DWEC
Unidad 3
D.O.M
y
Gestión de eventos

INIDICE

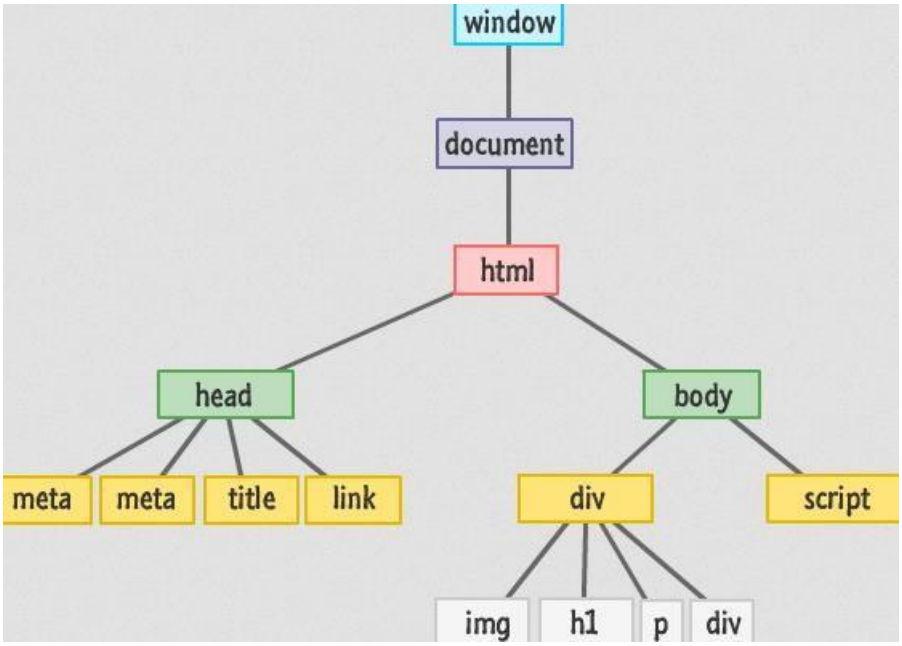
1. Introducción al DOM

El Modelo de Objetos del Documento (DOM) es una interfaz de programación de aplicaciones (API) para documentos HTML y XML. Define la estructura lógica de los documentos y el modo en que se accede y manipula un documento.

En la especificación del DOM, el término "documento" se utiliza en un sentido amplio. XML se utiliza cada vez más como un medio para representar muchas clases diferentes de información que puede ser almacenada en sistemas diversos, y mucha de esta información se vería, en términos tradicionales, más como datos que como documentos. Sin embargo, XML presenta estos datos como documentos, y se puede usar el DOM para manipular estos datos.

Con el Modelo de Objetos del Documento los programadores pueden construir documentos, navegar por su estructura, y añadir, modificar o eliminar elementos y contenido. Se puede acceder a cualquier cosa que se encuentre en un documento HTML o XML, y se puede modificar, eliminar o añadir usando el Modelo de Objetos del Documento, salvo algunas excepciones. En particular, aún no se han especificado las interfaces DOM para los subconjuntos internos y externos de XML.

Siendo una especificación del W3C, uno de los objetivos importantes del Modelo de Objetos del Documento es proporcionar un interfaz estándar de programación que pueda utilizarse en una amplia variedad de entornos y aplicaciones.

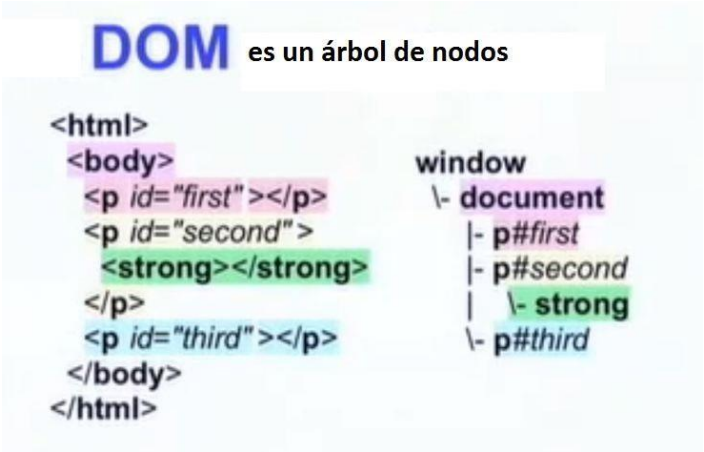


Ejemplo de DOM

2. El modelo de objetos de documento (DOM)

El DOM fue utilizado por primera vez con el navegador Netscape Navigator (versión 2.0 del navegador). A esta primera versión de DOM, se la considera modelo básico o DOM nivel 0. El primer navegador de Microsoft que comenzó a utilizar el modelo básico fue Internet Explorer 3.0.

En la versión 3.0 de Netscape y la 4.0 de Internet Explorer se comenzaron a utilizar rollovers. Rollover es un efecto que hace cambiar una imagen, por ejemplo, cuando pasamos el ratón por encima. A partir de aquí comenzaron a incluir en algunos navegadores la capacidad de detectar eventos de ratón y de teclado. Debido a las diferencias entre los navegadores W3C se emitió una especificación a finales de 1998 que se llamó DOM nivel 1. En esta especificación ya se consideraban las características y manipulación de todos los elementos existentes en los archivos HTML y también de XML. A finales del año 2000, W3C emitió la especificación DOM nivel 2. En esta especificación se incluyó el manejo de eventos en el navegador y la capacidad de interactuar con hojas de estilo CSS. En 2004 se emitió la especificación DOM nivel 3, la cual utiliza la definición de tipos de documento (DTD) y la validación de documentos.



El DOM es un árbol de nodos

2.1. Tipos de modelos DOM

Actualmente DOM se divide en tres partes diferentes según la W3C. A estas partes también se les llama niveles. A continuación, vamos a ver cuáles son estos tres niveles y en qué consiste cada uno de ellos:

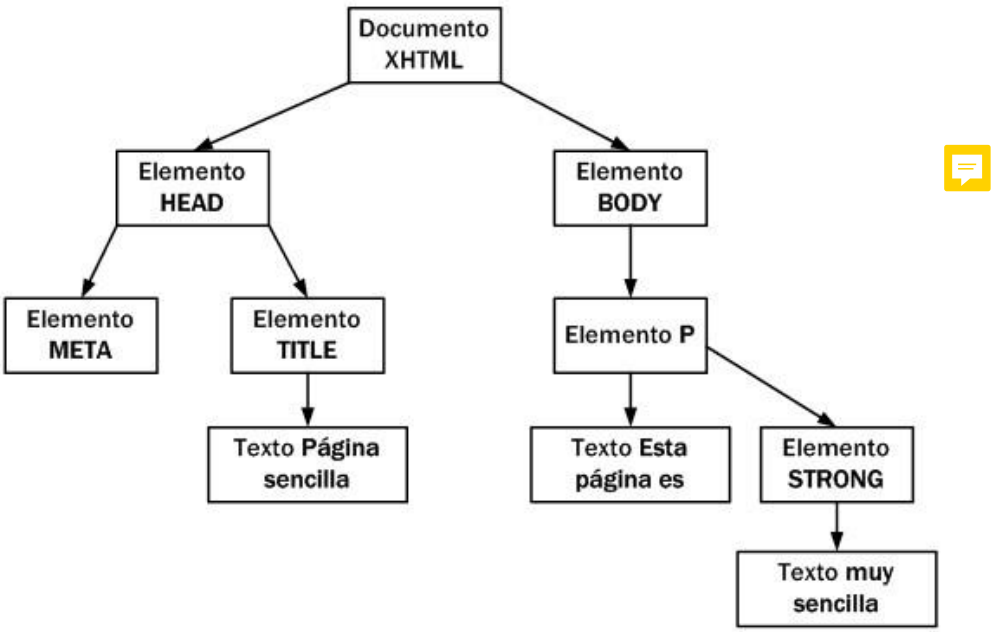
- **Núcleo del DOM**. Este es el modelo estándar para cualquier documento estructurado. En este modelo se especifican a nivel general las pautas para definir los objetos y propiedades de cualquier documento estructurado, así como los métodos para acceder a ellos.

- **XML DOM**. Este es el modelo estándar para los documentos XML. Este modelo define los objetos y propiedades de todos los elementos XML, así como los métodos para acceder a ellos.
- **HTML DOM**. Modelo estándar para los documentos HTML. Este es el modelo en el que nos vamos a centrar. El DOM HTML es un modelo de datos estándar para HTML. Una interfaz de programación estándar para HTML independiente de la plataforma y el lenguaje. Es un estándar de la W3C. El DOM HTML define los objetos y las propiedades de los elementos HTML. También define los métodos para acceder a ellos. **Por lo tanto, DOM HTML es un estándar sobre la forma de obtener, modificar, añadir o eliminar elementos HTML.**

2.2. Estructura del árbol DOM

El modelo DOM HTML se define a través de una estructura de árbol. Para poder utilizar las utilidades DOM, el navegador web transforma automáticamente todas las páginas web en una estructura de árbol. Las páginas web son una sucesión de caracteres, por lo que resultaría excesivamente complicado manipular los elementos si no fuera por esta conversión. La estructura de cada página web, se organiza de forma que se pueda acceder a los elementos a través de la estructura de árbol.

DOM transforma los documentos HTML en elementos. Estos elementos se llaman nodos. A su vez los nodos están interconectados y muestran el contenido de la página web y la relación entre nodos. Por lo tanto, al conjunto de nodos se le llama, árbol de nodos. A continuación mostramos un ejemplo de una estructura de nodos generada por DOM a partir de una página HTML.



Ejemplo de árbol DOM

Según vemos en la figura anterior, DOM crea una estructura por cada uno de los elementos de la página HTML. A estos elementos que se sitúan en el árbol se les llama nodos. Por lo tanto, el árbol de la figura anterior tendría once nodos.

En la estructura anterior el nodo <html> sería el nodo raíz, por tanto, todo está contenido dentro de él. La estructura en código HTML de la imagen sería la siguiente:

```
<html>
  <head>
  <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <title>Página sencilla</title>
  </head>
  <body>
    <p>Esta página es<strong> muy sencilla</strong> </p>
  </body> </html>
```

El navegador convierte cada página de la aplicación web en una estructura de árbol. Como podemos observar los datos de cada elemento se almacenan en un nodo aparte, que cuelga del nodo elemento.

