nombre de este método hay una "s" en Elements. Esta es una fuente frecuente de errores.

Este método permite recuperar, en forma de una tabla, todos los elementos de la familia. Si, en una página, queremos recuperar todos los <div> se hace de esta manera:

Código: JavaScript

```
var divs = document.getElementsByTagName ('div');
for (var i = 0, c = divs.length; i <c, i + +) {
        alert ('Elemento N º "+ (i + 1) +": "+ divs [i]);
}
```

El método devuelve una colección de elementos (utilizados de la misma manera que una tabla). Para acceder a cada elemento, es necesario pasar por la tabla con un pequeño bucle.

Dos pequeños consejos:

1. Este método es accesible a cualquier elemento HTML y no sólo en el objeto de `document`.
2. Como parámetros de este método se puede poner una cadena que contiene un asterisco * que recupera todos los elementos HTML contenidos en el elemento de destino.

- getElementsByName ()

Este método es similar al `getElementsByTagName()` y permite recuperar sólo los elementos que tienen un atributo `name` especificado. El atributo `name` sólo se utiliza dentro de los formularios, y se amortiza a partir de la especificación HTML5 en cualquier otra forma que no

sea un formulario. Por ejemplo, lo puedes utilizar un <input> pero no para <map>.

Ten en cuenta también que este método está en desuso en XHTML, pero ahora está estandarizado para HTML5.

Acceso a los elementos mediante las últimas tecnologías

En los últimos años, JavaScript ha evolucionado para facilitar el desarrollo web. Los dos métodos bajo consideración son nuevos y no son compatibles con las versiones antiguas de los navegadores. Puede ver la tabla de compatibilidad en MDN.

Estos son dos métodos `querySelector()` y `querySelectorAll()` y tienen la particularidad que simplifican enormemente la selección de elementos en el árbol DOM a través de su modo de funcionamiento. Estos dos métodos toman un parámetro como argumento: una cadena.

Esta cadena debe ser un selector CSS como los que utilizas en tus hojas de estilo. Ejemplo:

Código: CSS

```
#menu .item span
```

Este selector CSS indica que deseas seleccionar las etiquetas de tipo <span> contenidas en las clases `.item`. ellas mismas contenidas en un elemento cuyo identificante es #menu.

El principio es muy simple pero muy eficaz. Ten en cuenta que estos dos métodos también soportan selectores CSS 3. Puedes ver la lista de la especificación W3C.

Ahora, los detalles específicos de estos dos métodos. El primero, `querySelecto()` devuelve el primer elemento encontrado correspondiente al selector CSS, mientras que `querySelectorAll()` devolverá todos los elementos (como tabla) correspondiente al selector CSS. seleccionado. Consideremos un ejemplo simple:

Código: HTML

```
<div id="menu">
    <div class="item">
        <span> Elemento 1 </ span>
        <span> Elemento 2 </ span>
    </div>
    <div class="publicidad">
        <span> Elemento 3 </ span>
        <span> Elemento 4 </ span>
    </div>
```

```
</div>
        <div id="contenido">
                <span> Introducción a contenido de la página ... </ span>
        </div>
```

Ahora prueba el selector CSS mostrado anteriormente `#menu .item span.`

En el código siguiente, se utiliza una nueva propiedad denominada `innerHTML`, la estudiaremos más adelante en este capítulo. Mientras tanto, saber que proporciona acceso a los contenidos de un elemento HTML.

Código: JavaScript

```
var query = document.querySelector('#menu .item span'),
queryAll = document.querySelectorAll('#menu .item span');

alert (query.innerHTML) // Muestra: "Artículo 1"
alert (queryAll.length) // Muestra: "2"
alert(queryAll[0].innerHTML + ' - ' + queryAll[1].innerHTML); / Muestra:
 "Artículo 1 - Articulo 2"
```

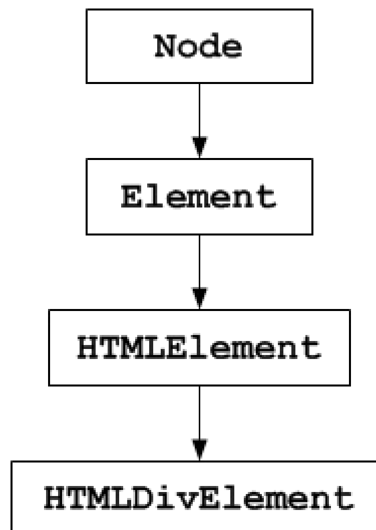
Logramos obtener los resultados deseados. Te aconsejamos que recuerdes estos dos métodos. Actualmente son proyectos útiles para ejecutar en los navegadores, y en pocos años podrían llegar a ser normales (al tiempo que las versiones antiguas de los navegadores desaparecen para siempre).

Herencia de propiedades y métodos

Javascript ve los elementos HTML como objetos, lo que significa que cada elemento HTML tiene propiedades y métodos. Sin embargo ten cuidado porque no todos tienen las mismas propiedades y métodos. Algunos son sin embargo comunes a todos los elementos HTML, porque todos los elementos HTML son del mismo tipo: el tipo `node`, que significa "nudo".

Noción de herencia

Hemos visto que un elemento <div> es un objeto `HTMLDivElement`, pero un objeto en Javascript, puede pertenecer a diferente grupos. Así nuestro <div> es un `HTMLDivElement`, que es un subobjeto de `HTMLElement` que el mismo es un subobjeto de `Element`. `Element` es finalmente un subobjeto de `Node`. Este esquema es más revelador:



El objeto `Node` proporciona una serie de propiedades y métodos que se pueden utilizar para uno de sus subobjetos. Es evidente que los subobjetos heredan las propiedades y métodos de objetos a los que pertenecen. Esto es que lo que se llama herencia.

Editar elementos HTML

Ahora que hemos visto cómo acceder a un elemento, vamos a ver cómo editarlo. Los elementos HTML a menudo se componen de atributos (atributo href de un <a> por ejemplo), y el contenido, que es de tipo `# text`. El contenido puede también ser otro elemento HTML. Como se ha indicado anteriormente, un elemento HTML es un objeto que pertenece a múltiples objetos, y por lo tanto hereda las propiedades y los métodos de su objeto padre.

