estudios abiertos
SEAS
GRUPO SANVALERO

2 UNIDAD DIDACTICA

Lenguaje JavaScript

2. DOM y Eventos en JavaScript

ÍNDICE

OBJETIVOS

- En esta unidad vamos a entrar en profundidad en el DOM (Document Objet Model) y como funcionar con él.
- Veremos su estructura, como se jerarquiza en un árbol de nodos y que tipos de nodos existen.
- Aprenderemos a acceder directamente a ellos, mediante las instrucciones getElementById, getElementByTagName y getElementByName.
- También aprenderemos a acceder y/o modificar los atributos y propiedades de los nodos de XHTML.
- Descubriremos como insertar, copiar y eliminar los nodos dinámicamente.
- Veremos los eventos, los tres modelos de eventos que existen y los manejadores del modelo básico de eventos.
- Capturaremos los eventos del teclado, los eventos del ratón y los diferentes tipos de eventos que existen.

INTRODUCCIÓN

En esta unidad vamos a profundizar en el funcionamiento del DOM y de los eventos en JavaScript.

La creación del *Document Object Model* ha sido una de las innovaciones más influyentes para el desarrollo de las páginas web dinámicas y de las aplicaciones web más complejas.

El DOM ha permitido a los programadores web acceder y manipular las páginas XHTML de una forma sencilla, como si se tratara de documentos XML. Originalmente DOM se diseñó para manipular los documentos XML.

Ahora mismo es una de las utilidades disponibles para la mayoría de lenguajes de programación (JavaScript , PHP, Java), siendo la forma de implementarlo la única diferencia entre ellos.

Podemos definir el DOM como una estructura de objetos que genera automáticamente el navegador cuando se carga un documento y que se puede alterar mediante código JavaScript para cambiar sus contenidos dinámicamente y el aspecto de la página.

Los **eventos** van a definir lo que va a hacer el usuario en la página, o en el elemento de la página al que se le aplica. Los eventos son cualquier acción que el usuario va a realizar: hacer click o doble click, mover el ratón, pulsar una tecla, entrar con el ratón en un elemento, seleccionarlo con el ratón o con la tecla TAB, etc. Incluso acabar de cargarse la página podemos decir que es un evento.

Con JavaScript detectaremos cuando se produce un determinado evento, y aplicarle un código que se ejecute cuando se produzca el evento. Así interactuamos con el usuario, el cual, cuando provoque el evento, desencadenará la ejecución del código JavaScript asociado.

2.1. DOM

Empezaremos definiendo que es DOM. Es la abreviatura de Document Objet Model. Lo podríamos traducir como un Modelo de Objeto de Documento, aunque nos podemos referir al DOM con el nombre de jerarquía de objetos del navegador, porque exactamente es una estructura jerárquica en el que existen varios objetos y unos dependen de los otros.

Las funciones de los objetos del DOM van a modelizar las ventanas del navegador, el historial de navegación, el documento o incluso la propia página web, aparte de los elementos que contiene la propia página, como las divisiones, los párrafos, formularios, tablas, etc. Mediante el DOM y a través de JavaScript, podemos acceder a cualquiera de estos elementos anteriormente descritos, o mejor dicho, a sus correspondientes objetos, y poder alterar las propiedades de estos o llamar a sus métodos. De esta manera, va a quedar disponible cualquier elemento de la página, y poder modificar, suprimir e incluso crear nuevos elementos y situarlos en la página.

Aunque hasta ahora no hayamos hablado del DOM como tal, durante este temario ya hemos tocado el mismo alguna vez. Por ejemplo, cuando hemos hecho alguna sentencia en la que hemos accedido al valor de un campo de formulario, con un código como este:

```
document.forms[0].elements.length
```

Lo que realmente hemos hecho es acceder a la estructura de objetos del DOM.

El DOM está definido y administrado por el W3C, por lo que los distintos navegadores aplican dichas especificaciones, dando soporte al DOM en sus aplicaciones. El DOM además de modificar páginas web en HTML, también puede modificar documentos XML.

El Consorcio World Wide Web (W3C) es una comunidad internacional donde las organizaciones Miembro, personal a tiempo completo y el público en general trabajan conjuntamente para desarrollar estándares Web. Para más información puede visitar su página web *http://www.w3c.es/*

Debemos saber que el DOM se organiza en niveles. Dichos niveles cambian de versión a versión del navegador y cada uno de los desarrolladores de los navegadores, entienden las especificaciones del DOM de manera distinta, por lo que se debe tener cuidado y plantear su posible implementación para que el código funcione con un navegador u otro.

2.1.1. Compatibilidad del DOM entre navegadores

A la hora de realizar páginas webs, uno de los problemas más comunes que nos solemos encontrar es que los distintos navegadores interpretan de diferente manera el mismo código. Esta situación nos va a ocurrir con cualquier lenguaje interpretado en el lado del cliente, ya sea *HTML, CSS* o *JavaScript*. El problema con *JavaScript* se acrecienta aún más, puesto que las diferencias entre navegadores son demasiado grandes e incluso entre versiones del mismo explorador.

Para corregir estos problemas, se pueden "solucionar" mediante una serie de técnicas que nos permitirán primeramente detectar el navegador del usuario para posteriormente ejecutar unas u otras sentencias.

Por desgracia, todas estas técnicas solo nos complican el código y el posible desarrollo de aplicaciones web con programación del lado del cliente. Para solucionarlo tenemos a nuestra disposición librerías o frameworks de *JavaScript,* que nos van a permitir realizar la programación, primero, sin tener que preocuparnos por las características de los distintos navegadores, y segundo, nos ofrecen una serie de funciones avanzadas para desarrollar interfaces de una manera sencilla y rápida.

Un ejemplo de estos frameworks es JQuery y lo veremos en unidades siguientes.

2.2. Introducción al DOM

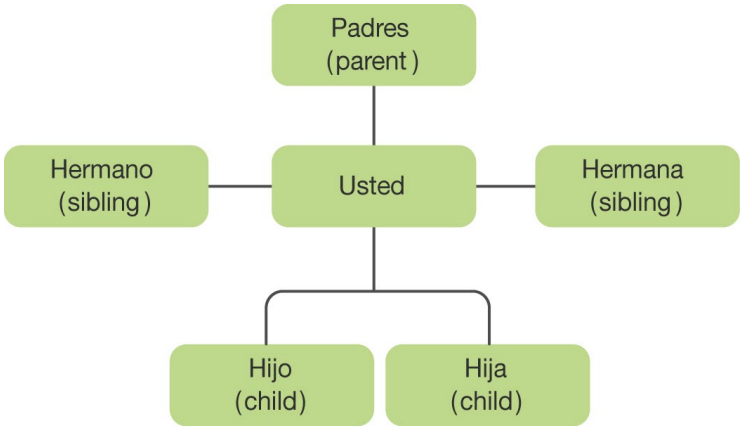
Como hemos comentado anteriormente, el DOM *(Document Object Model)* es la forma de acceder a los objetos de la pantalla. Y como ya hemos visto, existen numerosos navegadores que nos permiten el acceso a las capas con instrucciones como `document.layers`, `document.all`, `document.getElementById`. Para estandarizar el modelado del DOM, el W3C ha creado un modelo que actualmente implementan los navegadores.

Si no estás familiarizado con el lenguaje HTML, deberás estudiar el anexo de este temario, para tener la información sobre las etiquetas HTML, sus atributos y sus métodos.

2.2.1. Estructura del DOM

La clave del DOM es su estructura jerárquica y su estructura de "árbol genealógico". Con `document` nos referimos a la página actual y al contenido de esta página, sería la parte superior del árbol, y todo lo demás está por debajo de ella. Por ejemplo, la etiqueta `<html>` se encuentra bajo el documento y la etiqueta `<head>` se encuentra bajo la `<html>`, que como hemos dicho antes, está bajo el documento, etc.

Para tratar de entender DOM y lo que es, vamos a usar un ejemplo de un árbol genealógico. La parte superior sería la de nuestros padres y la siguiente rama seríamos nosotros, la siguiente nuestros hijos, etc.



En un documento HTML, la parte superior sería nuestro documento, y las ramas del árbol DOM, que las llamaremos nodos. Todos los elementos de una página son nodos. Un ejemplo de nodos serían las etiquetas `<li>`, `<p>` y `<td>`. De hecho, todos los atributos, tales como `align="center"`, son hasta cierto punto, nodos. Incluso el valor de un atributo (por ejemplo `"center"`) podríamos considerarlo como nodo (un nodo de texto).

Veamos ahora como sería el árbol genealógico del DOM, y cómo atravesar un documento con DOM (tengamos en cuenta que cuando hablemos del DOM a partir de ahora, nos estamos refiriendo a W3C DOM en lugar del actual DOM implementado en la mayoría de navegadores).

Como ya sabemos, podemos acceder a una capa con `document.getElementById()` si sabemos el id. Una vez que tengamos ese objeto, podemos acceder a otros niveles con relación a él. Echemos un vistazo a ejemplos de código y luego veremos qué relación tiene con nuestro árbol genealógico.

- **parentNode:** accedemos al nodo padre.

  Por ejemplo, podríamos usar:

  ```
  document.getElementById("myElement").parentNode.
  ```

- **childNodes:** da una lista de nodos de los hijos.

  Ejemplo: `document.getElementById("myElement").childNodes` A partir de ahí, tenemos un conjunto de nodos, que pueden acceder a la primera de este modo:

  ```
  document.getElementById("myElement").childNodes.item(0)
  ```

  Tenga en cuenta que `childNodes.item(0)` es el mismo que `childNodes.item[0]`

- **firstChild:** accede a nuestro primer nodo secundario.