El modelo DOM define la forma en que los objetos y elementos se relacionan entre sí, tanto en el navegador como en el documento. Cada objeto tiene un nombre único. Cuando existe más de un objeto de un tipo, estos objetos se organizan en un vector.

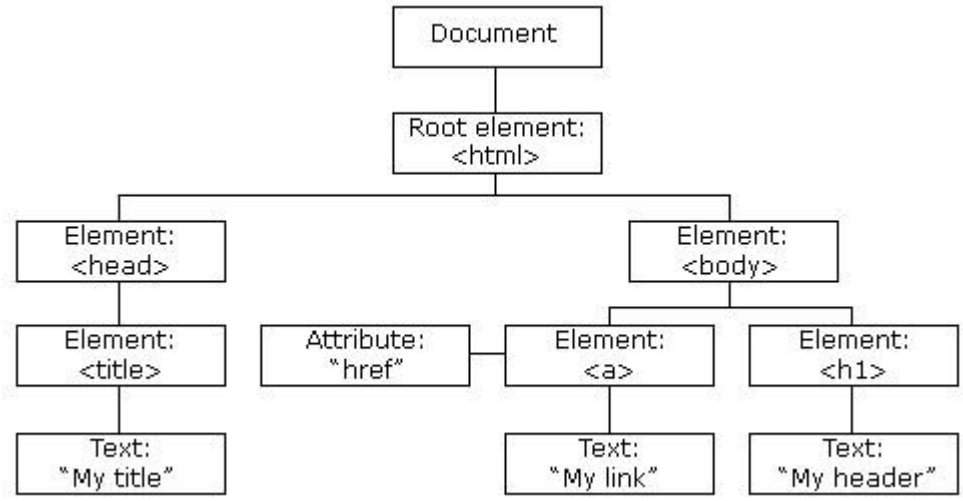
El primer objeto que nos encontramos es el objeto document, este objeto es la raíz del árbol de nodos. Por lo tanto, es un objeto nodo. Este nodo al ser el primero, no tiene nodo padre, tampoco tiene nodos a su mismo nivel. Solo tiene nodos hijo. De acuerdo con el DOM, todo lo que existe en un documento HTML es un nodo. Para ordenar la estructura del árbol, existen una serie de reglas que son de obligado cumplimiento:

- En el árbol de nodos, al nodo superior (document, visto anteriormente) se le llama **raíz**.
- Cada nodo, exceptuando el nodo raíz, tiene un padre.
- Un nodo puede tener cualquier número de hijos.
- Una hoja es un nodo sin hijos.
- Los nodos que comparten el mismo padre, son hermanos.

Estas normas siempre son respetadas para que la jerarquía tenga una integridad y poder asegurar el orden y acceso a los diferentes elementos.

3. Objetos del modelo. Propiedades y métodos

El Modelo de Objetos de Documento (DOM), está provisto de una serie de objetos que facilitan el manejo y acceso a los distintos elementos de un árbol DOM.



Árbol de objetos HTML DOM

A continuación vamos a definir cada uno de los tipos de nodos que existen en el DOM HTML.

Según la W3C, estos son los tipos de nodos más importantes:

- **Document**. Es el nodo raíz del documento HTML. Todos los elementos del árbol cuelgan de él.
- **DocumentType**. Este nodo indica la representación del DTD de la página. Un DTD es una definición de tipo de documento. Define la estructura y sintaxis de un documento XML. El DOCTYPE es el encargado de indicar el DocumentType.
- **Element**. Este nodo representa el contenido de una pareja de etiquetas de apertura y cierre (<etiqueta><… etiqueta>). También puede representar una etiqueta abreviada que se cierra a si misma (
). Este el único nodo que puede tener tantos nodos hijos como atributos.
- **Attr**. Este nodo representa el nombre del atributo o valor.
- **Text**. Este nodo almacena la información que es contenida en el tipo de nodo Element.
- **CDataSection**. Este nodo representa una secuencia de código del tipo < ! [CDATA [ ]] >. Este texto solo se analizado por un programa de análisis.
- **Coment**. Este nodo representa un comentario XML.

Viendo la relación que genera el estándar entre las partes de una aplicación web a nivel de código y los tipos nodos del árbol, queda estructurado el árbol para poder manipular de una forma fácil cada uno de los elementos, el punto siguiente vamos a definir cómo se accede al documento y a los tipos de nodos desde DOM.

3.1. La interfaz Node

Para poder manipular la información de los nodos, JavaScript crea un objeto, llamado Node. En este objeto se definen las propiedades y los métodos para procesar y manipular los documentos. El objeto Node define una serie de constantes que identifican los tipos de nodo. En la siguiente tabla vemos una relación entre las constantes y un número que se asigna a cada uno de los tipos de nodo. Asociación de constantes valor de tipo de nodo.

Tabla. Tipos de nodo DOM

Tipo de nodo=Valor
Node.ELEMENT_NODE = 1
Node.ATTRIBUTE_NODE = 2
Node.TEXT_NODE = 3
Node.CDATA_SECTION_NODE = 4
Node.ENTITY_REFERENCE_NODE = 5
Node.ENTITY_NODE = 6
Node.PROCESSING_INSTRUCTION_NODE = 7
Node.COMMENT_NODE = 8
Node.DOCUMENT_NODE = 9
Node.DOCUMENT_TYPE_NODE =10
Node.DOCUMENT_FRAGMENT_NODE =11
Node.NOTATION NODE = 12

Además de estas constantes el objeto Node proporciona una serie de propiedades y métodos para poder acceder a la jerarquía de elementos del árbol. Estos los mostramos en la tabla a continuación:

Tabla de propiedades del DOM

Propiedad	Valor	Descripción	IEWin	IEMac	Mozilla	Safari	Chrome
nodeName	String	Varía según tipo de nodo	Si	Si	Si	Si	Si
nodeValue	String	Varía según Tipo de nodo	Si	Si	Si	Si	Si
nodeType	Integer	Constante con cada tipo	Si	Si	Si	Si	Si
parentNode	Object	Referencia al siguiente contenedor más externo	Si	Si	Si	Si	Si

childNodes	Array	Todos los nodos hijos en orden	Si	Si	Si	Si	Si
firstChild	Object	Referencia el primer nodo hijo	Si	Si	Si	Si	Si
lastChild	Object	Referencia el último nodo hijo	Si	Si	Si	Si	Si
previousSibling	Object	Referencia el hermano anterior según su orden en el código fuente	Si	Si	Si	Si	Si
nextSibling	Object	Referencia el hermano siguiente según su orden en el código fuente	Si	Si	Si	Si	Si
attributes	NodeMap	Array de atributos de los nodos	Si	Algunos	Si	Si	Si
ownerDocument	Object	Objeto document	Si	Si	Si	Si	Si
namespaceURI	String	URI a la definición de namespace	Si	No	Si	Si	Si
Prefix	String	Prefijo del namespace	Si	No	Si	Si	Si
localName	String	Aplicable a los nodos afectados en el namespace	Si	No	Si	Si	Si

Métodos del DOM

Método	Descripción	IE	Mozilla	Chrome	Safari
appendChild(newChild)	Añade un hijo al final del nodo actual	Si	Si	Si	Si
cloneNode(deep)	Realiza una copia del nodo actual (opcionalmente con todos sus hijos)	Si	Si	Si	Si

hasChildNodes()	Determina si el nodo actual tiene o no hijos (valorboolean)	Si	Si	Si	Si
inserBefore(new,ref)	Inserta un nuevo hijo antes de otro nodo	Si	Si	Si	Si
removeChild(old)	Borra un hijo	Si	Si	Si	Si
replaceChild(new,old)	Reemplaza un hijo viejo con el nodo viejo	Si	Si	Si	Si
IsSupported(feature,versión)	Determina cuando el nodo soporta una característica especial	No	Si	Si	Si

4. Acceso al documento desde el código

Cuando el árbol de nodos DOM ha sido construido por el navegador de forma automática, y se ha cargado completamente, podemos acceder a cualquier nodo. Si existe más de un tipo de un elemento, estos se van almacenando en un vector. A continuación vamos a ver la estructura de acceso a través del árbol DOM en una página HTML. Supongamos que tenemos una pagina HTML con la siguiente estructura:

```
<html>
  <head>
    <title>Titulo DOM</title>
  </head>
  <body>
    <p>Parrafo DOM</p>
    <p>Parrafo DOM segundo</p>
    <p>Parrafo DOM tres</p>
  </body>
</html>
```

Uso de propiedades del DOM

Cuando la página ha sido cargada, podemos acceder a los elementos.

Según la estructura de DOM, el primer paso es recuperar el objeto que representa el elemento raíz de la página. En realidad, el objeto que se define como raíz es HTMLDocument. El objeto document es parte del BOM (Browser Object Model), pero se considera equivalente al document de DOM, por lo que document también hace referencia al nodo raíz las páginas HTML.

```
var obj_html=document.documentElement;
```

De esta forma cargamos en el objeto obj_html un objeto de tipo HTMLElement que representa al elemento html de nuestra estructura.

De este elemento derivan siempre según la estructura DOM <head> y <body>, al saber que son dos, podríamos acceder a ellos recuperando el primer y último hijo del nodo <html> utilizando dos de los métodos de la tabla anterior.

```
var obj_head = obj_html.firstChild;
var obj_body = obj_html.lastChild;
```

En el caso de que existan más de dos nodos y queramos acceder a ellos, podríamos hacerlo a través del índice, el modo de acceso es a través de un vector que se genera con los objetos que están al mismo nivel. La forma de acceder es la siguiente:

```
var obj_head=obj_html.childNodes[0];
var obj_body=obj_html.childNodes[1];
```

En el caso de que no sepamos el número de nodos hijo, podemos acceder a este valor utilizando la propiedad length sobre el método childNodes.

```
var numeroHijos=obj_html.childNodes.length;
```

Otra forma de acceder a un nodo en el árbol, es a través de su hijo. Para recuperar el elemento deberíamos utilizar el método parentNode de la siguiente manera:

```
var obj_html=obj_body.parentNode;
```

Como dijimos en uno de los puntos anteriores, el nombre de los element coincide con el nombre de su etiqueta.