- Atributos

A través del objeto `Element`

Para interactuar con los atributos, el objeto `Element` nos ofrece dos métodos, `getAttribute()` y `setAttribute()` respectivamente para recuperar y editar un atributo.

El primer parámetro es el nombre del atributo y el segundo, en el caso de `setAttribute()` únicamente, es el nuevo valor para el atributo. Pequeño ejemplo:

Código: HTML

```
<body>
        <a id="miLink" href="http://www.un_enlace.com">
        Un enlace  modificado dinámicamente </ a>
        <script>
                var link = document.getElementById('mILink');
```

```
            var href = link.getAttribute('href'); // Obtiene el atributo "href"

            alert (href);
            link.setAttribute('href', 'http://www.barzanallana.es'); // Se edita el atributo "href"
        </script>
</body>
```

Comenzamos `MiLink` recuperar el elemento, y leer su atributo `href` a través de `getAttribute()`. Luego cambiamos el valor del atributo `href` con `setAttribute()`. El enlace apunta ahora a http://www.barzanallana.es.

Atributos disponibles

De hecho, para la mayoría de los elementos comunes como <a>, es posible acceder a un atributo a través de un propiedad. Si deseas cambiar el destino de un vínculo, podemos utilizar la propiedad `href`, así:

Código: HTML

```
<body>
    <a id="miLink" href="http://www.un_enlace.com"> Un enlace dinámicamente modificado </ a>
        <script>
        var link = document.getElementById ('MiLink');
        var href = link.href;

        alert (href);

        link.href = 'http://www.barzanallana.es';
        </script>
</body>
```

Este es el enfoque que se utiliza sobre todo para formularios: para recuperar o modificar el valor de un campo, utiliza la propiedad value.

Clase

Es de suponer que para cambiar el atributo `class` de un elemento HTML, es suficiente utilizar `element.class`. No es posible porque la palabra clave `class` está reservada en JavaScript, aunque no sirve de nada. En lugar de la clase, debe utilizar `className`.

Código: HTML

```
<! DOCTYPE html>

<html>
<head>
<meta charset="utf-8"/>
<title> Título de la página </ title>

        <style type="text/css">
        .blue {
        background: blue;
        color: white;
        }
        </style>
</head>

<body>
<div id="miColorDiv">
        <p> un poco de texto <a>  y un enlace </ a></p>
</div>

        <script>
        document.getElementById ('miColorDiv').className = 'blue';
        </script>
</body>
</html>
```

En este ejemplo, se define la clase .blue para el elemento MiColorDiv, por lo que este elemento se mostrará con fondo azul y texto blanco.

Siempre en el mismo caso, el nombre `for` se ha reservado también en Javascript (para bucles). No se puede editar el atributo HTML para un <label> escribiendo `element.for`, en su lugar se debe usar `.htmlFor`.

Ten cuidado: si el elemento tiene varias clases (por ejemplo, <a class="external red u">) y recuperas la clase con `className`, no devolverá una tabla con las diferentes clases, sino la cadena "external red u", que no es realmente el comportamiento deseado. Entonces tienes que cortar la cadena con `split()` para obtener una matriz como se muestra:

Código: JavaScript

```
var misClases = document.getElementById('miLink').className;
var misClasesNew = [];
misClases = misClases.split(' ');
```

```
for (var i = 0, c = misClases.length; i < c; i++) {
        if (misClases[i]) {
        misClasesNew.push(misClases[i]);
        }
}
alert(misClasesNew);
```

Se recuperan l clases, se corta la cadena, pero es posible que varios espacios estén presentes entre cada nombre de clase, se comprueba cada elemento para ver si contiene algo (si no está vacío). Aprovechamos la oportunidad para crear una nueva tabla, `misClasesNew`, que contiene los nombres de las clases, sin "ruido".

Contenido: innerHTML

La propiedad `innerHTML` es especial y requiere poca presentación. Fue creada por Microsoft para Internet Explorer y sólo necesita ser estandarizada en HTML5. Aunque no fue estándar durante años, se ha convertido en un estándar, ya que todos los navegadores soportan esta propiedad, y no al revés como suele ser el caso.

Recuperar HTML

`innerHTML` recupera el código HTML hijo de un elemento en forma de texto. Por lo tanto, si están presentes etiquetas, las devuelva en forma de texto:

Código: HTML

```
<body>
<div id="miDiv">
        <p> un poco de texto <a> y un enlace </ a></ p>
</div>
<script>
        var div = document.getElementById ('miDiv');
        alert (div.innerHTML);
        </script>
</body>
```

Tenemos por lo tanto un cuadro de diálogo que muestra: **<p> un poco de texto <a> y un enlace </ a></ p>**

Agregar o editar HTML

Para editar o agregar contenido a HTML, simplemente se ha de hacer lo contrario, es decir, definir un nuevo contenido:

Código: JavaScript

```
document.getElementById('miDiv').innerHTML = ' <blockquote> Citación en lugar de el párrafo </ blockquote> ';
```

Si deseas agregar contenido, y no modificar el contenido ya en su sitio, sólo tienes que utilizar `+=` en lugar del operador de asignación:

Código: JavaScript

```
document.getElementById('miDiv').innerHTML += ' y <strong>una porción enfatizada</strong> ';
```

Sin embargo, una palabra de precaución: no es necesario utilizar `+=` en un bucle. En efecto, `innerHTML` ralentiza la ejecución de código cuando se opera de esta manera, es mejor para concatenar el texto en una variable y luego añadir todo a través de `innerHTML` Ejemplo:

Código: JavaScript

```
var texto =' ';
while (/ *condición* /) {
texto += 'su Texto' // Concatenamos en la variable "texto"
}
element.innerHTML = texto // Una vez completada la concatenación, se añade todo a "elemento" a través de innerHTML
```

Atención. Si alguna vez te apetece añadir una etiqueta <script> a la página a través de la propiedad `innerHTML`, ten en cuenta que esto no funciona. Sin embargo, es posible crear esta etiqueta a través del método `createElement()` que se estudia en el capítulo siguiente.

innerText y textContent

Consideremos ahora dos propiedades análogas a `InnerHTML`: `innerText` para Internet Explorer e `textContent` para otros navegadores.

innerText

Esta propiedad también se introdujo en Internet Explorer, pero a diferencia de su propiedad hermana innerHTML, nunca se ha normalizado y no está soportada por todos los navegadores. Internet Explorer (cualquier versión anterior a la 9) no admite esta propiedad y no es la versión estándar que se verá más adelante.