  Ejemplo: `document.getElementById("myElement").firstChild es lo mismo que document.getElementById("myElement").childNodes[0].`

- **lastChild:** accede a nuestro último nodo hijo.

  Ejemplo: `document.getElementById("myElement").lastChild es lo mismo que hacer document.getElementById("myElement").childNodes.item(document.getElementById("myElement").childNodes.length).`

- **previousSibling:** accede al nodo del mismo nivel anterior a este.

  Ejemplo:`document.getElementById("myElement").previuosSibling.`

- **nextSlibling:** accede al nodo del mismo nivel siguiente a este.

  Ejemplo: `document.getElementById("myElement").nextSibling.`

2.2.2. Árbol de Nodos

Cosas habituales que nos van a surgir a partir de ahora, en lo que es programación web, consistirá en manipular dichas páginas web. Lo normal será tener que obtener los valores almacenados por ciertos elementos (como por ejemplo los elementos de una formulario), o crear otros elementos como párrafos, , etc, y hacerlo de forma dinámica añadiéndolo a la página, o realizar una animación a algún elemento, ya sea que lo hagamos desaparecer y aparecer, moverlo por la página…

Todo lo comentado hasta ahora será sencillo de realizar gracias al DOM. Pero para poder conseguir esto, debemos realizar una transformación de la página original. Como sabrás, una pagina HTML, es una sucesión de caracteres, y estos son difíciles de manipular. De esta manera, los navegadores realizar automáticamente esta transformación en estructuras mas fáciles de manipular, permitiéndonos utilizar todas las herramientas del DOM.

Lo que realiza el DOM es transformar los documentos XHTML en conjuntos de elementos llamados nodos, todos estos interconectados y que representarán los contenidos de las páginas y las relaciones entre todos ellos. Por su aspecto, la unión de todos los nodos se llama "árbol de nodos".

Veamos con un ejemplo esto que acabamos de explicar. En la siguiente página XHTML sencilla:

```
<!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=iso-8859-1" />

  <title>Ejemplo de página sencilla</title>

</head>

<body>

  <p>

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

</body>

</html>
```

El ejemplo anterior lo podemos transformar en el siguiente árbol de nodos:

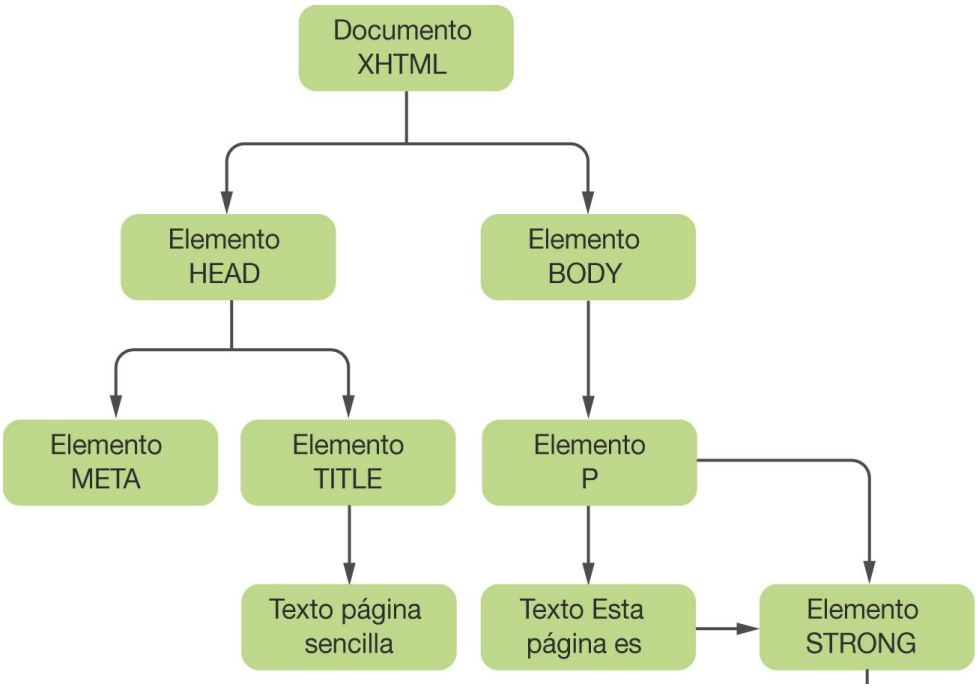


Figura 2.71. Árbol de nodos generado automáticamente por DOM a partir del código XHTML de la página.

Pasemos a explicar la figura anterior. En ella, cada rectángulo va a representar un nodo DOM y las flechas nos van a indicar las relaciones entre dichos nodos. Dentro de cada nodo hemos incluido su tipo (que se verá más adelante) y su contenido.

La raíz del árbol de nodos de una página XHTML siempre va a ser la misma: un nodo de tipo especial denominado "Documento". A partir de ese nodo, cada etiqueta se transformará en un nodo de tipo "Elemento". Esta conversión de etiquetas en nodos se va a realizar de una manera jerárquica. De esta forma, del nodo raíz solamente pueden derivar los nodos HEAD y BODY. A partir de esta primera derivación, cada etiqueta XHTML se irá transformando en un nodo que derivará del nodo correspondiente a su "etiqueta padre".

La transformación de las etiquetas XHTML habituales va a generar dos nodos: el primero que será del tipo "Elemento" (que corresponderá a la propia etiqueta XHTML) y el segundo, que será de tipo "Texto" y que contendrá el texto encerrado por dicha etiqueta XHTML.

De esta manera, la siguiente etiqueta XHTML:

```
<title>Ejemplo de página sencilla</title>
```

Nos va a generar dos nodos:

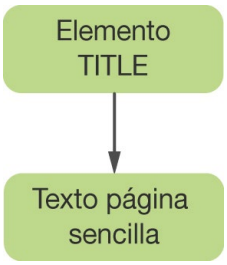


Figura 2.72. Nodos generados automáticamente por DOM para una etiqueta XHTML sencilla.

De la misma forma, la siguiente etiqueta XHTML:

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

Nos va a generar los siguientes nodos:

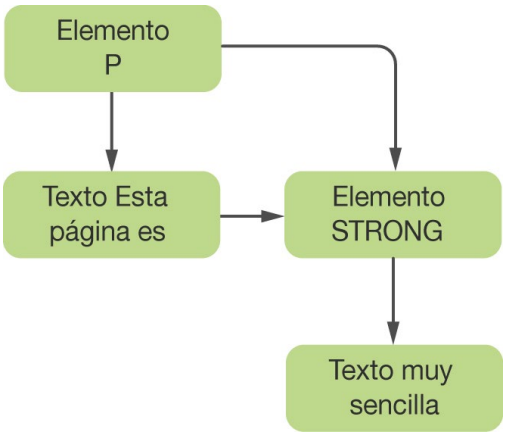


Figura 2.73. Nodos generados automáticamente por DOM para una etiqueta XHTML con otras etiquetas XHTML en su interior.

- Nodo de tipo “Elemento” que corresponde a la etiqueta <p>.
- Nodo de tipo “Texto” con el contenido texto de la etiqueta <p>.
- Como el contenido de <p> ya incluye en su interior otra etiqueta XHTML, la etiqueta interior se va a transformar en un nodo de tipo “Elemento” que representará la etiqueta y que derivará del nodo anterior.
- El contenido de la etiqueta generará a su vez otro nodo de tipo “Texto” que derivará del nodo generado por .

La transformación automática de la página en un árbol de nodos siempre sigue las mismas reglas:

- Las etiquetas XHTML se van a transformar en dos nodos: uno será la propia etiqueta y el otro nodo será hijo del primero y consistirá en el contenido de texto de la etiqueta.

- Si nos encontramos una etiqueta XHTML dentro de otra, seguiremos el procedimiento anterior, pero con la condición que los nodos generados, serán nodos hijo de su etiqueta padre.

Como debemos suponer, los árboles generados por las páginas XHTML habituales van a contener árboles con miles de nodos. No debemos preocuparnos por esto, ya que, el proceso de transformación es rápido y automático, siendo las funciones proporcionadas por DOM, las únicas que nos permitirán acceder a cualquier nodo de la página de una manera sencilla e inmediata.

2.2.3. Tipos de nodos

La especificación completa de DOM define 12 tipos de nodos, aunque las páginas XHTML habituales se pueden manipular manejando solamente cuatro o cinco tipos de nodos:

- *Document:* es el nodo raíz del que van a derivar todos los demás nodos del árbol.
- *Element:* con este vamos a representar cada una de las etiquetas XHTML. Va a ser el único nodo que podrá contener atributos y el único del que podrán derivar otros nodos.
- *Attr:* se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas XHTML, es decir, uno por cada par atributo=valor.
- *Text:* nodo que contiene el texto encerrado por una etiqueta XHTML.
- *Comment:* representa los comentarios incluidos en la página XHTML.

El resto de nodos existentes y que no vamos a considerar son DocumentType, CDataSection, DocumentFragment, Entity, EntityReference, ProcessingInstruction y Notation.

2.2.4. Acceso directo a los nodos

Una vez que ya se ha construido automáticamente todo el árbol de nodos DOM, ya podemos acceder de forma directa a cualquier nodo del árbol utilizando las funciones DOM. Como ya veremos más adelante, acceder a un nodo del árbol es lo mismo que acceder a “un trozo” de la página, una vez que se ha construido dicho árbol, ya es posible manejar de una manera fácil la página: podremos acceder al valor de un elemento, establecer el valor de dicho elemento, o mover el elemento dentro de la página, e incluso crear y añadir nuevos elementos, etc.