Para acceder a los elementos, DOM permite acceder a través del nombre del elemento. Esto asigna el elemento al objeto que pretendamos. En el código que mostramos a continuación, asignamos el elemento body (llamado en la página HTML, <body>...</body>), al objeto obj_body.

```
var obj_html= document.body;
```

4.1. Acceso a los tipos de nodo

Los objetos del modelo, se diferencian por su tipo. Existen distintos tipos de nodo. Vimos en la primera tabla, la relación entre la constante relacionada con el tipo de nodo y el número que se asigna al tipo. La forma de acceder al tipo de nodo, es a través de la

propiedad nodeType. A través de la interfaz Node, JavaScript accede al tipo de nodo. Por lo tanto, la forma de acceder a un tipo de nodo es la siguiente:

```
obj_tipo_document=document.nodeType;    //9
obj_tipo_elemento=document.documentElement.nodeType;    //    1
```

Las llamadas a los tipos de nodo anteriores, devuelven en el primer caso, "9" el valor asociado en la constante Node.DOCUMENT_NODE, asociado en la Tabla. En el segundo caso "1", valor asociado en la constante Node. ELEMENT_NODE según la Tabla de las constantes y los números asociados a los tipos. El uso de los tipos puede ser accedido a través de las constantes, que vienen definidas por defecto en la mayoría de los navegadores. Por lo tanto, podemos usar sentencias del tipo:

```
if (document.nodeType==Node.DOCUMENT_NODE)
{    alert ("Verdadero")   }    //    true
if(document.documentElement.nodeType==Node.ELEMENT_NODE)
{
alert("Verdadero")
} // true
```

Las condiciones devolverán verdadero, puesto que las constantes definidas en la Tabla de constantes, tienen los valores por defecto. En el caso de tener un navegador que no tenga las constantes definidas por defecto, habría que definirlas de forma explícita. Esto ocurre con la versión 7 y anteriores del navegador Internet Explorer.

4.2. Acceso directo a los nodos

Los métodos que hemos visto anteriormente, nos permiten acceder a los nodos a través de la jerarquía del árbol. Esta forma de acceso hace necesario acceder al nodo raíz de la página para llegar a cualquier otro. Al acceder a un nodo en una página HTML real, el árbol tiene infinidad de nodos. Acceder a estos nodos resulta una tarea tediosa. En el caso de que modifiquemos la estructura del árbol añadiendo, modificando o quitando nodos, podemos ocasionar problemas en el acceso a los nodos. Por esta razón DOM añade una serie de métodos, que nos permiten acceder de forma directa a los nodos. Los métodos son **getElementByTagName**(), **getElementsByName**() y **getElementById**(). Los nombres son largos, pero tiene la ventaja de que el nombre del método es autodescriptivo sobre su función. A continuación vamos a describir en qué consiste cada uno y cuál es la forma de invocarlos en el código.

El método **getElementByTagName** () recupera los elementos de la página HTML de la etiqueta que hayamos pasado como parámetro. Un ejemplo de cómo devuelve las etiquetas div es:

```
var divs = document.getElementsByTagName("div");
```

La función devuelve un vector con los nodos de tipo div. En realidad el vector es una lista de nodos (nodeList). La forma de acceder a los div, es:

```
var primerDiv = divs[0];
var segundDiv = divs[1];
```

Podemos recuperar todos los div existentes en la página con una estructura repetitiva como es el for. En el caso anterior la forma de recuperar los div de la página sería:

```
var divs=document.getElementsByTagName("div");
for (var i=0; i< div.length;i++)
{   var div=divs[i];
      ...
}
```

Esta función también la podemos aplicar para recuperar otra estructura dentro de una de un tipo ya recuperado. Siguiendo con el ejemplo anterior, mostramos cómo recuperar todos los enlaces del primer div del nodeList.

```
var    divs=document.getElementsByTagName("div");
var primerdiv=div[0];
var enlaces=primerdiv.getElementsByTagName("a") ;
```

En el código anterior recuperamos todos los enlaces contenidos en el primer div de la página.

El método getElementByName() recupera todos los elementos de la página HTML en los que el atributo name coincide con el parámetro pasado a través de la función:

```
var divPrimero=document.getElementsByName("primero");
```

```
<div name="primero">... </div>
<div name="segundo">... </div>
<div> … </div>
```

En el código anterior recuperamos el div que tiene por name primero. Habitualmente el name es único para cada elemento. En el caso de que existan varios elementos con el mismo name, recuperaríamos todos. En el caso de los input de tipo radiobutton, todos los elementos relacionados tienen el mismo name, en este caso la función recupera la colección de elementos. Esta función podríamos utilizarla para aplicar algo a un grupo de elementos que tengan el mismo name.

El método getElementByld () recupera el elemento HTML cuyo id coincida con el pasado a través de la función. Esta función accede directamente al nodo a través del id, que se le pasa a la función. Como el id de cada elemento tiene que ser único, es la función más utilizada para leer y modificar sus propiedades.

```
var divPrimero=document.getElementsById("primero");
 <div id="primero">... </div>
<div id="segundo">... </div>
<div> … </div>
```

4.3. Acceso a los atributos de un nodo tipo element

En los puntos anteriores hemos accedido a las etiquetas (nodo de tipo element) del árbol, pero DOM permite acceder directamente a todos los atributos de una etiqueta. Para que los atributos de una etiqueta, puedan ser accedidos, estos contienen la propiedad attributes. Esta propiedad nos permite acceder a todos los atributos de un nodo de tipo element. Para ello hace uso de los siguientes métodos:

- **getNamedItem(nomAttr)**. Devuelve el nodo de tipo attr(atributo), cuya propiedad nodeName (nombre del nodo) contenga el valor nomAttr.
- **removeNamedItem(nomAttr).** Elimina el nodo de tipo attr (atributo) en el que la propiedad nodeName (Nombre del nodo) coincida con el valor nomAttr.
- **setNamedItem(nodo).** Este método añade el nodo attr (atributo) a la lista de atributos del nodo element. Lo indexa según la propiedad nodeName (del atributo).
- **item(pos).** Devuelve el nodo correspondiente a la posición indicada por el valor numérico pos.

En los métodos indicados anteriormente, el tipo de nodo es attr. Este tipo de nodo corresponde con los atributos que están integrados dentro de un nodo de tipo element (etiqueta). El nodo de tipo attr no devuelve directamente el valor del atributo, si no el nombre del atributo. A continuación mostramos cómo podemos utilizar los métodos anteriores:

```
<body>
<p id="parraf" style="color:blue">Prueba de texto</p>

<script>
    var p=document.getElementById("parraf");
    //accedemos al valor de la propiedad style del elemento
    //cuyo id es parraf mediante getNamedItem
    var valorId=p.attributes.getNamedItem("style").nodeValue;
    alert (valorId);
    //accedemos al valor de la propiedad style del elemento
    //cuyo id es parraf mediante item
    valorId=p.attributes.item(1).nodeValue;
    alert (valorId);
    //modificamos la propiedad style
    p.attributes.getNamedItem("style").nodeValue="color:red";
    valorId=p.attributes.getNamedItem("style").nodeValue;
    alert (valorId);
</script>
</body>
```

Estos métodos pueden resultar algo tediosos a la hora de acceder a los atributos de un nodo, por ello DOM proporciona unos métodos que permiten acceso directo a la modificación, inserción y borrado de los atributos de una etiqueta.

- **getAttribute(nomAtributo)**. Este método equivale a:

  attributes.getNameltem (nombAtributo).

- **setAttribute(nomAtributo, valor Atributo)**. Este método equivale a la estructura:

  attributes.getNamedltem (norrtAtributo) .value = valor.

- **removeAttribute(nomAtributo)**. Este método equivale a la estructura: attributes.removeNameltem (nomAtributo).

Con los siguientes métodos, podemos hacer las acciones del código anterior de la siguiente manera:

```
<p id="parraf" style="color:blue">Prueba de texto</p>

<script>
    var p=document.getElementById("parraf");
    //accedemos al valor de la propiedad style del elemento
    //cuyo id es parraf mediante getAttribute
    var valorId=p.getAttribute("id");
    alert (valorId);
    //modificamos el valor de style con setAttribute
    p.setAttribute("style","color:red");
```

Uso de atributos

Con estos métodos podemos manejar todos los atributos que tiene una etiqueta HTML.

4.4. Acceso a los nodos de tipo texto

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

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

Ejemplo html

Para poder referenciar el fragmento "texto HTML" del nodo P, lo que haremos será utilizar la colección childNodes. Con la colección childNodes accederemos a los nodos hijo de un elemento, ya sean de tipo elemento o texto. Y el código de JavaScript para mostrar una alerta, con el contenido "texto HTML", sería:

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

Acceso a los nodos hijo

- **childNodes[1]** : selecciona el segundo hijo de <p> que sería el elemento <b> (el primer hijo es un nodo de tipo Texto "Esto es un...").
- **childNodes[0]** : selecciona el primer hijo del elemento <b> que es el nodo de texto "texto HTML"

En lugar de childNodes[0] también podríamos haber utilizado firstChild, el cual nos devuelve el primer hijo de un nodo. Por ejemplo:

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

Comprobación del valor del nodo

El tamaño máximo de lo que se puede almacenar en un nodo de texto, depende del navegador, por lo que muchas veces, si el texto es muy largo, tendremos que consultar varios nodos para ver todo el contenido.

En el DOM de HTML, para acceder al valor de un nodo de texto, o modificarlo, es muy común ver la propiedad innerHTML. Esta propiedad es de Microsoft al igual que outerHTML. Aunque esta propiedad está soportada por casi todos los navegadores, y es muy rápida en su uso para hacer modificaciones, del contenido de un DIV por ejemplo, se recomienda usar el DOM si es posible.

Para modificar el contenido del nodo, modificaremos la propiedad nodeValue y le asignaremos otro valor. Por ejemplo en el caso anterior si hacemos:

```
document.getElementsByTagName("p")[0].childNodes[1].firstChild.nodeValue = "Texto MODIFICADO";
```

Modificación del nodo

Veremos que en la página web se ha cambiado la cadena "texto HTML", por "Texto MODIFICADO".