El funcionamiento de `innerText` es el mismo que para `innerHTML` con la excepción de que sólo se recupera el texto, no las etiquetas. Esto es útil para recuperar el contenido sin formato, un pequeño ejemplo:

Código: HTML

```
<body>
<div id="miDiv">
        <p> un poco de texto <a>y un enlace </ a> </ p>
</div>
<script>
        var div = document.getElementById ('miDiv');
        alert (div.innerText);
</script>
</body>
```

Esto mostraría en pantalla "**Un poco de texto y un enlace**", sin las etiquetas. El texto se muestra sin etiquetas HTML

textContent

La propiedad `TextContent` es la versión estandarizada de `innerText` y es reconocida por todos los navegadores excepto Internet Explorer, aunque la versión 9 también la soporta. La operación es obviamente la mism.a Ahora surge una pregunta: ¿cómo hacer un *script* que funcione para Internet Explorer y otros navegadores? Esto es lo que vamos a ver.

Probar el navegador

Es posible a través de una condición simple probar si el navegador es compatible con un determinado método o propiedad.

Código: HTML

```
<body>
        <div id="miDiv">
                <p> un poco de texto <a>y un enlace </ a> </ p>
        </div>

<script>
        var div = document.getElementById('myDiv');
        var txt = '';

        if (div.textContent) {// "textContent" ¿existe? por lo tanto,  se utiliza
                txt = div.textContent;
```

```
	} else if (div.innerText) {// "innerText" ¿existe? por lo tanto, debe estar en IE.
		txt = div.innerText +  ' [via Internet Explorer] ';

	else {} // Si no existe ninguno, se debe probablemente el hecho de que no hay ningún texto
		txt ='' // Pone una cadena vacía
	}
	alert (txt);
</script>
</body>
```

Así que sólo prueba a través de una declaración de estado si funciona. Si `textContent` no funciona, no te preocupes, tenemos `innerText`: Con Internet Explorer se mostraría: **un poco de texto y un enlace [via Internet Explorer]**.

Este código es todavía muy largo y redundante. Es posible acortarlo dramáticamente:

Código: JavaScript

```
txt = div.textContent || div.innerText || '';
```

Resumen

- DOM se utiliza para acceder a los elementos HTML en un documento para editarlos e interactuar con ellos.
- El objeto `window` es un objeto global que representa la ventana del navegador. `document`, por su parte, es un subobjeto de `window` y la página web. Es gracias a él que podremos acceder a los elementos HTML de la página Web.
- Los elementos de la página están estructurados en forma de árbol, con el elemento <html> como elemento fundador.
- Los diferentes métodos, tales como `getElementById()`, `getElementsByTagName()`, `querySelector()` o `querySelectorAll()` están disponibles para acceder a los elementos.
- Los atributos se pueden editar con `setAttribute()`. Algunos elementos tienen propiedades que permiten cambiar estos atributos.
- La propiedad `innerHTML` permite recuperar o establecer el código HTML en el interior de un elemento.
- Por su parte, `textContent` y `innerText` sólo son capaces de establecer o recuperar texto sin formato, sin ninguna etiqueta HTML.

Manipular el código HTML (Parte 2/2)

La propiedad `innerHTML` como una cualidad principal de ser fácil de usar y esta es la razón por la que generalmente es preferida por los principiantes e incluso por muchos desarrolladores experimentado. `innerHTML` ha sido durante mucho tiempo una propiedad no normalizada, pero desde HTML5 es reconocida por W3C y se puede utilizar sin problemas.

En este segundo capítulo de manejo de contenido, vamos a discutir los cambios en el documento a través de DOM. Lo hemos hecho en el primer capítulo, sobre todo con `setAttribute()`. Pero aquí será crear, eliminar y mover los elementos HTML. Este es un aspecto de Javascript, no siempre fácil de entender.

Si `innerHTML` es suficiente, ¿por qué molestarse con DOM? DOM es más potente y requerido para procesar XML.

Navegar entre nodos

Hemos visto anteriormente los métodos utilizados, `getElementById()` y `getElementsByTagName()`, para acceso a elementos HTML. Una vez que se alcanza un elemento, es posible desplazarse de forma un poco más precisa, con una variedad de métodos y propiedades que vamos a discutir aquí.

- Propiedad ParentNode

`ParentNode` proporciona acceso al elemento padre de un elemento. Mira este código:

Código: HTML

```
<blockquote>
<p id="miP"> Este es un párrafo </ p>
</ Blockquote>
```

Supongamos que se debe acceder a `miP` y por otra razón por se debe acceder al elemento <blockquote> que es el padre de `miP`. Sólo tienes que ir a miP y entonces su padre, con `parentNode`:

Código: JavaScript

```
var parrafo document.getElementById('miP');
var blockquote = parrafo.parentNode;
```

- nodeType y nodeName

nodeType y nodeName sirven respectivamente para comprobar el tipo de un nodo y el nombre de un nodo. nodeType devuelve un número, que corresponde a un tipo de nodo. La tabla siguiente muestra los tipos posibles, y su número (los tipos más habituales se muestran en negrita):

Número	Tipo de nodo
1	Nodo elemento
2	Nodo atributo
3	Nodo texto
4	Nodo 4 para pasar CDATA (en relación con XML)
5	Nodo para referencia de la entidad
6	Nodo para entidad
7	Nodo para instrucción de tratamient
8	Nodos para comentarios
9	Nodo documento
10	Nodo tipo de document
11	Nodo de fragmento de documento
12	Nodo para notación

nodeName, por su parte, devuelve el nombre del elemento en mayúsculas. Es aconsejable utilizar toLowerCase() (o toUpperCase()) para forzar un formato de letra y evitar sorpresas desagradables.