DOM nos proporciona dos métodos alternativos para poder acceder a un nodo determinado: acceder a ellos a través de sus nodos padre y/o realizar un acceso directo.

Para acceder a un nodo a través de sus nodos padre, DOM nos proporciona funciones que consisten en acceder primeramente al nodo raíz de la página y posteriormente a sus nodos hijos y a su vez, a los nodos hijos de esos hijos y así sucesivamente hasta llegar al último nodo de la rama finalizada por el nodo buscado. Pero, como ya veremos, cuando queramos acceder a un nodo específico, será mucho más rápido acceder directamente a ese nodo sin tener que pasar hasta él a través de todos sus nodos padre.

De esta manera, no se van a ver en este punto las funciones necesarias para el acceso jerárquico de nodos y únicamente vamos a presentar las que permiten acceder de forma directa a dichos nodos.

Por último, debemos recordar que solamente, cuando el árbol DOM ha sido construido completamente, podremos realizar el acceso a los nodos, a su modificación y su eliminación, es decir, después de que la página XHTML se cargue por completo. Más adelante veremos como asegurar que un código JavaScript solamente se ejecute cuando el navegador ya ha cargado completamente la página XHTML.

2.2.5. getElementsByTagName()

Como veremos que va a suceder con todas las funciones que nos proporciona DOM, esta tiene un nombre muy largo, pero lo hace autoexplicativo.

La función `getElementsByTagName(nombreEtiqueta)` obtendrá todos los elementos de la página XHTML, cuya etiqueta será igual que el parámetro que se le pase a la función.

Por ejemplo, veamos cómo obtener todos los div de una página XHTML:

```
var parrafos = document.getElementsByTagName("p");
```

Vamos a explicar esta línea: lo primero es el valor que se escribe delante del nombre de la función, en este caso, *document,* va a ser el nodo desde el cual se realiza la búsqueda de los elementos. En este ejemplo, como queremos obtener todos los párrafos de la página, vamos a utilizar el valor document como punto de partida de la búsqueda.

Lo que nos va a devolver la función, va a ser un array con todos los nodos que cumplen dicha condición, la de que su etiqueta coincida con el parámetro pasado a la función. El valor devuelto va a ser un array de nodos DOM, no un array de cadenas de texto o un array de objetos normales. Por ello, debemos procesar cada valor del array de la siguiente manera.

Para obtener el primer párrafo de la página escribiremos:

```
var primerParrafo = parrafos[0];
```

Si en cambio, queremos recorrer todos los párrafos de la página podríamos insertar un *for* para recorrer todo el array:

```
for (var i = 0; i < parrafos.length; i++) {
  var parrafo = parrafos[i];
    }
```

La función getElementsByTagName() la podremos aplicar de forma recursiva a cada uno de los nodos que nos devuelva la función. Por ejemplo, para obtener todos los enlaces del primer párrafo de la página, procederíamos de la siguiente manera:

```
var parrafos = document.getElementsByTagName("p");
var primerParrafo = parrafos[0];
var enlaces = primerParrafo.getElementsByTagName("a");
```

2.2.5.1. getElementsByName()

Esta función es similar a la anterior, pero en vez de buscar los elementos por su tipo de etiqueta, realiza la búsqueda cuyo atributo `name` sea igual al parámetro pasado.

Veamos un ejemplo, en el que se obtiene directamente el único párrafo con el nombre indicado:

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>Ejemplo de getElementsByName() </title>
    <script type="text/javascript">
       var parrafoEspecial = document getElementsByName("especial");
  </script>
</head>
<body>
    <p name="prueba">
        ...</p>
    <p name="especial">
    ...</p>
  <p>
    ...</p>
</body>
```

Lo normal, es que el atributo `name` sea único para los elementos HTML que lo definen, por lo tanto, es un método muy práctico y rápido para acceder directamente al nodo deseado.

También debemos tener en cuenta, que en el caso de los elementos HTML `radiobutton`, el atributo `name` es común a todos los `radiobutton` que están relacionados, por lo tanto, la función devolverá una colección de elementos.

NOTA

Internet Explorer 6.0 no implementa correctamente esta función, ya que sólo la tiene en cuenta para los elementos de tipo <input> y <img>. Además, si usamos esta función, también tiene en consideración los elementos cuyo atributo id sea igual al parámetro de la función.

2.2.5.2. getElementById()

Esta función va a ser la más utilizada cuando se desarrollen aplicaciones web dinámicas. Es la función preferida para acceder directamente a un nodo y poder leer o modificar sus propiedades.

`GetElementById()` nos devolverá directamente el elemento XHTML cuyo atributo id coincide con el parámetro pasado en la función. Como el atributo id si que debe ser único para cada elemento de una misma página, la función nos devolverá solamente el nodo deseado.

```
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>Ejemplo de getElementById()</title>
  <script type="text/javascript">
        var cabecera = document.getElementById("cabecera");
  </script>
</head>
<body>
    <div id="cabecera">
        <a href="/" id="logo">...</a>
    </div>
</body>
```

Esta función es tan importante y tan utilizada en todas las aplicaciones web, que casi todos los ejemplos y ejercicios que siguen la van a utilizar constantemente.

NOTA

Internet Explorer 6.0 tampoco interpreta incorrectamente esta función. Cuando devuelve los elementos que coinciden con la id pasada, también lo hace con aquellos elementos cuyo atributo name coincida con el parámetro proporcionado a la función.

2.2.6. Acceso directo a los atributos XHTML

Una vez que ya hemos accedido a un nodo, el siguiente paso consiste en acceder y/o modificar sus atributos y propiedades. Mediante DOM, accederemos de forma sencilla a todos los atributos XHTML y todas las propiedades CSS de cualquier elemento de la página.

NOTA

Debemos recordar que los atributos XHTML de los elementos de la página se transforman automáticamente en propiedades de los nodos. Para acceder a su valor, simplemente se indica el nombre del atributo XHTML detrás del nombre del nodo.

Vamos a ver un ejemplo en el que obtenemos de forma directa la dirección a la que enlaza el enlace:

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-88-1"/>
  <title>Ejemplo de getElementById()</title>
    <script type="text/javascript">
    window.onload = function () {
      var enlace = document.getElementById("enlace");
            alert(enlace.href); // muestra http://www...com
    }
  </script>
</head>
<body>
    <a id="enlace" href="http://www.seas.com">Enlace</a>
</body>
</html>
```

En este ejemplo, hemos obtenido el nodo DOM que representa el enlace mediante la función document.getElementById(). A continuación, se obtiene el atributo href del enlace mediante `enlace.href`. Para obtener por ejemplo el atributo id, se utilizaría enlace.id.

Las propiedades CSS son más complicadas de obtener. Para obtener el valor de cualquier propiedad CSS del nodo, deberemos utilizar el atributo style. El siguiente ejemplo obtiene el valor de la propiedad margin de la imagen:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>Ejemplo de getElementById()</title>
    <script type="text/javascript">
    window.onload = function () {
      var imagen = document.getElementById("imagen");
       alert(imagen.style.margin);
         }
  </script>
</head>
<body>
    <img id="imagen" style="margin:0; border:0;" src="imagen.png" />
</body>
</html>
```

Si el nombre de una propiedad CSS es compuesto, accederemos a su valor modificando ligeramente su nombre:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>Ejemplo de getElementById()</title>
    <script type="text/javascript">
    window.onload = function () {
      var parrafo = document.getElementById("parrafo");
            alert(parrafo.style.fontWeight); // muestra "bold"
    }
```

```
  </script>
</head>
<body>
    <p id="parrafo" style="font-weight: bold;">...</p>
</body>
</html>
```

La manera de realizar la transformación del nombre de las propiedades CSS compuestas es muy sencilla. Debemos seguir la siguiente regla:

Transformación de etiquetas compuestas. Consiste en eliminar todos los guiones medios (-) y escribir en mayúscula la letra siguiente a cada guion medio.

A continuación se muestran algunos ejemplos:

- **font-weight** se transforma en *fontWeight*
- **line-height** se transforma en *lineHeight*
- **border-top-style** se transforma en *borderTopStyle*
- **list-style-image** se transforma en *listStyleImage*

El único atributo XHTML que no tiene el mismo nombre en XHTML y en las propiedades DOM es el atributo class. Al ser una palabra reservada por JavaScript, no se podrá utilizar para acceder al atributo class del elemento XHTML.

En su lugar, DOM utiliza el nombre `className` para acceder al atributo class de XHTML.