También podríamos por ejemplo, mover trozos de texto a otras partes. El siguiente ejemplo mueve el texto "en tu documento" a continuación de "Esto es un ejemplo de":

```
document.getElementsByTagName("p")[0].firstChild.nodeValue +=
    document.getElementsByTagName("p")[0].childNodes[2].nodeValue;
document.getElementsByTagName("p")[0].childNodes[2].nodeValue="";
```

Modificación del valor de los nodos

5. Creación y modificación de elementos

La creación y modificación de nodos fue uno de los objetivos para los que se creó el DOM. Se pueden crear elementos y luego insertarlos en el DOM, y la actualización quedará reflejada automáticamente por el navegador. Para la creación e inserción de nodos se utilizan los métodos **createElement()**, **createTextNode**() y **appendChild()**, que nos permitirán crear un elemento, crear un nodo de texto y añadir un nuevo nodo hijo.
En el siguiente ejemplo, vemos cómo se puede crear un nuevo párrafo, suponiendo que partimos del siguiente código HTML:

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

Para crear el nuevo párrafo haremos:

```
var nuevoParrafo = document.createElement('i');
```

```
var nuevoTexto = document.createTextNode('Contenido añadido al
párrafo en cursiva.');
nuevoParrafo.appendChild(nuevoTexto);
document.getElementById('parrafo').appendChild(nuevoParrafo);
```

Y obtendremos como resultado HTML:

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

Como alternativa, se puede utilizar **insertBefore** en lugar de **appendChild** o, incluso, añadir manualmente el nuevo elemento al final de la colección de nodos **childNodes**. Si usamos **replaceChild**, incluso podríamos sobrescribir nodos ya existentes. También es posible copiar un nodo usando **cloneNode**(true).

Para eliminar un nodo existente, lo podremos hacer con element.removeChild (referencia al nodo hijo).

Ejemplo de creación de elementos e inserción en el documento:

```
//Creamos tres elementos nuevos: p, b, br
var elementoP = document.createElement('p');
var elementoB = document.createElement('b');
var elementoBR = document.createElement('br');
//Le asignamos un nuevo atributo title al elementoP que hemos creado.
elementoP.setAttribute('title','Parrafo creado desde JavaScript');
//Preparamos los nodos de texto
var texto1 = document.createTextNode('Con JavaScript se ');
var texto2 = document.createTextNode('pueden realizar ');
var texto3 = document.createTextNode('un montón');
var texto4 = document.createTextNode(' de cosas sobre el documento.');
//Añadimos al elemento B los nodos de texto2, elemento BR y texto3.
elementoB.appendChild(texto2);
elementoB.appendChild(elementoBR);
elementoB.appendChild(texto3);
//Añadimos al elemento P los nodos de texto1, elemento B y texto 4.
elementoP.appendChild(texto1);
elementoP.appendChild(elementoB);
elementoP.appendChild(texto4);
//insertamos el nuevo párrafo como un nuevo hijo de nuestro parrafo
document.getElementById('parrafo').appendChild(elementoP);
```

6. Modelo de gestión de eventos

Cuando hablamos de una aplicación web, un evento es cualquier suceso relacionado con la aplicación web. Por lo tanto, un evento podría ser cerrar la ventana, dar un clic en un botón de radio o situarse con el ratón sobre la página.

Un elemento básico que permite a la aplicación web interaccionar con el usuario, es el formulario. Un formulario web, alojado en una página web, es una agrupación de objetos que tienen una función concreta. Estos objetos permiten al usuario introducir datos para enviarlos a un servidor y que sean procesados.

Los eventos son mecanismos que se accionan cuando el usuario realiza un cambio sobre una página web. En una página web existen multitud de elementos. Estos elementos son los objetos que modelan la página web. El encargado de crear la jerarquía de objetos que compone una página web es el DOM. Por lo tanto, es el DOM (Document Objet Model) el encargado de gestionar los eventos.

Los eventos se sitúan en componentes de la página HTML, si hacemos un clic en un componente de la página, podríamos desencadenar una acción. Por un lado deberíamos marcar en el componente cual es el evento que queremos que desencadene la acción. Para poder controlar un evento necesitamos un manejador. Un manejador es una palabra reservada que indica la acción que va a manejar. En el caso del evento click, el manejador sería onclick. En este caso podemos asociar a un componente el manejador y desencadenar una acción cuando se haga un clic sobre ese componente.

Cuando ha sucedido el evento y este tiene un manejador asociado necesitamos la ayuda de un lenguaje de script como JavaScript.

A continuación, vamos a ver un ejemplo de cómo JavaScript muestra un mensaje de alerta cuando hacemos clic sobre una imagen existente en la página web:

```
<img src="mundo.jpg" onclick="alert('Click en imagen');"/>
```

En el código anterior vemos que invocamos directamente a la función alert () de JavaScript. Si la acción a desencadenar es simple la podemos incluir directamente, aunque normalmente se realiza una llamada a una función. Será esta función la encargada de desencadenar la acción deseada. De esta forma estructuramos mejor el código y evitamos repetir sentencias si vamos a hacer varias veces una llamada para replicar un mismo comportamiento, pudiendo reutilizar una función en otra página si es preciso

A continuación mostramos la misma acción integrada en una página HTML llamando a una función:

```
<html>
```

```
  <head>
    <title>Pagina de Evento</title>
    <script>
      function func1 ()
      {
        alert("Click en imagen");
      }
    </script>
  </head>
 <body>
    <img src="mundo.jpg" onclick="func1();"/>
  </body>
</html>
```

En el código superior vemos como al el manejador onclick llama a la función, func1 para que se ejecute. En la parte superior del código, dentro del head y entre las etiquetas <script>...</script> es donde debemos definir la función func1 para que muestre la alerta cuando esta función sea llamada.

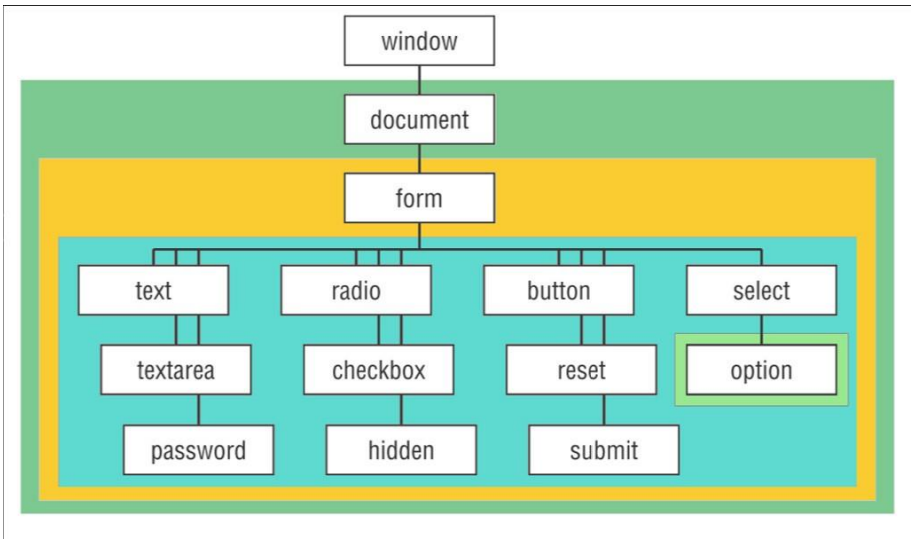


Imagen Nivel 0 del Dom

A continuación, vamos a ver que la especificación de DOM define grupos de eventos dividiéndolos según su origen, estos son: eventos del ratón, del teclado, HTML y de DOM:

- **Eventos del ratón**. Estos eventos se originan cuando el usuario hace uso del ratón para realizar una acción. Cualquier movimiento del ratón o acción sobre algunos de sus botones, puede desencadenar un evento. Cabe decir que no es necesario que la acción del ratón se realice sobre ningún objeto que interactúe con el usuario como puede ser un botón de radio o un check.
- **Eventos del teclado**. Estos eventos se originan cuando el usuario pulsa alguna tecla del teclado.

- **Eventos HTML**. Estos eventos se producen cuando hay algún cambio en la página del navegador. También pueden ocurrir cuando existe alguna interacción entre el cliente y el servidor.
- **Eventos DOM**. Los eventos DOM, o eventos de mutación, son los que se originan cuando existe algún cambio en la estructura DOM de la página.

6.1. Eventos del ratón

Los eventos del ratón son los más usuales puesto que la mayor parte de las acciones en una aplicación web las realizamos a través de este. A continuación, vamos a ver cuáles son los eventos que define la especificación DOM:

- **Click**. Este evento se produce cuando pulsamos sobre el botón izquierdo del ratón. El manejador de este evento es onclick.
- **Dblclick**. Este evento se acciona cuando hacemos un doble clic sobre el botón izquierdo del ratón. El manejador de este evento es ondblclick.
- **Mousedown**. Este evento se produce cuando pulsamos un botón del ratón. El manejador de este evento es onmousedown.
- **Mouseout**. Este evento se produce cuando el puntero del ratón está dentro de un elemento y este puntero es desplazado fuera del elemento. El manejador de este evento es onmouseout.
- **Mouseover**. Este evento al revés que el anterior se produce cuando el puntero del ratón se encuentra fuera de un elemento y éste se desplaza hacia el interior. El manejador de este evento es onmouseover.
- **Mouseup**. Este evento se produce cuando soltamos un botón del ratón que previamente teníamos pulsado. El manejador de este evento es onmouseup.
- **Mousemove**. Se produce cuando el puntero del ratón se encuentra dentro de un elemento. Es importante señalar que este evento se producirá continuamente una vez tras otra mientras el puntero del ratón permanezca dentro del elemento. El manejador de este evento es onmousemove.

Es importante destacar que el orden de ejecución de los eventos es el siguiente: **mousedown, mouseup, click, dblclick**. Siendo el último evento de la secuencia el doble clic.

Cabe señalar que todos los elementos de las páginas web soportan los eventos del ratón, pero no todos los eventos que existen son soportados por todos los elementos de las páginas.

Cuando se produce un evento el objeto event se crea automáticamente. Este objeto contiene características adicionales sobre el evento que se ha producido. Estos datos pueden ser, la posición del ratón, el elemento que ha producido el evento, etc.

Para los eventos de ratón, el objeto event tiene una serie de propiedades, algunas de ellas son; las coordenadas del ratón (screenX, screenY), el nombre del evento (type), el elemento que origina el evento (button), etc.