Código: JavaScript

```
var parrafo = document.getElementById('miP');
alert(parrafo.nodeType + '\n\n' +
parrafo.nodeName.toLowerCase());
```

Enumerar y navegar por nodos hijos

- firstChild y lastChild

Como su nombre indica, son, respectivamente, `lastChild` y `firstChild` permiten el acceso al primero y al último hijo de un nodo.

Código: HTML

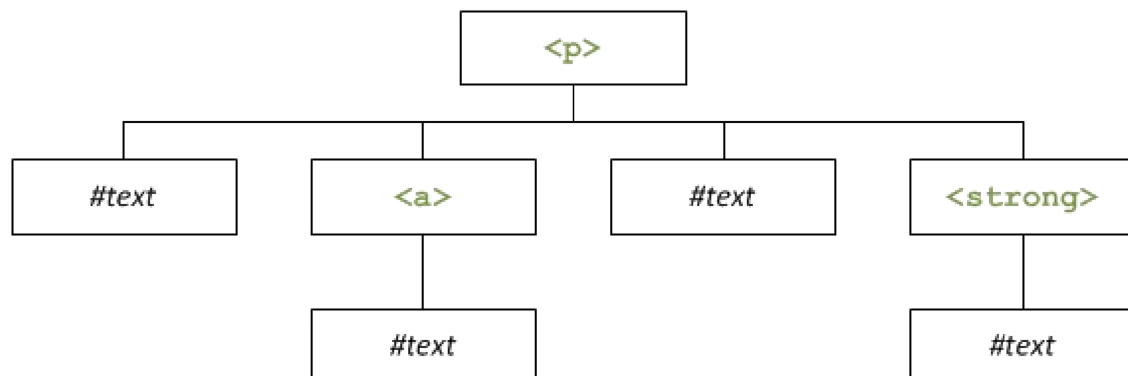
```
<! DOCTYPE html>

<html>
<head>
        <meta charset="utf-8" />
        <title> Título de la página </ title>
</head>

<body>
<div>
        <p id="miP"> Un pequeño texto, <a> un enlace </ a> <strong> y
        una parte enfatizada </ strong> </ p>
</div>

<script>
        var parrafo = document.getElementById ("miP");
        var primero = parrafo.firstChild;
        var ultimo = parrafo.lastChild;
        alert (first.nodeName.toLowerCase ());
        alert (last.nodeName.toLowerCase ());
</script>
</body>
</html>
```

Esquematizando se tiene:



El primer hijo de <p> es un nodo de texto, mientras que el último elemento hijo es un elemento <strong>.

En caso de que desees recuperar los hijos que son considerados como elementos HTML (y por lo tanto evitar los nodos #text, por ejemplo), recuerda que existen las propiedades `firstElementChild` y `lastElementChild`. Así, en el ejemplo anterior, la propiedad `firstElementChild` devolverá el elemento <a>.

Por desgracia, se trata de dos propiedades recientes, su uso se limita a las versiones más modernas de los navegadores (a partir de la versión 9 de Internet Explorer).

- nodeValue y data

Cambia el problema: debes recuperar el texto del primer hijo, y el texto contenido en el elemento, <strong>pero ¿cómo?

Debes utilizar la propiedad `nodeValue` o la propiedad data. Si recodificamos el *script* anterior, obtenemos lo siguiente:

Código: JavaScript

```
var parrafo = document.getElementById ("miP");
var primero = parrafo.firstChild;
var ultimo = parrafo.lastChild;
alert (first.nodeValue);
alert (last.firstChild.data);
```

primero contiene el primer nodo, un nodo de texto. Es suficiente aplicar la propiedad `nodeValue` (o `data`) para recuperar su contenido, no hay problema aquí. Sin embargo, hay una pequeña diferencia con el elemento <strong>: ya que las propiedades `nodeValue` y `data` sólo se aplican a los nodos de texto, primero debes acceder al nodo de texto que contiene nuestro elemento, es decir, su nodo hijo. Para ello, utilizamos `firstChild` (no `firstElementChild`), y luego se recupera el contenido con `nodeValue` o `data`.

- childNodes

Esta propiedad devuelve una tabla que contiene la lista de elementos secundarios de un elemento. El siguiente ejemplo ilustra la operación de esta propiedad, con el fin de recuperar el contenido de elementos secundarios:

Código: HTML

```
<body>
<div>
        <p id="miP">un poco de texto  <a> y un enlace</ a> </ p>
</div>
<script>
var parrafo = document.getElementById ("miP");
        var hijo = parrrafo.childNodes;
        for (var i = 0, c = hijo.length; i <c, i ++) {
                if (hijo[i].nodeType === 1) {// Esto es un elemento HTML
                        alert (hijo[i]firstChild.data.);
                        } else {  // Esto es sin duda un nodo de texto
                        alert (hijo[i].data);
                        }
        }
</script>
<body>
```

- nextSibling y previousSibling

`nextSibling` y `previousSibling` son dos atributos que permiten el acceso al nudo siguiente y anterior respectivamente

Código: HTML

```
<body>
<div>
        <p id="miP"> Un pequeño texto, <a> enlace </ a> y <strong>
        parte enfatizada </strong> </ p>
</div>
<script>
        var parrafo = document.getElementById ("miP");
        var nombre = parrafo.firstChild;
        var = first.nextSibling siguiente;
        alerta (next.firstChild.data) // Muestra "enlace"
</script>
</body>
```

En este ejemplo, obtenemos el primer hijo de miP, y el hijo utiliza `nextSibling`, lo que permite recuperar el objeto <a>. Con esto, es posible incluso recorrer los hijos de un elemento mediante un bucle `while`:

Código: HTML

```
<body>
```

```
<div>
        <p id="miP"> Un pequeño texto <a> y un enlace </ a></ p>
<div>
<script>
        var parrafo = document.getElementById ("miP");
        var = hijo parrafo.lastChild // Se toma el último hijo
        while (hijo) {
                if (hijo.nodeType === 1) {// Se trata de un elemento HTML
                alert (hijo.firstChild.data);
                } else { // Esto es sin duda un nodo de texto
                alert (hijo.data);
                }
                hijo = child.previousSibling = // Cada bucle toma al hijo
                anterior
        }
</script>
</body>
```

Para variar, el bucle se ejecuta "al revés", ya que primero recupera el último hijo y camina hacia atrás.