Veamos un ejemplo de cómo hacerlo:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>Ejemplo de getElementById()</title>
  <script type="text/javascript">
    window.onload = function () {
```

```
        var parrafo = document.getElementById("parrafo");
              alert(parrafo.class); // muestra "undefined"
         alert(parrafo.className); // muestra "normal"
      }
  </script>
</head>
<body>
    <p id="parrafo" class="normal">
    ...</p>
</body>
</html>
```

2.2.7. Otras formas de acceder a un elemento

Hasta ahora hemos visto que podemos acceder a un elemento (por ejemplo, una capa) por su ID con `document.getElementById(ID)`. También podemos acceder a los nodos del elemento con `document.getElementByTagName(tagName)`, que devuelve un Array de elementos con ese `tagName`.

Ejemplo: `document.getElementByTagName("TD").item(0)`

En el ejemplo, cogeremos la primera tabla (`<td>`) del documento. Para obtener un Array de todas las etiquetas en el documento, podemos utilizar la siguiente sintaxis:

```
document.getElementByTagName("*")
```

Por lo tanto, para acceder a la etiqueta `<body>`, se podría utilizar: `document.getElementByTagName("BODY").item(0)`

2.2.8. Advertencias

Antes de aplicar la información proporcionada en anteriores capítulos, debemos tener en cuenta dos advertencias:

- No es realista poder hacer referencia a otros nodos del documento, por algo diferente al atributo id en HTML (en XML sí que podríamos).
- Dentro del elemento `<p>` no se pueden apilar otros elementos, de acuerdo con la especificación HTML 4.01. Tal y como ha dicho el W3C, "El elemento P representa un párrafo. No puede contener otros niveles de elementos (incluyendo otros P dentro de sí mismo)."

Lo que quiere decir es que es muy poco realista para acceder a un elemento, el uso de su relación a otro (con parentNode, firstChild, previousSibling, etc). Y, por último, está diciendo que los elementos `<p>` no deben encontrarse dentro de otro elemento `<p>`.

2.2.9. Un ejemplo

Veamos un ejemplo en el que referenciamos a cada elemento de diferente forma

```
<body>
  <ol id="ol1">
    <li id="li1">
      <span id="span1">Texto 1 </span>
    </li>
    <li id="li2">
       <span id="span2">Texto 2 </span>
    </li>
    <li id="li3">
      <span id="span3">Texto 3 </span>
    </li>
  </ol>
</body>
```

Veamos cómo podemos acceder a la etiqueta `ol1` de diferentes maneras:

```
document.getElementById("ol1")
document.getElementsById("li1").parentNode
document.getElementsByTagName("ol").item(0)
document.getElementsByTagName("li").item(0).parentNode
document.getElementsByTagName("li").item(1).parentNode
document.getElementsByTagName("l1").item(2).parentNode
document.getElementsByTagName("span").item(2).parentNode.parentNode
document.getElementsByTagName("body").item(0).childNodes.item(0)
document.body.childNodes.item(0)
```

Debido a que probablemente vamos a acceder muchas veces a la etiqueta `<body>`, se puede utilizar un acceso directo, en lugar de utilizar su tagName de esta forma, `document.getElementsByTagName("body").item(0)`, o mediante la asignación de un ID al body y usarlo:

```
document.body
```

Para acceder a l2 lo podemos hacer de la siguiente forma:

```
document.getElementById("l2")

document.getElementById("l1").nextSibling

document.getElementById("l3").previousSibling

document.getElementsByTagName("li").item(1)

document.getElementsByTagName("span").item(1).parentNode

document.getElementsByTagName("span").item(2).parentNode.previousSibling

document.getELementsByTagName("ol").childNodes.item(1).parentNode.
childNodes.item(1).parentNode.childNodes.item(1)

document.getElementsByTagName("body").item(0).firstChild.firstChild.
nextSibling

document.body.childNodes.item(0).lastChild.previousSibling
```

Si tratamos de encontrar el nodo de texto con el siguiente texto - "Texto 3", podríamos hacerlo, en primer lugar accediendo a `span3`, y luego acceder a ella mediante su `firstChild`. Ejemplo:

```
document.getElementById("span3").firstChild
```

Antes dijimos que los atributos eran parecidos a los nodos. Es por eso, que no aparecen cuando se intenta acceder a los nodos de un elemento menor. Más adelante, veremos cómo cambiar, establecer y eliminar atributos, pero antes de eso, vamos a pasar a algunos otros elementos esenciales a los que se puede acceder desde un nodo del elemento.

2.2.10. Propiedades y métodos de los nodos

Seguimos con el ejemplo anterior de HTML:

```
<body>
    <ol id="ol1">
        <li id="li1">
            <span id="span1"> Texto 1 </span>
        </li>
        <li id="li2">
            <span id="span2"> Texto 2 </span>
        </li>
        <li id="li3">
      <span id="span3"> Texto 3 </span>
        </li>
    </ol>

</body>
```

- `hasChildNodes()`: nos devuelve un valor booleano de true o false, si el nodo tiene nodos secundarios.

  Por ejemplo, si escribimos la siguiente sentencia:

  `document.getElementById("span3").hasChildNodes()` nos devolverá true, pero sí en cambio escribimos:

  `document.getElementById("span3").firstChild.hasChildNodes()` nos devolverá falso porque un nodo de texto no puede tener hijos.

- `nodeName:` devuelve el "nombre" de un nodo:

  document.getElementById("span3").nodeName nos devolverá "SPAN"

- `tagName:` devuelve el tagName de un elemento.

  document.getElementById("span3").tagName devuelve "SPAN"

Llegados a este punto, podemos preguntarnos cuál es la diferencia entre `nodeName` y `tagName`. Bueno, si accedemos a un nodo de texto de `tagName`, no hay `tagName` porque no es una etiqueta (esto sólo funciona para los elementos), pero si tiene acceso a un nodo de texto de `nodeName`, se le dará "# texto". Por lo tanto:

`document.getElementById("span3").firstChild.nodeName` devuelve "#texto",

`document.getElementById("span3").firstChild.tagName` da "undefined"

- `nodeType` le dice de qué tipo de nodo es un elemento. Devuelve 1, 2 ó 3.
  - Devuelve 1, si se trata de un elemento.
  - Devuelve 2, si se trata de un atributo.
  - Devuelve 3, si se trata de texto.

  Por ejemplo:

  `document.getElementById("span3").nodeType` devuelve 1,

  `document.getElementById("span3").firstChild.nodeType` devuelve 3.

- `nodeValue:` nos devuelve el valor del nodo. Podemos usar esto para recuperar o modificar este valor. Si el nodo es un nodo de texto (`nodeType` de 3), nos devuelve el texto, si se trata de un atributo (`nodeType` de 2), nos devolverá el valor de ese atributo, y si se trata de un elemento (`nodeType` de 1), nos devolverá un valor nulo. Por ejemplo:

  `document.getElementById("span3").firstChild.nodeValue` nos devuelve "Texto 3", mientras que un elemento nos devolverá un valor nulo:

  `document.getElementById("span3").nodeValue` devuelve un valor nulo.

2.2.11. Get, Set, y Delete valores de atributo

Como ya hemos dicho anteriormente, todos los elementos de una página (incluyendo atributos) son un nodo. A diferencia de lo que podría haber pensado, los atributos no son considerados como hijos del elemento que se aplican. Aquí, vamos a ver cómo recuperar, establecer y quitar atributos. Imaginemos el siguiente ejemplo:

```
<iframe id="myIFrame" src="Pagina1.html"
align="center" width="400" height="200">
</iframe>
```

A este `<iframe>` podremos acceder a través de la siguiente sentencia:

```
document.getElementById("myIFrame")
```

Ahora, echemos un vistazo de cómo acceder a los atributos de este elemento.

- `getAttribute(atributo)`: toma un argumento, el atributo a recuperar y devuelve su valor. Ejemplo:

  ```
  document.getElementById("myIFrame").getAttribute("width")
  devuelve "400"
  ```

- `setAttribute(atributo, nuevoValor)`: toma dos argumentos, el atributo cuyo valor desea cambiar (o crear) y el nuevo valor. Por ejemplo:

  ```
  document.getElementById("myIFrame").setAttribute("height","50")
  ```
  Ahora, el `width` de `myIFrame` es de 200. Ten en cuenta el uso de comillas, incluso con los enteros. Esto se debe que a los atributos del elemento siempre deben ser una cadena (aunque por lo general los suele aceptar, incluso si no son cadena de texto). También podría cambiar en esta `<iframe>` el atributo `src` con la siguiente sintaxis:

  ```
  document.getElementById("myIFrame").setAttribute("src","otraPagina.
  html")
  ```

- `removeAttribute(atributo)`: toma como argumento el atributo que debe ser eliminado. Por ejemplo:

  ```
  document.getElementById("myIFrame").removeAttribute("width")
  ```
  eliminaría el atributo "`width`", y establecerá el ancho por defecto del navegador para la `<iframe>`.

2.2.12. Crear nodos nuevos

Además de trabajar con los nodos existentes y con sus métodos y propiedades (haschildnodes(), innertext, children, etc…), también podemos crear nuevos nodos.

- `document.createElement(tagName)`: crea un nuevo elemento nodo con el `tagName` especificado. Realmente es increíble que se pueda crear simplemente de la nada. Se acepta un argumento, el `tagName` para crear. Vemos aquí un ejemplo:

```
newSPAN = document.createElement("SPAN");
```

  Observe que createElement() sólo se puede ejecutar desde el "documento", y que debe almacenar este nuevo elemento de nodo en una variable.

Más tarde, vamos a aprender qué hacer con este nuevo elemento y la forma de añadir (o insertarlo) en el documento.

- `document.createTextNode(texto)`: crea un nuevo nodo de texto. Se toma un argumento, el "texto" para el nodo de texto. Veamos un ejemplo:

```
newText = document.createTextNode("Este es el nuevo texto");
```

  Tenga en cuenta que esto no acepta HTML, sólo texto (en otras palabras, las etiquetas HTML se interpretan literalmente como texto normal).

Ahora, vamos a aprender a insertar el nodo de texto y el nuevo elemento que acabamos de crear.

2.2.13. Insertar nodos

- `appendChild(nodo_nuevo)` inserta el nuevo nodo al final de la lista de hijos del nodo donde se ejecutará. Echemos un vistazo al ejemplo. Tomaremos como argumento el nuevo nodo a añadir. Parece que suena difícil, pero no lo es.

Veamos un ejemplo, pero primero, vamos a necesitar algunos elementos en nuestra página, para que podamos insertar nuestro nuevo nodo después de ellos.