Los eventos del ratón son uno de los eventos más importantes en JavaScript. Cada vez que un usuario hace click en un elemento, al menos se disparan tres eventos y en el siguiente orden:

1. **mousedown**, cuando el usuario presiona el botón del ratón sobre el elemento.
2. **mouseup**, cuando el usuario suelta el botón del ratón.
3. **click**, cuando el usuario pulsa y suelta el botón sobre el elemento.

En general, los eventos de **mousedown** y **mouseup** son mucho más útiles que el evento **click**. Si por ejemplo presionamos el botón sobre un elemento A, nos desplazamos y soltamos el botón sobre otro elemento B, se detectarán solamente los eventos de **mousedown** sobre A y **mouseup** sobre B, pero no se detectará el evento de **click**. Esto quizás pueda suponer un problema, dependiendo del tipo de interacción que quieras en tu aplicación. Generalmente a la hora de registrar eventos, se suele hacer para **mousedown** y **mouseup**, a menos de que quieras el evento de **click** y no ningún otro. El evento de **dblclick** no se usa muy a menudo. Incluso si lo usas, tienes que ser muy prudente y no registrar a la vez **click** y **dblclick** sobre el mismo elemento, para evitar complicaciones.

El evento de **mousemove** funciona bastante bien, aunque tienes que tener en cuenta que la gestión de este evento le puede llevar cierto tiempo al sistema para su procesamiento. Por ejemplo, si el ratón se mueve 1 pixel, y tienes programado el evento de **mousemove**, para cada movimiento que hagas, ese evento se disparará, independientemente de si el usuario realiza o no realiza ninguna otra opción. En ordenadores antiguos, esto puede ralentizar el sistema, ya que para cada movimiento del ratón estaría realizando las tareas adicionales programadas en la función. Por lo tanto se recomienda utilizar este evento sólo cuando haga falta, y desactivarlo cuando hayamos terminado.

Otros eventos adicionales del ratón son los de **mouseover** y **mouseout**, que se producen cuando el ratón entra en la zona del elemento o sale del elemento. Si, por ejemplo, tenemos tres contenedores anidados divA, divB y divC: si programamos un evento de **mouseover** sobre la divA y nos vamos moviendo hacia el contenedor interno, veremos que ese evento sigue disparándose cuando estemos sobre divB o entremos en divC. Esta reacción se debe al burbujeo de eventos. Ni en divB o divC tenemos registrado el evento de **mouseover**, pero cuando se produce el burbujeo de dicho evento, se encontrará que tenemos registrado ese evento en el contenedor padre divA y por eso se ejecutará.

Muchas veces es necesario saber de dónde procede el ratón y hacia dónde va, y para ello W3C añadió la propiedad **relatedTarget** a los eventos de **mouseover** y **mouseout**. Esta propiedad contiene el elemento desde dónde viene el ratón en el caso de **mouseover**, o el elemento en el que acaba de entrar en el caso de **mouseout**.

Para saber los botones del ratón que hemos pulsado, disponemos de las propiedades **which** y **button**. Y para detectar correctamente el botón pulsado, lo mejor es hacerlo en los eventos de **mousedown** o **mouseup**. **which** es una propiedad antigua de Netscape, así que simplemente vamos a citar, **button** que es la propiedad propuesta por el W3C:

Los valores de la propiedad **button** pueden ser:

- **Botón izquierdo**: 0
- **Botón medio**: 1
- **Botón derecho**: 2

También es muy interesante conocer la posición en la que se encuentra el ratón, y para ello disponemos de un montón de propiedades que nos facilitan esa información:

- **clientX, clientY**: devuelven las coordenadas del ratón relativas a la ventana.
- **offsetX, offsetY**: devuelven las coordenadas del ratón relativas al objeto destino del evento.
- **pageX, pageY**: devuelven las coordenadas del ratón relativas al documento. Estas coordenadas son las más utilizadas.
- **screenX, screenY**: devuelven las coordenadas del ratón relativas a la pantalla.

Ejemplo que muestra las coordenadas del ratón al moverlo en el documento:

```
<input type="text" id="coordenadas" name="coordenadas" size="12"/>
<script>
    function mostrarCoordenadas(elEvento)
    {
        document.getElementById("coordenadas").value=elEvento.clientX+" : "+elEvento.clientY;
    }
    document.addEventListener('mousemove',mostrarCoordenadas,false);
</script>
```

Uso de las coordenadas

6.2. Eventos del teclado

Los eventos de teclado son los que suceden cuando pulsamos una tecla. Según la especificación DOM existen los siguientes eventos relacionados con la actividad del teclado.

- **Keydown**. Este evento se produce cuando pulsamos una tecla del teclado. Si mantenemos pulsada una tecla de forma continua, el evento se produce una y otra vez hasta que soltemos la misma. El manejador de este evento es **onkeydown**.
- **Keypress**. Este evento se produce si pulsamos una tecla de un carácter alfanumérico (el evento no se produce si pulsamos Enter, la barra espaciadora, etc.). En el caso de mantener una tecla pulsada, el evento se produce de forma continuada. El manejador de este evento es **onkeypress**.
- **Keyup**. Este evento se produce cuando soltamos una tecla. El manejador de este evento es **onkeyup**.

Las propiedades de event para los eventos de teclado son: el código numérico de la tecla pulsada (keyCode), el código unicode del carácter correspondiente a la tecla pulsada (charCode), el elemento que origina el evento (target) y otras para identificar si hemos pulsado shift (shif tKey), control (ctrlKey), alt (altKey) o meta (metaKey).

Es importante destacar que el orden de secuencia de las teclas no es el mismo para un tipo de teclas que para otras. A continuación, veremos cuál es el orden:

- Cuando pulsamos una tecla que corresponda a un carácter alfanumérico, la secuencia de eventos es la siguiente: **keydown**, **keypress**, **keyup**.
- En el caso de pulsar una tecla que no corresponda a un carácter alfanumérico, la secuencia de eventos es: **keydown**, **keyup**.
- Además, existe la posibilidad de que dejemos una tecla pulsada. Para el primer caso, se repiten de forma continua los eventos **keydown**, **keypress**. En el segundo caso se repite de forma continuada el evento **keydown** solamente.

En el proceso de pulsación de una tecla se generan tres eventos seguidos: keydown, keypress y keyup. Y para cada uno de ellos disponemos de las propiedades **keyCode** y **charCode**. Para saber la tecla que se ha pulsado lo más cómodo es acceder al evento keypress.

- **keydown**: se produce al presionar una tecla y mantenerla presionada. Su comportamiento es el mismo en todos los navegadores.
    - Propiedad **keyCode**: devuelve el código interno de la tecla.
    - Propiedad **charCode**: no está definida.
- **keypress**: se produce en el instante de presionar la tecla.
    - Propiedad **keyCode**: devuelve el código interno de las teclas especiales, para las teclas normales no está definido.
    - Propiedad **charCode**: devuelve 0 para las teclas especiales o el código del carácter de la tecla pulsada para las teclas normales. (En Internet Explorer y Edge **keyCode** devuelve el carácter de la tecla pulsada, y charCode no está definido).
- **keyup**: se produce al soltar una tecla presionada. Su comportamiento es el mismo en todos los navegadores.
    - Propiedad **keyCode**: devuelve el código interno de la tecla. ○ Propiedad **charCode**: no está definida.

Ejemplo que mueve el foco de un campo de texto a otro, dentro de un formulario, al pulsar la tecla **ENTER** dentro de cada campo:

Responder a eventos DOM

El formulario sería el siguiente.

```
<form name="formulario" id="formulario">
    <label for="nombre">Nombre</label><input type="text" name="nombre" id="nombre">
    <label for="apellidos">Apellidos</label><input type="text" name="apellidos" id="apellidos">
    <label for="provincia">Provincia</label><input type="text" name="provincia" id="provincia">
    <input type="button" id="enviar" value="enviar">
</form>
```

El script seria el que sigue a continuación.

```
<script type="text/javascript">
    function cambiar(evt)
    {
        if (evt.keyCode == 13) //Tecla enter
            {
            // la propiedad type solo funciona con button, para las demás hay que usar nodeName
                if (this.nextElementSibling.nodeName=="LABEL")
                    this.nextElementSibling.nextElementSibling.focus();
                else
                    document.getElementById('nombre').focus();
            }
    }
    var inputs= document.getElementsByTagName('input');
    for (i=0; i<inputs.length; i++)
    {
        inputs[i].addEventListener("keypress",cambiar,false);
    }
</script>
```

En la estructura HTML del formulario, los campos del formulario no llevan saltos de línea entre unos y otros, por las siguientes razones:

- La propiedad de sólo lectura **Node.nextSibling** devuelve el siguiente nodo con respecto al indicado en la lista de nodos (childNodes) a la que este pertenece o null si el nodo especificado es el último en dicha lista.
- **Element.nextElementSibling** es utilizado para obtener el siguiente elemento ignorando cualquier nodo en blanco.

6.3. Eventos html

Los eventos HTML, como hemos visto anteriormente, son los que actúan cuando hay cambios en la ventana del navegador. También se producen cuando hay ciertas interacciones entre cliente y el servidor.

- **Load**. El evento load hace referencia a la carga de distintas partes de la página. Este se produce en el objeto Window cuando la página se ha cargado por completo. En el elemento <img> actúa cuando la imagen se ha cargado. En el elemento <object> se acciona al cargar el objeto completo. El manejador es onload.
- **Unload**. El objeto unload actúa sobre el objeto Window cuando la página ha desaparecido por completo (por ejemplo, si pulsamos el aspa cerrando la ventana del navegador). También se acciona en el elemento <object> cuando desaparece el objeto. El manejador es onunload.
- **Abort**. Este evento se produce cuando el usuario detiene la descarga de un elemento antes de que haya terminado, actúa sobre un elemento <object>. El manejador es onabort.
- **Error**. El evento error se produce en el objeto Window cuando se ha producido un error en JavaScript. En el elemento <img> cuando la imagen no se ha podido cargar por completo y en el elemento <object> en el caso de que un elemento no se haya cargado correctamente. El manejador es onerror.
- **Select**. Se acciona cuando seleccionamos texto de los cuadros de textos <input> y <textarea>. El manejador es onselect.
- **Change**. Este evento se produce cuando los cuadros de texto <input> y <textarea> pierden el foco y el contenido que tenían ha variado. También se producen cuando un elemento <select> cambia de valor. El manejador es onchange.
- **Submit**. Este evento se produce cuando pulsamos sobre un botón de tipo submit. El manejador es onsubmit.
- **Reset**. Este evento se produce cuando pulsamos sobre un botón de tipo reset. El manejador es onreset.
- **Resize**. Este evento se produce cuando redimensionamos el navegador, actúa sobre el objeto Window. El manejador es onresize.
- **Scroll**. Se produce cuando varía la posición de la barra de desplazamiento (scroll) en cualquier elemento que la tenga. El manejador es onscroll.
- **Focus**. Este evento se produce cuando un elemento obtiene el foco. El manejador es onfocus.
- **Blur**. Este evento se produce cuando un elemento pierde el foco. El manejador es onblur.