Al igual que `firstChild` y `lastChild`, sabemos que existen las propiedades `nextElementSibling` y `previousElementSibling` permitiendo también recuperar sólo los elementos HTML. Estos dos propiedades tienen los mismos problemas de compatibilidad que `lastElementChild` y `firstElementChild`.

Ten cuidado con los nodos vacíos

Teniendo en cuenta el código HTML siguiente, podemos suponer que el elemento <div> contiene sólo tres hijos <p>:

Código: HTML

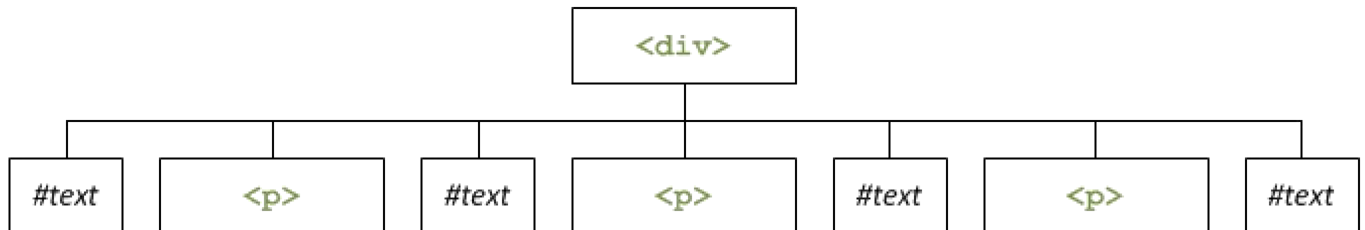
```
<div>
        Párrafo <p> 1 </ p>
        Párrafo <p> 2 </ p>
        Párrafo <p> 3 </ p>
</div>
```

Pero ten cuidado, ya que este código es radicalmente diferente de lo siguiente:

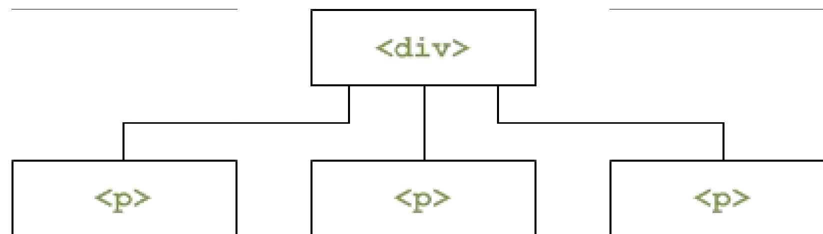
Código: HTML

```
<div> parrafo 1 </ p> parrafo 2 </ p> parrafo 3 </ p> </ div>
```

De hecho, los espacios entre los elementos en forma de saltos de línea se tratan como nodos de texto (depende del navegador). Por lo tanto, si diagramas el primer código, obtenemos lo siguiente:



Mientras que el segundo código se puede resumir así:



Afortunadamente, hay una solución a este problema. Los atributos `firstElementChild`, `lastElementChild`, `previousElementSibling` y `nextElementSibling` devuelven sólo los elementos HTML y así permiten hacer caso omiso de los nodos de texto. Se usan en exactamente la misma forma que los atributos básicos (`firstChild`, `lastChild`, etc.) Sin embargo, estos atributos no son compatibles con Internet Explorer 8 y versiones anteriores y no hay otra opción.

Creación e inserción de elementos. Añadir elementos HTML

Con DOM, añadir un elemento HTML se divide en tres etapas:

1. Creamos el elemento;
2. Asignamos atributos;
3. Se inserta en el documento, y es sólo entonces que es "añadido".

- Creación del elemento

La creación de un elemento se hace con el método `createElement()`, un subobjeto del objeto raíz, es decir, en la mayoría de los casos document:

Código: JavaScript

var NuevoEnlace = document.createElement ('a');

Se crea aquí un nuevo elemento <a>. Este elemento se crea, pero no se inserta en el documento, no es visible. Dicho esto, ya podemos trabajar en él, mediante la adición de atributos o incluso eventos.

Si trabajas en una página web, el elemento raíz siempre será `document`, salvo en el caso de los marcos. La creación de elements en archivos XML se discutirá más adelante.

-Asignación de atributos

Esto es como hemos visto anteriormente: definimos los atributos con `setAttribute()`, o ya sea directamente con las propiedades adecuadas.

Código: JavaScript

```
newLink.id = 'sdz_link';
newLink.href = 'http://www.um.es/dociencia/barzana';
newLink.title = 'Descubre el sitio Daweb !';
newLink.setAttribute('tabindex', '10');
```

Inserción de elemento

Utilizamos el método `appendChild()` para insertar el elemento. A`ppend child` significa "añadir hijo", lo que significa que necesitamos saber el elemento al que vamos a añadir el elemento creado. Así que considera el siguiente código:

Código: HTML

```
<! DOCTYPE html>
<html>
<head>
        <meta /> charset="utf-8"
        <title> título de la página </ title>
</head>

<body>
<div>
        <p id="miP"> Un poco de texto <a>y un enlace </ a> </ p>
</div>
</body>
```