```
<body>
  <div align="center" id="div1">
    <span id="span1">Este es el texto de SPAN1</span>
    Este texto no esta en una etiqueta span.
  </div>
</body>
```

```
newSPAN.appendChild(newText);
document.getElementById("div1").appendChild(newSPAN);
```

```
alert(document.getElementById("div1").lastChild.nodeName);
```

Lo que hemos hecho es añadir el nodo de texto como el último hijo del nodo `newSPAN`. Después, hemos añadido el nodo `newSPAN` como el último hijo del nodo `div1`. A continuación se genera un mensaje de alerta de "SPAN", mientras que antes de que adjuntemos en el nuevo elemento nodo, nos han llegado un mensaje de alerta con el "#texto". También debemos tener en cuenta que añadimos el nodo de texto que hemos creado para la etiqueta `<span>` antes de a su vez añadir a la etiqueta `<span>` "`div1`" en el documento.

Podríamos haber añadido si queríamos la etiqueta `<span>` y la inserción del nodo de texto, de la siguiente manera:

```
document.getElementById("div1").appendChild(newSPAN);
document.getElementById("div1").lastChild.appendChild(newText);
alert(document.getElementById("div1").lastChild.nodeName);
```

2.2.14. Clonar nodos

- `cloneChild(nodo)`: es una función que copia y devuelve un nodo. Se acepta un argumento, y es la copia "en profundidad". Vamos a explicar esto en el siguiente ejemplo:

```
<body>
  <div id="div1">
    <span id="span1">Este es el texto del SPAN1</span>
    <span id="span2">este es el texto del SPAN2</span>
    Este texto no esta dentro de la etiqueta SPAN.
  </div>
</body>
```

Ahora, vamos a hacer la copia:

```
div2=document.getElementById("div1").cloneNode(false)
```

Simplemente va a copiar `<div id="div1">…</div>`, y no copiará nada entre las etiquetas. Nótese que también copia los atributos del nodo. En este momento, podemos agregar atributos, eliminar atributos, agregar los childs, sacar los childs, añadir ese nodo en cualquier parte del documento, etc... Ahora bien, si se ejecuta la siguiente línea:

```
div2=document.getElementById("div1").cloneNode(true)
```

Realizará la copia de los nodos secundarios de “div1”, y estaría copiando:

```
<div id=”div1”>
  <span id=”span1”>Este es el texto del SPAN1</span>
  <span id=”span2”>este es el texto del SPAN2</span>
  Este texto no esta dentro de la etiqueta SPAN.
</div>
```

Si no se define el argumento “profundidad” como falso o verdadero, es falso de forma automática. Una vez que un nodo es clonado, ya podría cambiar sus atributos, los nodos secundarios, sus nexos, etc.

2.2.15. Eliminar Nodos

- *removeChild(nodo):* elimina el nodo pasado como argumento. Tenga en cuenta que este método (al igual que los otros mencionados aquí) se ejecuta desde el padre del nodo que desea eliminar. Ejemplo:

```
<div id=”div1”>
  <span id=”span1”>Este es el texto del SPAN1</span>
  <span id=”span2”>este es el texto del SPAN2</span>
  Este texto no esta dentro de la etiqueta SPAN.
</div>
```

Ahora, si ejecutamos:

```
document.getElementById(“div1”).removeChild(document.
getElementById(“SPAN2”)).
```

Y se elimina el nodo “SPAN2” y sus nodos secundarios, dejando a nuestro documento el aspecto de:

```
<div id=”div1”>
  <span id=”span1”>Este es el texto del SPAN1</span>
  Este texto no esta dentro de la etiqueta SPAN.
</div>
```

- *replaceChild(newChildNode,oldChildNode):* remplaza el nodo “oldChildNode” con el nodo “newChildNode”. Por ejemplo, si tenemos este documento:

```
<div id=”div1”>
   <span id=”span1”>Este es el texto del SPAN1</span>
  Este texto no esta dentro de la etiqueta SPAN.
</div>
```

Y ejecutamos:

```
newSPAN = document.createElement("SPAN");
newSPAN.setAttribute("id","span2");
newSPAN.setAttribute("align","center");
newText = document.createTextNode("Este es el texto del SPAN2");
newSPAN.appendChild(newText);
div1 = document.getElementById("div1");
div1.replaceChild(newSPAN,div1.firstChild);
```

Ahora, nuestro documento modificado se verá así:

```
<div id="div1">
   <span id="span2" align="center">Este es el texto del SPAN2</span>
Este texto no esta dentro de una etiqueta span.
</div>
```

Esos son los principales métodos y propiedades de la W3C DOM Nivel 1.

2.3. Eventos

Hasta este momento, los scripts y aplicaciones que hemos ido creando tienen una cosa en común, o ejecutamos desde la primera instrucción hasta la última de una manera secuencial. Únicamente, por las estructuras de control de flujo (if, for, while) hemos modificado ligeramente este comportamiento, ya sea repitiendo algunos trozos del script y saltándonos otros trozos en función de algunas condiciones.

Vemos que con este tipo de aplicaciones no podemos realizar grandes cosas, y son poco útiles, al no interactuar con los usuarios y no poder responder a los diferentes eventos que se producen durante la ejecución.

2.3.1. Qué son los eventos

Por suerte, con el lenguaje JavaScript, las aplicaciones web creadas pueden utilizar el modelo de programación basada en eventos.

En este tipo de programación, necesitamos que el usuario interactúe con la página y sus scripts, ya sea que pulse una tecla, que mueva el ratón, que cierre la ventana del navegador, o cualquier otro evento que declaremos. En ese momento, es cuando el script va a responder a la acción del usuario normalmente procesando esa información y generando un resultado.

Mediante los eventos será posible que los usuarios pasen dicha información a los programas. JavaScript ya define numerosos eventos que permitirán una interacción completa entre usuario y páginas/aplicaciones web. Son eventos la pulsación de una tecla, pinchar o mover el ratón, seleccionar un elemento de un formulario, redimensionar la ventana del navegador, etc.

JavaScript nos va a permitir asignar una función a cada uno de los eventos. De esta forma, cuando se produzca, JavaScript ejecutará la función asociada a dicho evento.

Este tipo de funciones se denominan *"event handlers"* en inglés y suelen traducirse por *"manejadores de eventos"*.

2.3.2. Modelos de eventos

Debido a las incompatibilidades entre navegadores crear páginas y aplicaciones web siempre ha sido mucho más complejo de lo que debería. Incluso existiendo decenas de estándares para las tecnologías empleadas, los navegadores del mercado no los soportan completamente o incluso los ignoran.

Las principales incompatibilidades que muestran los navegadores:

- Se producen en el lenguaje XHTML.
- En el soporte de hojas de estilos CSS.
- Y sobre todo en la implementación de JavaScript.

De todas ellas, la más importante se da en el modelo de eventos del navegador. Por ello, existen hasta tres modelos diferentes de manejadores de eventos, y todo ello, dependiendo del navegador en el que se ejecute la aplicación.

Los tres modelos de eventos que existen actualmente son:

- Modelo básico de eventos.
- Modelo de eventos estándar.
- Modelo de eventos de Internet Explorer.

2.3.2.1. Modelo básico de eventos

Este modelo se introdujo para la versión 4 del estándar HTML y es parte del nivel más básico de DOM. Aunque posee características limitadas, es el único modelo compatible en todos los navegadores y el único que permite crear aplicaciones que funcionan de la misma manera en todos los navegadores.

Tipos de eventos

En este modelo, cada elemento o etiqueta XHTML define su propia lista de posibles eventos que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos XHTML diferentes y un mismo elemento XHTML puede tener asociados varios eventos diferentes.

Los nombres de los eventos se construirán mediante el prefijo on, seguido del nombre en inglés de la acción asociada al evento. Así, por ejemplo, onclick será el evento de pinchar un elemento con el ratón y onmouseover será el evento asociado a la acción de mover el ratón.

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>,<textarea>, <body>
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
ondblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>,<textarea>, <body>
onkeydown	Pulsar una tecla (sin soltar)	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	La página se ha cargado completamente	<body>
onmouse-down	Pulsar (sin soltar) un botón del ratón	Todos los elementos
onmouse-move	Mover el ratón	Todos los elementos
onmouseout	El ratón *"sale"* del elemento (pasa por encima de otro elemento)	Todos los elementos
onmouseover	El ratón *"entra"* en el elemento (pasa por encima del elemento)	Todos los elementos
onmouseup	Soltar el botón que estaba pulsado en el ratón	Todos los elementos
onreset	Inicializar el formulario (borrar todos sus datos)	<form>
onresize	Se ha modificado el tamaño de la ventana del navegador	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página (por ejemplo al cerrar el navegador)	<body>

Figura 2.74. Tabla resumen de eventos básicos.

Los eventos más utilizados en las aplicaciones web tradicionales son:

- `onload` para esperar a que se cargue la página por completo,
- Los eventos `onclick, onmouseover, onmouseout` para controlar el ratón
- `onsubmit` para controlar el envío de los formularios.

Otros eventos de la tabla anterior `(onkeydown, onclick, onsubmit, onkeypress, onreset)` nos van a permitir evitar la "acción por defecto" de ese evento. Más adelante se mostrará en detalle este comportamiento, que puede resultar muy útil en algunas técnicas de programación.

Se puede producir una sucesión de eventos cuando se realizan las acciones típicas en una página web. Por ejemplo, al pulsar sobre un botón de tipo <input type="submit"> se van a desencadenar los eventos onmousedown, onclick, onmouseup y onsubmit de forma consecutiva.

Manejadores de eventos

Un evento de JavaScript por sí mismo carece de utilidad. Se deben asociar funciones o código JavaScript a cada evento, para que estos resulten útiles. De esta manera, cuando se produzca un evento se ejecutará el código indicado, por lo que la aplicación responderá ante cualquier evento que se produzca durante su ejecución.

Los "manejador de eventos" son las funciones o código JavaScript que se definen para cada evento y al ser JavaScript un lenguaje muy flexible, hay varias formas de indicar los manejadores:

- Manejadores como atributos de los elementos XHTML

  Es el método más sencillo y menos profesional de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento. Lo que se hace es incluirlo en un atributo del propio elemento XHTML. En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pinche con el ratón sobre un botón:

```
<html xmlns="http://www.w3.org/1999/xhtml">

<head>

  <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />

  <title>UD3 Eventos</title>

</head>

<body>

    <input type="button" value="Pinchame y verás"
onclick="alert('Gracias por pinchar');" />

</body>

</html>
```

  En este método, hemos definido atributos XHTML con el mismo nombre que los eventos que se van a manejar. Solo vamos a controlar el evento de pinchar con el ratón, cuyo nombre es `onclick`. Así, el elemento XHTML para el que se quiere definir este evento, debe incluir un atributo llamado `onclick`.

  Los atributos contendrán una cadena de texto con todas las instrucciones JavaScript que se ejecutarán al producirse el evento. En el ejemplo anterior, vemos que es muy sencillo (`alert('Gracias por pinchar');"`), ya que únicamente queremos mostrar un mensaje.

Veamos otro ejemplo, en el cual, cuando el usuario pinche sobre el elemento se mostrará un mensaje y otro diferente cuando el usuario pase el ratón por encima del elemento:

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>UD3 Eventos</title>
</head>
<body>
    <div onclick="alert('Has pinchado con el ratón');"
onmouseover="alert('Acabas de pasar el ratón por encima');">
```

Puedes pinchar sobre este elemento o simplemente pasar el ratón por encima:

```
    </div>
</body>
</html>
```

Este otro ejemplo incluye una de las instrucciones más utilizadas en las aplicaciones JavaScript más antiguas:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>UD3 Eventos</title>
</head>
<body onload="alert('La página se ha cargado completamente');">
  ...
</body>
</html>
```

Una vez que se haya cargado completamente la pagina, se mostrará el mensaje, es decir, después de que se haya descargado su código HTML, sus imágenes y cualquier otro objeto incluido en la página.

NOTA

El evento **onload** es uno de los más utilizados ya que, como se vio en el capítulo de DOM, las funciones que permiten acceder y manipular los nodos del árbol DOM solamente están disponibles cuando la página se ha cargado completamente

Manejadores de eventos y variable this

JavaScript define una variable especial llamada this que se crea automáticamente y que se emplea en algunas técnicas avanzadas de programación. En los eventos, la usaremos para referirnos al elemento XHTML que ha provocado el evento.

Veamos un ejemplo de su uso. Por ejemplo, cuando el usuario pasa el ratón por encima del , el color del borde se mostrará de color negro y cuando salga del , se volverá a mostrar el borde con el color gris claro original.

Este seria el elemento original:

```
  <div id="contenidos" style="width: 150px; height: 60px; border:
thin solid silver">

    Sección de contenidos...

  </div>
```

Si no se utiliza la variable this, el código necesario para modificar el color de los bordes, sería el siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml">

<head>

    <meta http-equiv="Content-Type" content="text/html;
     charset=iso-8859-1" />

  <title>UD3 Eventos</title>

</head>

<body>

    <div id="contenidos" style="width: 150px; height:
       60px; border: thin solid silver"

   onmouseover="document.
getElementById('contenidos').style.borderColor='black';"

   onmouseout="document.getElementById('contenidos').
style.borderColor='silver';">

    Sección de contenidos...

  </div>

</body>

</html>
```

Como podéis observar, el código anterior es demasiado largo y muy fácil cometer un error en su composición.

NOTA

Dentro del código de un evento, JavaScript crea automáticamente la variable *this,* que hace referencia al elemento XHTML que ha provocado el evento.

De esta manera, el ejemplo anterior se podría escribir de la siguiente manera:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=iso-8859-1" />
  <title>UD3 Eventos</title>
</head>
<body>
    <div id="contenidos" style="width: 150px; height: 60px;
      border: thin solid silver"
   onmouseover="this.style.borderColor='black';"
     onmouseout="this.style.borderColor='silver';">
    Sección de contenidos...
  </div>
</body>
</html>
```

Podemos ver, que el código anterior es mucho más corto, más fácil de leer y de escribir y su funcionamiento es el mismo, aunque se modificase el valor del atributo id del .

- Manejadores de eventos como funciones externas

Como ya hemos comentado antes, la forma menos aconsejable de definir los manejadores de eventos es hacerlo en los atributos XHTML, aunque sea el método más sencillo de realizar. Se ira complicando en exceso en cuanto se añadan algunas pocas instrucciones, por lo que recomiendo su uso únicamente para los casos más sencillos.

Los más aconsejable es agrupar todo el código JavaScript en una función externa y llamar a esta función desde el elemento XHTML, cuando se realicen aplicaciones complejas, como por ejemplo la validación de un formulario, es lo más aconsejable.

Siguiendo con el ejemplo anterior, el que mostraba un mensaje al pinchar sobre un botón:

```
<input type="button" value="Pinchame y
verás"onclick="alert('Gracias por pinchar');" />
```

Utilizando una función externa podemos transformar el código en lo siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html;
     charset=iso-8859-1" />
  <title>UD3 Eventos</title>
  <script type="text/javascript">
        function muestraMensaje() {
            alert('Gracias por pinchar');
        }
   </script>
</head>
<body>
    <input type="button" value="Pinchame y verás"
      onclick="muestraMensaje()" />
</body>
</html>
```

Lo que hacemos es extraer todas las instrucciones de JavaScript y agruparlas en una función externa. Una vez definida, en el atributo del elemento XHTML incluiremos el nombre de la función, indicando de esa manera cual va a ser la función que se ejecutará cuando se produzca el evento.

La llamada a esa función la realizaremos de la forma habitual, indicaremos su nombre, a continuación los paréntesis y de manera opcional, incluiremos todos los argumentos y parámetros necesarios para la función.

El principal inconveniente de este método es que en las funciones externas no se puede seguir utilizando la variable this y por tanto, es necesario pasar esta variable como parámetro a la función:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>UD3 Eventos</title>
    <script type="text/javascript">
        function resalta(elemento) {
      switch (elemento.style.borderColor) {
        case 'silver':
        case 'silver silver silver silver':
         case '#c0c0c0':
                    elemento.style.borderColor = 'black';
          break;
        case 'black':
        case 'black black black black':
         case '#000000':
          elemento.style.borderColor = 'silver';
          break;
     }
        }
  </script>
</head>
<body>
    <div style="width: 150px; height: 60px; border: thin solid
silver" onmouseover="resalta(this)" onmouseout="resalta(this)">
   ección de contenidos...
  </div>
</body>
</html>
```

En el ejemplo, le pasamos como parámetro this, que dentro de la función se denominará elemento. La dificultad de este ejemplo reside en la forma en la que cada navegador almacena el valor de la propiedad borderColor.

Mientras que Firefox almacena el valor black (solo en caso de que los cuatro bordes coincidan en color), Internet Explorer lo almacena como black black black black y Opera almacena su representación hexadecimal #000000.

Manejadores de eventos semánticos

Hasta ahora, los métodos usados para añadir manejadores de eventos (como atributos XHTML y como funciones externas) "ensucian" el código XHTML de la página.

Como programador de páginas web, una de las buenas prácticas básicas en el diseño de páginas y aplicaciones web es la separación de los contenidos (XHTML) y su aspecto o presentación (CSS).

Y siempre que sea posible, debemos separar el comportamiento o programación (JavaScript) de los contenidos (XHTML).

Juntar los elementos XHTML con el código JavaScript solo contribuirá

- a complicar el código fuente de la página
- a dificultar la modificación y mantenimiento de la página
- a reducir la semántica del documento final producido.

Por suerte, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica es una evolución del método de las funciones externas, y lo que haremos es utilizar las propiedades DOM de los elementos XHTML para asignar todas las funciones externas que actúan de manejadores de eventos. Así, el siguiente ejemplo:

```
<body>
    <input id="pinchable" type="button" value="Pinchame y verás"
onclick="alert('Gracias por pinchar');" />
</body>
```

Lo transformaremos en:

```
<script type="text/javascript">
  // Función externa
  function muestraMensaje() {
        alert('Gracias por pinchar');
    }
  // Asignar la función externa al elemento
  document.getElementById("pinchable").onclick = muestraMensaje;
</script>
```

Y dentro del body de la página:

```
<body>
  // Elemento XHTML
  <input id="pinchable" type="button" value="Pinchame y verás" />
</body>
```

Explicamos a continuación en que consiste la técnica de los manejadores semánticos:

- Lo primero asignaremos un identificador único al elemento XHTML mediante el atributo id.
- Después crearemos una función de JavaScript que será la encargada de manejar el evento.
- Y por ultimo, asignaremos la función externa al evento correspondiente en el elemento deseado.

  El último paso es la clave de esta técnica. En primer lugar, se obtiene el elemento al que se desea asociar la función externa:

  ```
  document.getElementById("pinchable")
  ```

  A continuación, utilizamos una propiedad del elemento que tendrá el mismo nombre que el evento que queremos manejar. En este caso, la propiedad es onclick:

  ```
  document.getElementById("pinchable").onclick = …
  ```

  Y por último, asignaremos la función externa mediante su nombre sin paréntesis. Lo más importante y lo que produce la mayoría de los errores, debemos indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