6.4. Eventos DOM

Estos eventos hacen referencia a la especificación DOM. Se accionan cuando varía el árbol DOM. Estos eventos no están implementados en todos los navegadores, por tanto,

habría que comprobar que el navegador soporta estos eventos, aun sabiendo que el navegador cumple con la especificación DOM.

- **DOMSubtreeModified**. Este evento se produce cuando añadimos o eliminamos nodos en el subárbol de un elemento o documento.
- **DOMNodeInserted**. Este evento se produce cuando añadimos un nodo hijo a un nodo padre.
- **DOMNodeRemoved**. Este evento se produce cuando eliminamos un nodo.
- **DOMNodeRemovedFromDocument**. Este evento se produce cuando eliminamos un nodo del documento.
- **DOMNodeInsertedIntoDocument**. Este evento se produce cuando añadimos un nodo al documento.

7. Manejadores de eventos

Hay que tener en cuenta que, sin eventos prácticamente no hay scripts. En casi todas las páginas web que incorporan JavaScript, suele haber eventos programados que disparan la ejecución de dichos scripts. La razón es muy simple, JavaScript fue diseñado para añadir interactividad a las páginas: el usuario realiza algo y la página reacciona. Por lo tanto, JavaScript necesita detectar de alguna forma las acciones del usuario para saber cuándo reaccionar. También necesita saber las funciones, que queremos que ejecute cuando se produzcan esas acciones. Cuando el usuario hace algo se produce un evento. También habrá algunos eventos que no están relacionados directamente con acciones de usuario: por ejemplo el evento de carga (load) de un documento, que se producirá automáticamente cuando un documento ha sido cargado.

7.1. Modelo de registro de eventos en línea

En el modelo de registro de eventos en línea (estandarizado por Netscape), el evento es añadido como un atributo más a la etiqueta HTML, como por ejemplo:

```
<a href="pagina.html" onClick="alert('Has pulsado en el enlace')">Pulsa aqui</a>
```

Cuando hacemos click en el enlace, se llama al gestor de eventos onClick (al hacer click) y se ejecuta el script; que contiene en este caso una alerta de JavaScript. También se podría realizar lo mismo pero llamando a una función:

```
<a href="pagina.html" onClick="alertar()">Pulsa aqui</a>


function alertar()
{     alert("Has pulsado en el enlace");
}
```

Este modelo no se recomienda, tiene el problema de que estamos mezclando la estructura de la página web con la programación de la misma, y lo que se intenta hoy

en día es separar la programación en JavaScript, de la estructura HTML, por lo que este modelo no nos sirve.

En el ejemplo anterior, cuando se hace click en el enlace se mostrará la alerta y a continuación te conectará con la pagina.html. En ese momento desaparecerán de memoria los objetos que estaban en un principio, cuando se programó el evento. Esto puede ser un problema, ya que, si por ejemplo la función a la que llamamos, cuando se produce el evento, tiene que realizar varias tareas, éstas tendrían que ser hechas antes de que nos conecte con la nueva página. Este modo de funcionamiento ha sido un principio muy importante en la gestión de eventos. Si un evento genera la ejecución de un script y además también se genera la acción por defecto para ese objeto entonces:

1. El script se ejecutará primero.
2. La acción por defecto se ejecutará después.

A veces es interesante el bloquear o evitar que se ejecute la acción por defecto. Por ejemplo, en nuestro caso anterior podríamos evitar que nos conecte con la nueva pagina.html. Cuando programamos un gestor de eventos, ese gestor podrá devolver un valor booleano true o false. Eso tendremos que programarlo con la instrucción return true|false. False quiere decir "no ejecutes la acción por defecto".

Por lo tanto nuestro ejemplo quedará del siguiente modo:

```
function alertar()
{
  alert("Has pulsado en el enlace");
  return(false)
}
```

De esa forma, cada vez que pulsemos en el enlace realizará la llamada a la función alertar() y cuando termine ejecutará la instrucción "return false", que le indicará al navegador que no ejecute la acción por defecto asignada a ese objeto (en este caso la acción por defecto de un hiperenlace es conectarnos con la página href de destino). También sería posible que nos preguntara si queremos o no queremos ir a la pagina.html. Eso podríamos hacerlo sustituyendo true o false por una función que devuelva true o false según la respuesta que demos al "confirm":

```
<a href="pagina.html" onClick="return preguntar()">Pulsa aqui</a>

function preguntar()
{         return  confirm("¿Desea  realmente  ir  a  esa
dirección?");
}
```

7.2. Modelo de registro de eventos tradicional.

En los navegadores antiguos, el modelo que se utilizaba era el modelo en línea. Con la llegada de DHTML, el modelo se extendió para ser más flexible. En este nuevo modelo el evento pasa a ser una propiedad del elemento, así que por ejemplo los navegadores modernos ya aceptan el siguiente código de JavaScript:

```
elemento.onclick = hacerAlgo;

// cuando el usuario haga click en el objeto, se llamará a la función
// hacerAlgo()
```

Esta forma de registro, no fue estandarizada por el W3C, pero debido a que fue ampliamente utilizada por Netscape y Microsoft, todavía es válida hoy en día. La ventaja de este modelo es que podremos asignar un evento a un objeto desde JavaScript, con lo que ya estamos separando el código de la estructura. Fíjate que aquí los nombres de los eventos sí que van siempre en minúsculas.

```
elemento.onclick = hacerAlgo;
```

Para eliminar un gestor de eventos de un elemento u objeto, le asignaremos null:
`elemento.onclick = null;`

Otra gran ventaja es que, como el gestor de eventos es una función, podremos realizar una llamada directa a ese gestor, con lo que estamos disparando el evento de forma manual. Por ejemplo:

```
elemento.onclick();

// Al hacer ésto estamos disparando el evento click de forma //manual
y se ejecutará la función hacerAlgo()
```

En el registro del evento no usamos paréntesis (). El método onclick espera que se le asigne una función completa. Si se hace: ***element.onclick = hacerAlgo();*** la función será ejecutada y el resultado que devuelve esa función será asignado a onclick. Pero esto no es lo que queremos que haga, queremos que se ejecute la función cuando se dispare el evento.

USO DE LA PALABRA RESERVADA THIS

En el modelo en línea podemos utilizar la palabra reservada this cuando programamos el gestor de eventos. Por ejemplo:

```
<a href="pagina.html" id="mienlace" onclick="alertar(this)">Pulsa aqui</a>

<script>

  function alertar(objeto)
  {
    alert("Te conectaremos con la página: "+objeto.href);
  }
```

```
</script>
 Su equivalente en el <strong>modelo tradicional</strong> sería:
<a id="mienlace" href="pagina.html">Pulsa aqui</a>
<script>
  document.getElementById("mienlace").onclick = alertar;
    function alertar()
  {
    alert("Te conectaremos con la página: "+this.href);
  }
</script>
```

Usamos this dentro de la función, sin pasar ningún objeto. En el modelo tradicional el this que está dentro de la función, hace referencia al objeto donde hemos programado el evento. También hay que destacar que en este modelo es importante que el hiperenlace sea declarado antes de programar la asignación onclick, ya que si por ejemplo escribimos el hiperenlace después del bloque de código de JavaScript, éste no conocerá todavía el objeto y no le podrá asignar el evento de onclick. Esto también se podría solucionar, programando la asignación de eventos a los objetos, en una función que se ejecute cuando el documento esté completamente cargado. Por ejemplo con window.onload=asignarEventos.

Si por ejemplo queremos que cuando se produzca el evento se realicen llamadas a más de una función lo podremos hacer de la siguiente forma:

```
elemento.onclick = function {llamada1; llamada2 };
```

7.3. Modelo de registro avanzado de eventos según w3c

El W3C en la especificación del DOM de nivel 2, pone especial atención en los problemas del modelo tradicional de registro de eventos. En este caso ofrece una manera sencilla de registrar los eventos que queramos, sobre un objeto determinado.

La clave para poder hacer todo eso está en el método addEventListener().

Este método tiene tres argumentos: el tipo de evento, la función a ejecutar y un valor booleano (true o false), que se utiliza para indicar cuándo se debe capturar el evento: en la fase de captura (true) o de burbujeo (false).

```
elemento.addEventListener('evento', función, false|true)
```

Por ejemplo para registrar la función alertar() de los ejemplos anteriores, haríamos:

```
document.getElementById("mienlace").addEventListener('click',alertar,
false);


function alertar()
{     alert("Te conectaremos con la página: "+this.href);    }
```

La ventaja de este método, es que podemos añadir tantos eventos como queramos. Por ejemplo:

```
document.getElementById("mienlace").addEventListener('click',alertar,
false);
document.getElementById("mienlace").addEventListener('click',avisar,
false);
document.getElementById("mienlace").addEventListener('click',chequear,
false);
```

Por lo tanto, cuando hagamos click en "mienlace" se disparará la llamada a las tres funciones. Por cierto, el W3C no indica el orden de disparo, por lo que no sabemos cuál de las tres funciones se ejecutará primero. Además, el nombre de los eventos al usar addEventListener no lleva 'on' al comienzo. También se pueden usar funciones anónimas (sin nombre de función), con el modelo W3C:

```
element.addEventListener('click', function () {this.style.backgroundColor =
'#cc0000';}, false)
```

Uno de los problemas de la implementación del modelo de registro del W3C, es que no podemos saber con antelación, los eventos que hemos registrado a un elemento. En el modelo tradicional si hacemos: alert(elemento.onclick), nos devuelve undefined, si no hay funciones registradas para ese evento, o bien el nombre de la función que hemos registrado para ese evento. Pero en este modelo no podemos hacer eso.