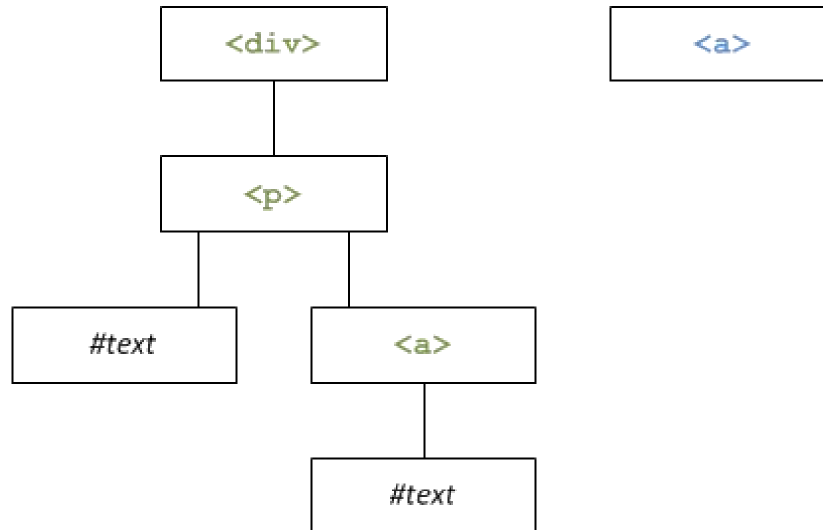
</html>

Vamos a añadir nuestro elemento <a> en el elemento <p> con identidad miP. Para ello, basta con recuperar el elemento, y agregar nuestro elemento <a> a través de `appendChild ()`:

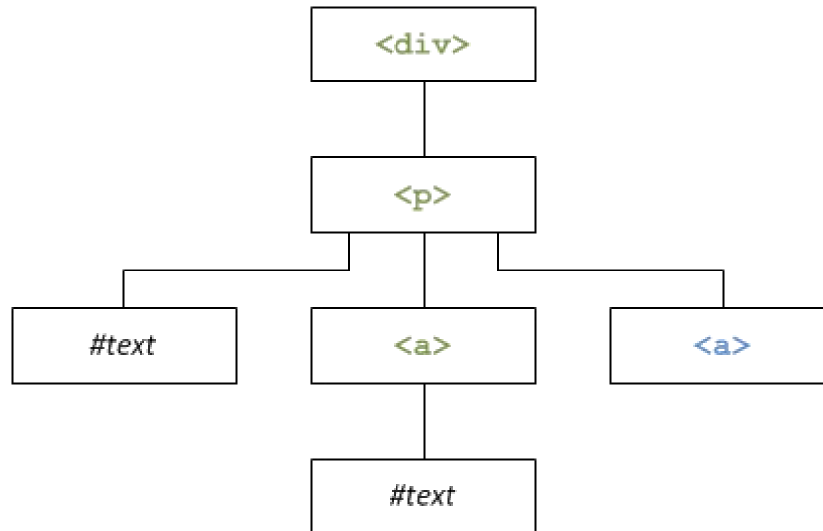
Código: JavaScript

```
document.getElementById ("miP").appendChild (Newlink)
```

Una pequeña explicación es necesaria. Antes de insertar nuestro elemento <a>, la estructura DOM del documento es la siguiente:



Vemos que el elemento <a> existe, pero no está ligado. Un poco como si estuviera libre en el documento: no se ha definido aún. El objetivo es colocarlo como un elemento hijo de miP. El método `appendChild()` va a desplazar nuestro <a> para colocarlo como el último hijo de miP:



Esto quiere decir que `appendChild()` siempre insertará el elemento como último hijo, que no es siempre muy práctico. Más adelante veremos cómo insertar un elemento antes o después de un determinado hijo.

Agregar nodos de texto

El elemento ha sido insertado, sólo hay algo que falta: el contenido textual. El método `createTextNode()` se utiliza para crear un texto en el nodo (tipo #text), así:

Código: JavaScript

```
var = newLinkText document.createTextNode ("Sitio de UMU");
newLink.appendChild (newLinkText);
```

La inserción se realiza aquí también con `appendChild()` en el element NewLink. Para ver con más claridad, resumimos el código:

Código: HTML

```
<body>
<div>
	<p id="mPi"> Un poco de texto <a>y un enlace </ a> </ p>
</div>
<script>
	newLink.id = 'sdz_link';
	newLink.href = 'http://www.um.es/docencia/barzana';
	newLink.title = 'Descubre el sitio UMU';
```

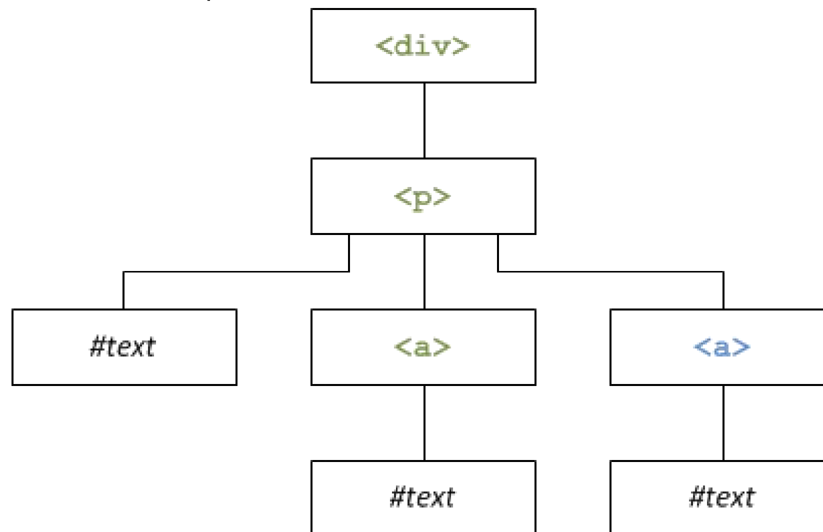
```
        newLink.setAttribute('tabindex', '10');

        document.getElementById ("miP"). appendChild (Newlink)

        var = newLinkText document.createTextNode ("UMU");

        newLink.appendChild (newLinkText);
</script>
</body>
```

Así que lo que tenemos, se esquematiza:

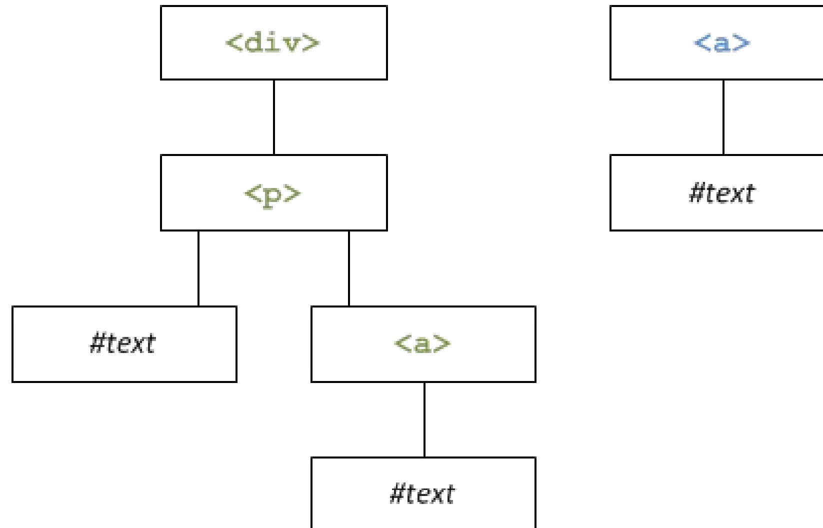


Hay algo que sabemos: el hecho de insertar a través de `appendChild()` no tiene efecto en el orden de ejecución de las instrucciones. Esto significa que puedes trabajar en los elementos HTML y nodos de texto sin estar previamente insertados en el documento. Por ejemplo, el código podría ser ordenado como sigue:

Código: JavaScript

```
var newLink = document.createElement('a');
var newLinkText = document.createTextNode("UMU");
newLink.id = 'sdz_link';
newLink.href = 'http://www.um.es/docencia/barzana';
newLink.title = 'Descubre el sitio de UMU';
newLink.setAttribute('tabindex', '10');
newLink.appendChild(newLinkText);
document.getElementById('myP').appendChild(newLink);
```

Aquí, primero se crean los dos elementos (el enlace y el nodo de texto), afectan a las variables en lugar de añadir el nodo de texto. En este punto, el elemento HTML contiene el nodo de texto, pero este elemento no está insertado en el documento:



Recomendamos organizar el código como en el ejemplo anterior, es decir, la creación de todos los elementos al principio y después las distintas operaciones de afectación. Finalmente, la inserción de los elementos unos en los otros y, finalmente, la inclusión en el documento. Al menos de la manera en que está estructurado, es claro y sobre todo más poderoso.

`appendChild()` devuelve una referencia (ver detalles más abajo) que apunta al objeto que se ha insertado. Lo se puede utilizar en caso de que no se hubiera declarado una variable intermedia en el proceso de creación de su elemento. Por ejemplo, el siguiente código no plantea problemas:

Código: JavaScript

```
var span = document.createElement ('span');
document.body.appendChild (span);

span.innerHTML = 'Algo de texto y mucho más!';
```

Sin embargo, si se elimina el paso intermedio (la primera línea) para ganar una línea de código, a continuación se ha de modificar el contenido:

Código: JavaScript

```
document.body.appendChild (document.createElement ('span'));

span.innerHTML = 'Algo de texto y mucho más!' // La variable "span"  ya
no existe
```

La solución a este problema es usar la referencia devuelta por `appendChild ()` de la siguiente manera:

Código: JavaScript

```
var span = document.body.appendChild (document.createElement ('span'));
span.innerHTML = 'Algo de texto y mucho más!' / / Aquí, todo funciona!
```

Nociones sobre referencias

En JavaScript como en otros muchos lenguajes, el contenido de las variables se "pasa por valor". Esto significa que si la variable nick1 contiene el nombre "Marisol" y se asigna este valor a una variable, el valor se copia en la nueva. Obtenemos dos variables distintas que contienen el mismo valor:

Código: JavaScript

```
var nick1 = 'Marisol';
var nick2 = nick1;
```

Si cambia el valor de nick2, el valor se mantiene sin cambios en nick1, las dos variables son distintas.

Referencias

Además del "paso por valor" JavaScript tiene un "paso por referencia". De hecho, cuando se crea una variable, su valor se almacena en el ordenador. Para encontrar este valor, está asociado con una única dirección que el equipo conoce y maneja (no importa el procedimiento).

Cuando se pasa un valor por referencia, se pasa la dirección del valor, lo que permite tener dos variables que apuntan a un mismo valor. Desafortunadamente, un ejemplo teórico de paso por referencia no es realmente considerado en esta etapa de la guía de aprendizaje, se verá al abordar el capítulo de creación de objetos. Dicho esto, cuando manejamos una página Web con DOM, nos encontramos con referencias, como en el siguiente capítulo sobre los hechos.

Referencias con DOM

Esquematizar el diseño de referencia con DOM es bastante simple: dos variables pueden acceder al mismo elemento. Se muestra en este ejemplo:

Código: JavaScript

```
var newLink = document.createElement('a');
var newLinkText = document.createTextNode('UMU');

newLink.id = 'sdz_link';
newLink.href = 'http://www.um.es/docencia/barzana';
newLink.appendChild(newLinkText);
document.getElementById('myP').appendChild(newLink);
// Se recupera a través de su ID, el elemento recién insertado
var sdzLink = document.getElementById('sdz_link');
sdzLink.href = 'http://www.barzanallana.es';
// newLink.href muestra bien la nueva dirección URL:
alert(newLink.href);
```

La variable `Newlink` contiene realmente una referencia al elemento que se ha creado <a>. `Newlink` no contiene el elemento, contiene una dirección que apunta al famoso <a>. Una vez que el elemento se inserta en la página HTML, se puede acceder a él de muchas otras maneras, tales como con `getElementById()`. Cuando accedes a un elemento mediante `getElementById()`, se hace también por medio de una referencia.

Lo que hay que recordar acerca de todo esto es que a los objetos DOM se accede siempre por referencia, y esta es la razón por lo que este código no funciona:

Código: JavaScript

```
var newDiv1 = document.createElement('div');
var newDiv2 = newDiv1; // Se copia <div>
```

Si lo has seguido, newDiv2 contiene una referencia que apunta hacia el mismo <div> que newDiv1. Pero cómo.