  ```
  document.getElementById("pinchable").onclick = muestraMensaje;
  ```

  Si añadiésemos los paréntesis después del nombre de la función, en realidad estaríamos ejecutando la función y guardando el valor devuelto por la misma en la propiedad onclick de elemento.

  Lo mejor de usar este método es que el código XHTML resultante es muy "limpio", ya que este no se va a mezclar nunca con el código JavaScript. Además, sí que se podremos utilizar la variable this para referirse al elemento que provocará el evento dentro de las funciones externas asignadas.

  Lo que tenemos que tener en cuenta, es que la página se deberá cargar completamente antes de que se puedan utilizar dichas funciones DOM que asignaran los manejadores a los elementos XHTML. Lo más sencillo para asegurarnos que cierto código se ejecutará después de que la página se cargue por completo, será utilizando el evento onload:

```
window.onload = function () {
        document.getElementById("pinchable").onclick =
muestraMensaje;
    }
```

Mediante la técnica anterior introduciomos el concepto de funciones anónimas, que aunque no se va a estudiar, nos permitirán crear un código compacto y muy sencillo. Esta será la forma, con la que nos aseguraremos que un código JavaScript se ejecutará después de que la página se haya cargado completamente, y sólo será necesario incluir esas instrucciones entre los símbolos { y }:

```
window.onload = function () {
        …
    }
```

En el siguiente ejemplo, vamos a añadir eventos a los elementos de tipo input=text de un formulario complejo:

```
<script type="text/javascript">
    function resalta() {
  // Código JavaScript
}

    window.onload = function () {
  var formulario = document.getElementById("formulario");
  var camposInput = formulario getElementsByTagName("input");
  for (var i = 0; i < camposInput.length; i++) {
    if (camposInput[i].type == "text") {
            camposInput[i].onclick = resalta;
        }
      }
    }
</script>
```

2.3.2.2. Modelo de eventos estándar

Las versiones más avanzadas del estándar DOM (DOM nivel 2) definen un modelo de eventos mucho más poderoso que el original y completamente nuevo. Todos los navegadores modernos lo incluyen, salvo Internet Explorer, que posee su modelo propio.

2.3.2.3. Modelo de eventos de Internet Explorer

Debes saber, que Internet Explorer utiliza su propio modelo de eventos, muy similar, pero incompatible con el modelo estándar. La primera vez que se utilizó fue en Internet Explorer 4 y Microsoft decidió seguir utilizándolo en el resto de versiones, y eso que la empresa había participado en la creación del estándar de DOM, en el que se definió el modelo de eventos estándar.

2.3.3. Obteniendo información del evento (objeto event)

Para ir un poco más allá en el tema, debemos saber que los manejadores de eventos pueden requerir de información adicional para procesar sus tareas. Por ejemplo, si estamos trabajando con una función, que, procesa el evento onclick, quizás necesite conocer la posición en la que esta el ratón en el momento de pinchar el botón.

Lo más habitual será conocer información adicional es en el caso de los eventos asociados al teclado. Es muy importante conocer la tecla que se ha pulsado, por ejemplo para diferenciar las teclas normales de las teclas especiales (ENTER, tabulador, Alt, Ctrl., etc.).

JavaScript nos permitirá obtener información sobre el ratón y el teclado mediante un objeto especial llamado event. Al igual que pasaba con los manejadores de eventos, los diferentes navegadores presentan diferencias muy notables a la hora de tratar la información sobre los eventos.

La principal diferencia reside en la forma en la que se obtiene el objeto event:

- Internet Explorer lo considera como parte del objeto window.
- El resto de navegadores lo consideran como el único argumento que tienen las funciones manejadoras de eventos.

Aunque al principio nos pueda parecer un comportamiento muy extraño, excepto Internet Explorer, todos los demás navegadores modernos crean mágicamente y de forma automática un argumento que se pasa a la función manejadora, por lo que no será necesario incluirlo en la llamada a la función manejadora.

Así, para utilizar este “argumento mágico”, sólo será necesario asignarle un nombre, ya que los navegadores lo crearán automáticamente.

Veamos con un ejemplo la forma de obtenerlo en los diferentes navegadores. En Internet Explorer, el objeto event se obtiene directamente mediante:

```
var evento = window.event;
```

En el resto de navegadores, el objeto event se obtendrá mágicamente a partir del argumento que el navegador crea automáticamente:

```
function manejadorEventos(elEvento) {
    var evento = elEvento;
}
```

Debemos tener en cuenta que si se quiere programar una aplicación que funcione correctamente en todos los navegadores, deberemos obtener el objeto event de forma la correcta para cada navegador. El siguiente código muestra la forma correcta de obtener el objeto event en cualquier navegador:

```
function manejadorEventos(elEvento) {
        var evento = elEvento || window.event;
    }
```

Una vez obtenido el objeto event, ya podremos acceder a toda la información relacionada con el evento, que dependerá del tipo de evento producido.

2.3.3.1. Información sobre el evento

Disponemos de la propiedad type para que nos indique el tipo de evento producido, lo que nos será muy útil cuando una misma función sea utilizada para manejar varios eventos:

```
var tipo = evento.type;
```

Esta propiedad nos devolverá el tipo de evento producido, que será igual al nombre del evento pero sin el prefijo on.

Podemos rehacer de forma más sencilla el ejemplo del punto anterior en el que resaltabamos una sección de contenidos al pasar el ratón por encima:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
  <title>Ejemplo de getElementById()</title>
  <script type="text/javascript">
    function resalta(elEvento) {
      var evento = elEvento || window.event;
       switch (evento.type) {
        case 'mouseover':
          this.style.borderColor = 'black';
         break;
        case 'mouseout':
         this.style.borderColor = 'silver';
         break;
          }
```

```
        }
        window.onload = function () {
            document.getElementById("seccion").onmouseover = resalta;
            document.getElementById("seccion").onmouseout = resalta;
        }
    </script>
</head>
<body>
    <div id="seccion" style="width: 150px; height: 60px; border:
     thin solid silver">
    Sección de contenidos...
  </div>
</body>
</html>
```

2.3.3.2. Información sobre los eventos de teclado

Estos van a ser los eventos mas difíciles de manejar dadas las incompatibles entre diferentes navegadores. Existen muchas diferencias entre los navegadores, los teclados y los SO, debidos a los diferentes idiomas.

Lo primero debemos saber que hay tres eventos diferentes para las pulsaciones de las teclas:

- Onkeyup.
- Onkeypress.
- onkeydown .

Y después, que existen dos tipos de teclas:

- teclas normales (como letras, números y símbolos normales)
- teclas especiales (como ENTER, Alt, Shift, etc.)

Cuando un usuario pulsa una tecla normal, se producen tres eventos seguidos y en este orden:

- *onkeydown:* este evento se corresponde con el hecho de pulsar una tecla y no soltarla.
- *onkeypress:* es la propia pulsación de la tecla
- *onkeyup:* y este evento hace referencia al hecho de soltar una tecla que estaba pulsada.

La manera más sencilla de obtener que tecla se ha pulsado será con el evento onkeypress, ya que onkeydown y onkeyup nos proporcionan una información de los eventos que se puede considerar como más técnica, ya que devuelven el código interno de cada tecla y no el carácter que se ha pulsado.

Veamos a continuación una lista con todas las propiedades diferentes de todos los eventos de teclado especificando sus diferentes usos dependiendo del navegador:

- Evento keydown:
  - Mismo comportamiento en todos los navegadores:
    - Propiedad `keyCode`: código interno de la tecla
    - Propiedad `charCode:` no definido
- Evento keypress:
  - Internet Explorer:
    - Propiedad `keyCode:` el código del carácter de la tecla que se ha pulsado
    - Propiedad `charCode:` no definido
  - Resto de navegadores:
    - Propiedad `keyCode:` para las teclas normales, no definido. Para las teclas especiales, el código interno de la tecla.
    - Propiedad `charCode:` para las teclas normales, el código del carácter de la tecla que se ha pulsado. Para las teclas especiales, 0.
- Evento keyup:
  - Mismo comportamiento en todos los navegadores:
    - Propiedad `keyCode:` código interno de la tecla
    - Propiedad `charCode:` no definido

Utilizaremos la función String.fromCharCode() para convertir el código de un carácter (que no es el código interno) al carácter que representa la tecla que se ha pulsado.

Veamos ahora un ejemplo que muestra toda la información sobre los tres eventos de teclado:

```
<html xmlns="http://www.w3.org/1999/xhtml">

<head>

  <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />

  <title>Ejemplo de getElementById()</title>

    <script type="text/javascript">
```

```
    window.onload = function () {
            document.onkeyup = muestraInformacion;
            document.onkeypress = muestraInformacion;
            document.onkeydown = muestraInformacion;
        }
     function muestraInformacion(elEvento) {
       var evento = window.event || elEvento;
      var mensaje = "Tipo de evento: " + evento.type + "<br>" +
         "Propiedad keyCode: " + evento.keyCode + "<br>" +
        "Propiedad charCode: " + evento.charCode + "<br>" +
         "Carácter pulsado: " + String.fromCharCode(evento.charCode)
      info.innerHTML += "<br>------------------<br>" + mensaje
    }
  </script>
</head>
<body>
    <div id="info">
    </div>
</body>
</html>
```

Si ejecutamos el script y pulsamos por ejemplo la tecla “a”, veremos en pantalla el resultado en la primera columna. En la segunda, veremos los resultados de los eventos cuando pulsamos la tecla A (habiendo activado previamente las mayúsculas):

Tecla pulsada “a”	Tecla pulsada “A”
----------	----------
Tipo de evento: keydown	Tipo de evento: keydown
Propiedad keyCode: 65	Propiedad keyCode: 65
Propiedad charCode: 0	Propiedad charCode: 0
Carácter pulsado:	Carácter pulsado:
----------	----------
Tipo de evento: keypress	Tipo de evento: keypress
Propiedad keyCode: 97	Propiedad keyCode: 65
Propiedad charCode: 97	Propiedad charCode: 65
Carácter pulsado: a	Carácter pulsado: A
----------	----------
Tipo de evento: keyup	Tipo de evento: keyup
Propiedad keyCode: 65	Propiedad keyCode: 65
Propiedad charCode: 0	Propiedad charCode: 0
Carácter pulsado:	Carácter pulsado:

Figura 2.75. Diferencias en el navegador al pulsar “a” y “A”.