El W3C en el reciente nivel 3 del DOM, introdujo un método llamado eventListenerList, que almacena una lista de las funciones que han sido registradas a un elemento. Este método no está soportado por todos los navegadores. Para eliminar un evento de un elemento, usaremos el método removeEventListener():

elemento.removeEventListener('evento', función, false|true);

Para cancelar un evento, este modelo nos proporciona el método preventDefault().

7.4. Modelo de registro de eventos según Microsoft

Microsoft también ha desarrollado un modelo de registro de eventos. Es similar al utilizado por el W3C, pero tiene algunas modificaciones importantes.

Para registrar un evento, tenemos que enlazarlo con attachEvent(): elemento.attachEvent('onclick', hacerAlgo); o si se necesitan dos gestores del mismo evento:

```
elemento.attachEvent('onclick',  hacerUnaCosa);
elemento.attachEvent('onclick', hacerOtraCosa);
```

Para eliminar un evento, se hará con detachEvent():

```
elemento.detachEvent('onclick', hacerAlgo);
```

Comparando este modelo con el del W3C tenemos dos diferencias importantes:

- Los eventos siempre burbujean, no hay forma de captura.

- La función que gestiona el evento está referenciada, no copiada, con lo que la palabra reservada this siempre hace referencia a window y será completamente inútil.

Como resultado de estas dos debilidades, cuando un evento burbujea hacia arriba es imposible conocer cuál es el elemento HTML que gestionó ese evento.

7.5. Orden de disparo de los eventos

Imaginemos que tenemos un elemento contenido dentro de otro elemento, y que tenemos programado el mismo tipo de evento para los dos (por ejemplo el evento click). ¿Cuál de ellos se disparará primero? Sorprendentemente, esto va a depender del tipo de navegador que tengamos.

El problema es muy simple. Imagina que tenemos el siguiente gráfico:

y ambos tienen programado el evento de click. Si el usuario hace click en el elemento2, provocará un click en ambos: elemento1 y elemento2. ¿Pero cuál de ellos se disparará primero?, ¿cuál es el orden de los eventos? Tenemos dos Modelos propuestos por Netscape y Microsoft en sus orígenes:

- Netscape dijo que, el evento en el elemento1 tendrá lugar primero. Es lo que se conoce como "captura de eventos".
- Microsoft mantuvo que, el evento en el elemento2 tendrá precedencia. Es lo que se conoce como "burbujeo de eventos".

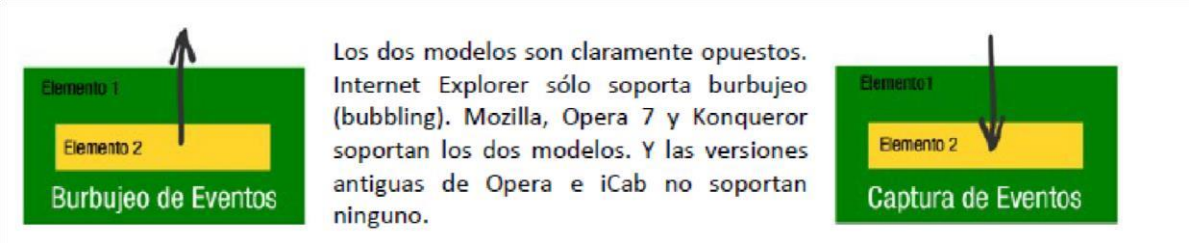


Imagen Burbujeo y captura de eventos

Modelo W3C

W3C decidió tomar una posición intermedia. Así supone que, cuando se produce un evento en su modelo de eventos, primero se producirá la fase de captura hasta llegar al elemento de destino, y luego se producirá la fase de burbujeo hacia arriba. Este modelo es el estándar, que todos los navegadores deberían seguir para ser compatibles entre sí.

Podremos decidir cuándo se registra el evento: en la fase de captura o en la fase de burbujeo. El tercer parámetro de **addEventListener** te permitirá indicar si lo hacemos en la fase de captura (true), o en la fase de burbujeo(false). Por ejemplo:

```
elemento1.addEventListener('click',hacerAlgo1,true);
elemento2.addEventListener('click',hacerAlgo2,false);
```

Si el usuario hace click en el elemento2 ocurrirá lo siguiente:

El evento de click comenzará en la fase de captura. El evento comprueba si hay algún ancestro del elemento2 que tenga un evento de onclick para la fase de captura (true).

El evento encuentra un elemento1.hacerAlgo1() que ejecutará primero, pues está programado a true.

El evento viajará hacia el destino, pero no encontrará más eventos para la fase de captura. Entonces el evento pasa a la fase de burbujeo, y ejecuta hacerAlgo2(), el cual hemos registrado para la fase de burbujeo (**false**).

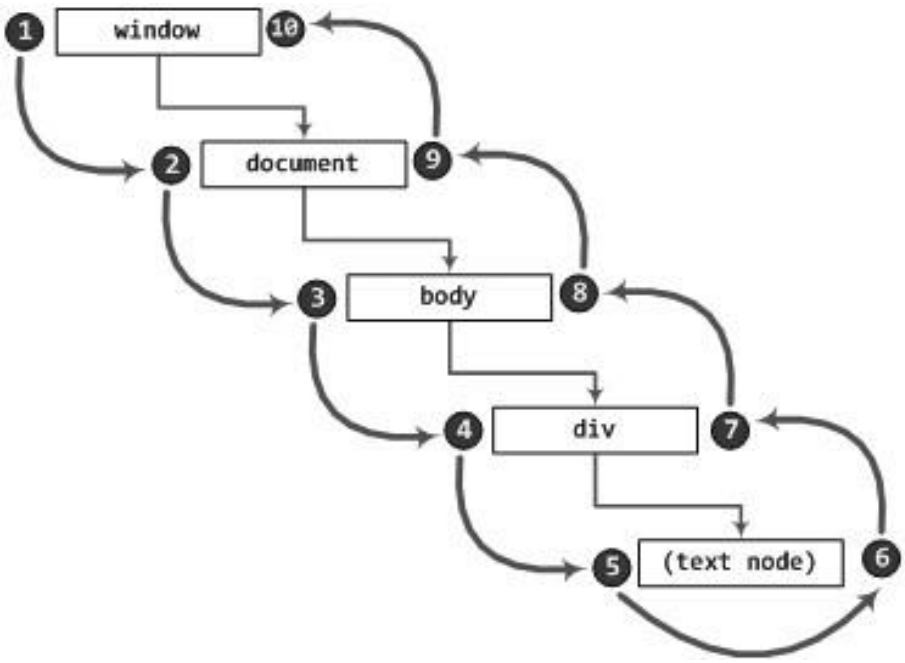
El evento viaja hacia arriba de nuevo y chequea si algún ancestro tiene programado un evento para la fase de burbujeo. Éste no será el caso, por lo que no hará nada más.

Imagen Modelo W3c

Para detener la propagación del evento en la fase de burbujeo, disponemos del método **stopPropagation**(). En la fase de captura es imposible detener la propagación.

8. Programación de eventos

A lo largo de los puntos anteriores, hemos visto multitud de acciones asociadas al modelo de objetos del documento (DOM). Hemos visto como podíamos modificar la estructura del árbol, añadir nodos, eliminarlos. En una aplicación web, es el usuario el que interactúa, además de desencadenantes como el tiempo u otras razones externas a la aplicación web. Por esta razón necesitamos algo que relacione la interacción del usuario con las acciones de DOM vistas anteriormente. Para relacionar esto, utilizamos los eventos. En el capítulo anterior veíamos como un evento era capaz de capturar una acción determinada. En función de la acción, por ejemplo hacer clic con el ratón, podíamos tomar una decisión programada. El lenguaje de programación de script que vamos a utilizar es JavaScript.



Esquema del flujo de eventos del modelo DOM

8.1. Modelo de eventos

Tenemos varios modelos de registro de eventos:

- **Modelo de registro de eventos en línea:**
  - Los eventos se añaden como un atributo más del objeto.
  - No es un modelo recomendado hoy en día, porque el código de JavaScript está integrado con el HTML y lo que se intenta conseguir es tener separación entre la estructura y la programación. Ejemplo:

```
<a href="pagina.html" onClick="alertar()">Pulsa aqui</a>
```

- **Modelo de registro de eventos tradicional:**
  - Los eventos se asignan como una propiedad del objeto y fuera de la estructura HTML.
  - No es un modelo estándar de registro.
  - Uso de la palabra reservada **this**, para hacer referencia al objeto dónde se programó el evento.
  - Para asignar un evento se podría hacer: **elemento.evento = hacerAlgo;** ○ Para eliminar ese evento del objeto: **elemento.evento = null;** ○ Ejemplo: `document.getElementById("mienlace").onclick = alertar;`

- **Modelo de registro avanzado de eventos según W3C:** ○ Es el estándar propuesto por el W3C. ○ Para asignar un evento se usa **addEventListener**(). ○ Para eliminar un evento se usa **removeEventListener**().
    - Se puede programar cuando queremos que se dispare el evento: en la fase de captura o burbujeo.
    - Uso de la palabra reservada this, para hacer referencia al objeto dónde se programó el evento.
    - Por ejemplo:

```
document.getElementById("mienlace").addEventListener('click',alertar
,false);
```