Clonar, sustituir, eliminar … Clonación de un elemento

Para clonar un objeto, nada más simple: `cloneNode()`. Este método requiere un parámetro booleano (`true o false`): si se desea clonar el nodo (`true`) o no (`false`) sus hijos y atributos. Pequeño ejemplo simple: crear una <hr />, y un segundo <hr>, clonando la primera:

Código: JavaScript

```
// Vamos a clonar un elemento creado:
var hr1 = document.createElement ('h');
var hr2 = hr1.cloneNode (false) / / No tiene hijos …
```

```
// Se clona un elemento existente:
var parrrafo1 = document.getElementById(mi'P');
var parrafor2 = parrafo1.cloneNode(true);

/ / Y ten cuidado, el elemento es clonado, pero no "insertado" como no llamemos appendChild ():
parrafo1.parentNode.appendChild(parrafo2);
```

Una cosa muy importante que hay que recordar, concerniente al próximo capítulo, es que el método `cloneNode ()` no copia eventos asociados y creados con DOM (con `addEventListener ()`), incluso con un parámetro en `true`.

Reemplazar un elemento por otro

Para reemplazar un elemento con otro, nada más simple, hay `replaceChild()`. Este método acepta dos parámetros: el primero es el nuevo elemento y el segundo es el elemento a reemplazar. Como `cloneNode ()`, este método se utiliza en todos los tipos de nodos (elementos, nodos de texto, etc) .

En el siguiente ejemplo, el contenido textual (recuerda, este es el primer hijo <a>) del enlace será reemplazado por otro. El método `ReplaceChild()` se ejecuta en el elemento <a>, es decir, el nodo padre del nodo que va a ser reemplazado.

Código: HTML

```
<body>
<div>
	<p id="miP"> Un poco de texto <a>y un enlace </ a> </ p>
</div>
<script>
	var link = document.getElementsByTagName ('a') [0];
	var = newLabel document.createTextNode ('y un enlace a');
	link.replaceChild (newLabel, link.firstChild);
</script>
</body>
```

Eliminar un elemento

Para insertar un elemento se utiliza `appendChild()`, y para borrar uno, `removeChild ()`. Este método toma como parámetro el nodo hijo a eliminar. Si se superpone el código HTML en el ejemplo anterior, el script es el siguiente:

Código: JavaScript

```
var link = document.getElementsByTagName ('a') [0];
link.parentNode.removeChild (link);
```

No hay necesidad de recuperar miP (elemento padre) con `getElementById()`, tal como directamente con `parentNode`.

Ten en cuenta que `removeChild ()` devuelve el elemento eliminado, lo que significa que es perfectamente posible eliminar un elemento HTML y luego volver a donde se desee en DOM:

Código: JavaScript

```
var link = document.getElementsByTagName ('a') [0];

var = oldLink link.parentNode.removeChild (enlace) // Se elimina elemento y se tiene en stock

document.body.appendChild(oldLink) // A continuación, restablece el elemento eliminado donde quieras y cuando quieras
```

Otras acciones. Comprueba si hay elementos hijos

Nada es más fácil de verificar que la presencia de elementos hijos: `hasChildNodes()`. Sólo tienes que utilizar este método en el elemento deseado, si el elemento tiene al menos un hijo, el método devolverá true:

Código: HTML

```
<div>
<p id="miP"> Un poco de texto  <a> y un enlace </ a> </ p>
</div>
<script>
var = document.getElementsByTagName párrafo ('p') [0];
alert (paragraph.hasChildNodes ()) / / Muestra verdadero
</script>
```

Introducir en el lugar correcto: insertBefore ()

El método `insertBefore()` sirve para insertar un elemento antes de que otro. Toma dos parámetros: el primero es el elemento a insertar, mientras que el segundo es el elemento delante del cual se inserta el elemento.

Ejemplo:

Código: HTML

```
<p id="miP"> Un poco de texto <a> y un enlace </ a> </ p>
<script>
	var parrafo= document.getElementsByTagName('p')[0];
	var enfasis = document.createElement('em'),
	enfasisText = document.createTextNode(' un énfasis  ligero');
	enfasis.appendChild(enfasisText);
	parrafo.insertBefore(enfasid, parrafoastChild);
</script>
```

Como para `appendChild ()`, este método se aplica al elemento padre.

Un buen consejo: insertAfter ()

JavaScript proporciona `insertBefore()`, pero no `insertAfter()`. Es una lástima, porque aunque podamos pensar, lo contrario a veces es bastante útil. Si no existe, se podría crear esa función. Por desgracia, no es posible en esta etapa del curso, crear un método que se aplicaría de la siguiente manera:

Código: JavaScript

```
element.insertAfter (newElement, afterElement)
```

No, vamos a tener que conformarnos con una "simple" función:

Código: JavaScript

```
insertAfter (newElement, afterElement)
```

- Algoritmo

Para insertar después de un elemento, primero se ha de recuperar el elemento principal. Esto es lógico, ya que la inserción del elemento será ya sea a través de `appendChild()` o a través de `insertBefore()`: si deseas agregar este elemento después del último hijo, es simple. Basta con aplicar `appendChild()`. Por contra, si el elemento detrás del cual deseas insertar este elemento no es el último que se utiliza `insertBefore()` dirigida al próximo hijo con `nextSibling`:

Código: JavaScript

```
function insertAfter (newElement, afterElement) {
```

```
var parent = afterElement.parentNode;
if (parent.lastChild === afterElement) {// Si el último elemento es el mismo que el elemento detrás del cual se ha de insertar, simplemente haz appendChild ()
parent.appendChild (newElement);
} else { /  En caso contrario, se hace una insertBefore () en elemento siguiente
parent.insertBefore (newElement, afterElement.nextSibling);
}
}
```

Resumen

Una vez que se ha accedido a un artículo, puedes navegar a otros elementos con `parentNode`, `previousSibling` y `nextSibling` y recuperar información acerca de los nombres de los elementos y su contenido.

Para agregar un elemento, primero debes crearlo, luego agregar atributos a él y finalmente insertarlo en la ubicación deseada dentro del documento.

Además del "paso por valor" JavaScript tiene un "paso por referencia", que es común al manejar DOM. Es esta historia de referencia la que nos obliga a utilizar un método tal como `cloneNode()` para duplicar un elemento. De hecho, copiar la variable que apunta a este elemento es inútil.

Si `appendChild()` es particularmente útil, `insertBefore()` también lo es para insertar un elemento antes de otro. Crear una función `InsertAfter()` es bastante sencillo y puede hacernos ahorrar tiempo.