Si nos fijamos en los resultados, en los eventos keydown y keyup, la propiedad keyCode valdrá lo mismo en los dos casos. Esto sucede por que keyCode almacenará el código interno de la tecla, por lo que si se pulsa la misma tecla, se obtendrá el mismo código, independientemente de que una misma tecla puede producir caracteres diferentes (mayúsculas y minúsculas).

En cambio, en el evento keypress, el valor de charCode si que varía, ya que el carácter a, no es el mismo que el carácter A. En este caso, el valor de charCode coincide con el código ASCII del carácter pulsado.

Siguiendo en el navegador Chrome, vamos a hacer la prueba con una tecla especial, como por ejemplo el tabulador, y la vamos a realizar también en el Iternet Explorer, para ver la diferencia entre ambos:

Tecla Tabulador en Chrome	Tecla Tabulador en Internet Explorer
----------	----------
Tipo de evento: keydown	Tipo de evento: keydown
Propiedad keyCode: 9	Propiedad keyCode: 9
Propiedad charCode: 0	Propiedad charCode: undefined
Carácter pulsado:	Carácter pulsado:

Figura 2.76. Diferencias entre navegadores.

Podemos observar, que las teclas especiales no disponen de la propiedad charCode, por que solamente se guarda el código interno de la tecla pulsada en la propiedad keyCode, en este caso el código 9. No obstante, dependiendo del teclado utilizado para pulsar las teclas y dependiendo de la disposición de las teclas en función del idioma del teclado, estos códigos podrían variar.

Por último, las propiedades *altKey, ctrlKey y shiftKey* almacenarán un valor booleano que indicará si alguna de esas teclas estaba pulsada al producirse el evento del teclado. Aunque parezca mentira, estas propiedades funcionan de la misma forma en todos los navegadores:

```
if (evento.altKey) {
        alert('Estaba pulsada la tecla ALT');
    }
```

También vamos a ver el caso en el que se pulsa la tecla Shift y sin soltarla, pulsamos sobre la tecla que contiene el número 2 (nos referimos a la tecla que se encuentra en la parte superior del teclado).

Tanto Internet Explorer como Chrome muestran la misma secuencia de eventos:

Tecla “Shift” + “2” en Chrome	Tecla “Shift” + “2” en Internet Explorer
----------	----------
Tipo de evento: keydown	Tipo de evento: keydown
Propiedad keyCode: 16	Propiedad keyCode: 16
Propiedad charCode: 0	Propiedad charCode: undefined
Carácter pulsado:	Carácter pulsado:
----------	----------
Tipo de evento: keydown	Tipo de evento: keydown
Propiedad keyCode: 50	Propiedad keyCode: 50
Propiedad charCode: 0	Propiedad charCode: undefined
Carácter pulsado:	Carácter pulsado:
----------	----------
Tipo de evento: keypress	Tipo de evento: keypress
Propiedad keyCode: 34	Propiedad keyCode: 34
Propiedad charCode: 34	Propiedad charCode: undefined
Carácter pulsado: “	Carácter pulsado:
----------	----------
Tipo de evento: keyup	Tipo de evento: keyup
Propiedad keyCode: 50	Propiedad keyCode: 50
Propiedad charCode: 0	Propiedad charCode: undefined
Carácter pulsado:	Carácter pulsado:
----------	----------
Tipo de evento: keyup	Tipo de evento: keyup
Propiedad keyCode: 16	Propiedad keyCode: 16
Propiedad charCode: 0	Propiedad charCode: undefined
Carácter pulsado:	Carácter pulsado:

Figura 2.77. Diferencias entre navegadores al pulsar "Shift" + "2".

Recordar que el único evento que permite obtener el carácter realmente pulsado va a ser keypress, ya que al pulsar sobre la tecla 2 habiendo pulsado la tecla Shift previamente, se obtiene el carácter “ comillas), que es precisamente el que muestra el evento keypress.

Con el siguiente código de JavaScript podrás obtener de forma correcta en cualquier navegador el carácter correspondiente a la tecla pulsada:

```
<html xmlns=”http://www.w3.org/1999/xhtml”>

<head>

    <meta http-equiv=”Content-Type” content=”text/html;
charset=iso-8859-1” />

    <title>Ejemplo de getElementById()</title>

    <script type=”text/javascript”>
```

```
        window.onload = function () {
            document.onkeypress = manejador;
        }
        function manejador(elEvento) {
            var evento = elEvento || window.event;
            var caracter = evento.charCode || evento.keyCode;
            //para mostrarlo mediante un alert descomentar la siguiente linea
           //alert("El carácter pulsado es: " + String.
            fromCharCode(caracter));
            //la siguiente linea mostrará el mensaje en el mismo navegador
            info.innerHTML += "Carácter pulsado: " + String.
           fromCharCode(caracter) + "<br>";
        }
    </script>
</head>
<body>
    <div id="info">
    </div>
</body>
</html>
```

2.3.3.3. Información sobre los eventos de ratón

En este punto vamos a tratar los eventos del ratón. Veremos como mostrar las coordenadas de la posición del puntero del ratón. Debemos saber que el origen de las coordenadas siempre se encuentra en la esquina superior izquierda, aunque este dato puede variar.

De esta manera, podemos obtener la posición del ratón respecto a:

- La pantalla del ordenador.
- La ventana del navegador.
- La propia página HTML (que se utiliza cuando el usuario ha hecho scroll sobre la página).

Las más sencillas de obtener mediante las propiedades clientX y clientY, son las que se refieren a la posición del puntero respecto de la ventana del navegador:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
    <meta http-equiv="Content-Type" content="text/html;
     charset=iso-8859-1" />
    <title>Ejemplo de getElementById()</title>
    <script type="text/javascript">
        window.onload = function () {
            document.onclick = muestraInformacion;
        }

        function muestraInformacion(elEvento) {
            var evento = elEvento || window.event;
            var coordenadaX = evento.clientX;
            var coordenadaY = evento.clientY;
            alert("Has pulsado el ratón en la posición: " +
     coordenadaX + ", " + coordenadaY);
        }
    </script>
</head>
<body>
    <div id="info">
    </div>
</body>
</html>
```

Si ejecutamos el script, y pulsamos con el ratón en la ventana del navegador, obtendremos la información de las coordenadas pulsadas:

Si queremos obtener las coordenadas respecto de la pantalla completa del ordenador del usuario, lo haremos mediante las propiedades screenX y screenY, y debemos cambiar esas dos líneas y escribir las siguientes:

```
            var coordenada = evento.screenX;
            var coordenadaY = evento.screenY;
```

En otros casos, necesitaremos obtener las coordenadas que corresponden a la posición del ratón respecto del origen de la página. Debemos saber que estas coordenadas no siempre van a coincidir con las coordenadas respecto del origen de la ventana del navegador, por que podría ser que el usuario hubiese hecho scroll sobre la página web.

Respecto al tema de cómo obtenerlas dependiendo del navegador que estemos usando, Internet Explorer no nos las va a proporcionar de forma directa, mientras que el resto de navegadores sí que lo harán. Por lo tanto, tendremos que detectar antes si el navegador es de tipo Internet Explorer y si es así, realizar un cálculo sencillo:

```
// Detectar si el navegador es Internet Explorer
var ie = navigator.userAgent.toLowerCase().indexOf('msie') != -1;


if (ie) {
    coordenadaX = evento.clientX + document.body.scrollLeft;
    coordenadaY = evento.clientY + document.body.scrollTop;
}
else {
    coordenadaX = evento.pageX;
    coordenadaY = evento.pageY;
}
```

Si el navegador en el que se ejecuta el script es de tipo Internet Explorer (cualquier versión) la variable ie valdrá true y false en caso contrario. El resto de navegadores, las coordenadas respecto del origen de la página se obtendrán mediante las propiedades pageX y pageY. En el caso de Internet Explorer, se obtienen sumando la posición respecto de la ventana del navegador (clientX, clientY) y el desplazamiento que ha sufrido la página (document.body.scrollLeft, document.body.scrollTop).

RESUMEN

- En esta unidad hemos aprendido el funcionamiento de DOM. Su estructura, como funciona el árbol de nodos y los diversos tipos de nodos existentes.
- Hemos visto como acceder a los nodos mediante las funciones getElementsByTagName(), getElementsByName() y getElementsById(). Ademas, hemos visto como acceder a los atributos de XHTML.
- Conocemos las diferentes propiedades y métodos de los nodos que proporciona JavaScript. Y hemos aprendido a crear, insertar clonar y eliminar nodos de una pagina XHTML.
- Ya sabemos que son los eventos, como funcionan y la forma de manejarlos en las páginas web. Hemos conocido los tres modelos existentes y las diferentes formas de manipularlos en el modelo básico de eventos.
- Y por ultimo hemos visto como obtener la información del Objeto Event, los tipos de eventos que existen y como capturar los eventos del teclado y del ratón.