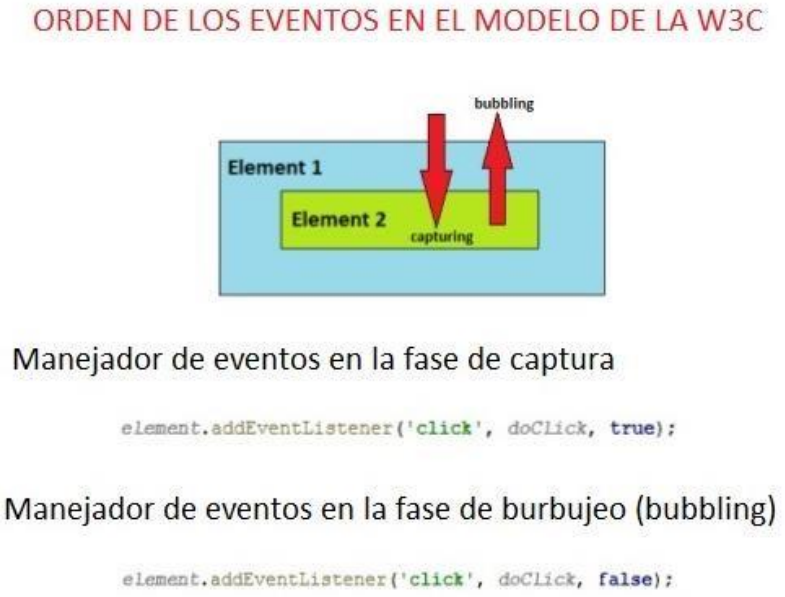
- **Modelo de registro de eventos según Microsoft:** ○ Se parece al utilizado por el W3C. ○ Para asignar un evento se usa **attachEvent**(). ○ Para eliminar un evento se usa **detachEvent**(). ○ Aquí los eventos siempre burbujean, no hay forma de captura.
    - No se puede usar la palabra reservada this, ya que la función es copiada, no referenciada.
    - El nombre de los eventos comienza por "**on**" + **nombre** de **evento** Por ejemplo:

```
document.getElementById("mienlace").attachEvent('onclick', alertar);
```

Y tenemos tres modelos propuestos de disparo de eventos, que clarificarán el orden de disparo de los mismos, cuando se solapen eventos sobre elementos anidados:

- **Modelo de captura de eventos:** En este modelo los eventos se van disparando de afuera hacia adentro. Es decir, primero se disparará el evento asignado al elemento exterior, y continúa descendiendo y disparando los eventos que coincidan, hasta llegar al elemento interior.
- **Modelo de burbujeo de eventos**: En este modelo los eventos se van disparando desde dentro hacia afuera. Es decir, primero se disparará el evento asignado al elemento interior, y continúa subiendo y disparando los eventos que coincidan, hasta llegar al elemento exterior.
- **Modelo W3C:** En este modelo se integran los dos modelos anteriores. Simplemente se realiza la fase de captura de eventos primero y, cuando termina, se realiza la fase de burbujeo. En este modelo cuando registramos un evento con **addEventListener**(evento, funcion, true|false) tenemos la opción de indicar cuándo queremos que se dispare el evento:
    - en la fase de captura (**, , true)**
    - en la fase de burbujeo (**, , false**)

- También disponemos de un nuevo método para cancelar eventos con **preventDefault**(), y de un método para detener la propagación de eventos en la fase de burbujeo, con **stopPropagation**().



Orden de captura de eventos en el modelo w3c

8.2. Tipos de eventos

Los navegadores anteriores a la versión 4 no tenían acceso al objeto "Evento". Posteriormente, cuando incorporaron los eventos, sólo dejaban asignar algunos tipos de eventos a ciertos elementos HTML, pero, hoy en día, ya podemos aplicar tipos de eventos virtualmente a casi cualquier elemento.

Al principio los eventos se solían asociar en línea en la etiqueta HTML, con un atributo que comenzaba por "on" seguido del tipo del evento, por ejemplo: onClick=..., onSubmit=..., onChange=..., pero hoy en día esa forma de uso está quedando obsoleta, debido a los nuevos modos de registro de eventos propuestos por elW3C, y que soportan ya la mayoría de navegadores modernos.

Eventos comunes en el W3C. Hay una colección enorme de eventos que pueden ser generados para la mayor parte de elementos HTML:

- Eventos de ratón**.**
- Eventos de **teclado**.
- Eventos **objetos frame**.
- Eventos de **formulario**.
- Eventos de **interfaz** de **usuario**.
- Eventos de **mutación** (notifican de cualquier cambio en la estructura de un documento).

Tabla. Tabla de eventos

Evento	Descripción	Elementos para los que está definido
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Deseleccíonar 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>
onmousedown	Pulsar (sin soltar) un botón del ratón	Todos los elementos
onmouseover	Mover el ratón	Todos los elementos
onmouseout	Ei ratón "sale" del elemento {pasa por encima de otro	Todos los elementos
onffmouseover	El elemento)ratón "entra" en el elemento (pasa por encima	Todos los elementos
onmouseup	Soltar e botón que estaba del elemento) 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 de! navegador	< body >
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página	<body>

8.3. El objeto Event

Generalmente, los manejadores de eventos necesitan información adicional para procesar las tareas que tienen que realizar. Si una función procesa, por ejemplo, el evento click, lo más probable es que necesite conocer la posición en la que estaba el

ratón en el momento de realizar el click; aunque esto quizás tampoco sea muy habitual, a no ser que estemos programando alguna utilidad de tipo gráfico.

Lo que sí es más común es tener información adicional en los eventos del teclado. Por ejemplo, cuando pulsamos una tecla nos interesa saber cuál ha sido la tecla pulsada, o si tenemos alguna tecla especial pulsada como Alt, Control, etc.

Para gestionar toda esa información disponemos del objeto Event, el cual nos permitirá acceder a esas propiedades adicionales que se generan en los eventos.

Como siempre, los navegadores gestionan de forma diferente los objetos Event. Por ejemplo, en las versiones antiguas de Internet Explorer, el objeto Event forma parte del objeto Window, mientras que en otros navegadores como Firefox, Chrome, etc., para acceder al objeto **Event** lo haremos a través de un parámetro, que escribiremos en la función que gestionará el evento.

Por ejemplo:

```
document.getElementById("unparrafo").addEventListener('click',gestionar,false);
function gestionar(miEvento)
{
    alert (miEvento.type);
    // Mostrará una alerta con el tipo de evento que en este caso es 'click'.
}
```

Añadir manejadores

En el código del ejemplo anterior cuando se produce el evento de click en un párrafo con id="unparrafo", durante la fase de burbujeo, se llamará a la función gestionar. En la función gestionar hemos creado un argumento que le llamamos miEvento, y es justamente en ese argumento que hemos puesto en la función, dónde el navegador de forma automática, pondrá todos los datos referentes al evento que se ha disparado.

Una vez dentro de la función, mostramos una alerta con el tipo de evento (propiedad type del objeto Event) que se acaba de disparar.

Tabla. Propiedades del objeto Event

Propiedades	Descripción
altKey, ctrlKey, metaKey, shiftKey	Valor booleano que indica si están presionadas algunas de las teclas Alt, Ctrl, Meta o Shift en el momento del evento
bubbles	Valor booleano que indica si el evento burbujea o no
button	Valor integer que indica que botón del ratón ha sido presionado o soltado 0=izquierdo, 2=derecho, 1=medio
cancelable	Valor booleano que indica si el evento se puede cancelar
charCode	Indica el carácter Unicode de la tecla presionada
clientX, clientY	Devuelve las coordenadas de la posición del ratón en el momento del evento
currentTarget	El elemento al que se asignó el evento.

evenPhase	Un valor integer que indica la fase del evento que está siendo procesada. Fase de captura (1), en destino (2) o fase de burbujeo (3).
layerX, layerY	Devuelve las coordenadas del ratón relativas a un elemento posicionado absoluta o relativamente. Si el evento ocurre fuera de un elemento posicionado se usará la esquina superior izquierda del documento
pageX, pageY	Devuelve las coordenadas del ratón relativas a la esquina superior izquierda de una página
relatedTarget	En un evento de “mouseover” india el nodo que ha abandonado el ratón. En un evento “mouseout” indica el nodo hacia el que se ha movido el ratón
screenX, screenY	Devuelve las coordenadas del ratón relativas a la pantalla dónde se diparó el evento.
target	El elemento donde se originó el evento, que puede diferir del elemento que tenga asignado el evento
timestamp	Devuelve la hora a la que se creó el evento
type	Una cadena de texto que indica el tipo de evento “click”, “mouseout”, “mouseover”
which	Indica el Unicode de la tecla presionada.

Tabla. Método de Event

Métodos	Descripción
preventDefault()	Cancela cualquier acción asociada por defecto a un evento
stopPropagation()	Evita que un evento burbujee.

8.4. Comprobar si el árbol DOM está cargado

Una condición indispensable para que se genere la estructura de árbol, es que la página se haya cargado completamente. En el caso de que la página no se haya cargado, no podemos acceder a la estructura de árbol, puesto que esta no se ha generado, o al menos no se ha generado completamente. Por lo tanto, necesitamos conocer si la página se ha cargado, antes de realizar cualquier actuación sobre la estructura del árbol. Para comprobar si la página se ha cargado completamente, disponemos del evento onload. Este evento está definido para el nodo de tipo element, <body>. A continuación, vamos a invocar al evento en una página HTML. Nos mostrará el mensaje ("Página cargada completamente") cuando ésta se cargue completamente.

```
<html>
  <head>
    <title>Titulo DOM</title>
  </head>
  <body onload="alert( 'Página cargada completamente');">
    <p>Primer parrafo</p>
  </body>
</html>
```

Ejemplo de carga

El código anterior muestra el mensaje "Página cargada completamente" cuando la página se carga. De esta forma podemos asegurarnos de que el árbol esta completado.

En DOM podemos modificar la estructura original de la página como hemos visto en los puntos anteriores. Por lo tanto, necesitamos poder saber si una vez hemos realizada una modificación en el árbol, la página se ha cargado de nuevo por completo.

A continuación, vamos a comprobar a través de una función si se ha cargado la página. En el caso de que se haya cargado, al hacer clic sobre el párrafo, mostraremos el mensaje: "Se cargó la página correctamente".

```
<html>
  <head>
    <title>Titulo DOM</title>
    <script>
      function cargada ()
      {
        window.onload="true";
        if (window.onload) {
          return true;
         }
        return false;
      }
      function  pulsar ()
      {
        if (cargada ())
        {
          alert("Se cargó la página correctamente");
        }
      }
    </script>
  </head>
  <body>
      <p onclick="pulsar();">Primer párrafo</p>
  </body>
</html>
```

Control de carga del DOM

En el código anterior vemos que la función **cargada()**, comprueba sobre el objeto window (objeto de más alto nivel, la ventana), si ocurrió el evento onload (el evento onload ocurre, cuando la página se ha cargado completamente). Cuando pulsemos con un clic sobre "Primer párrafo" se comprueba si la función cargada() es verdadera. En caso de ser verdadera, nos muestra el mensaje "Se cargó la página correctamente". Si no, no muestra nada.

8.5. Actuar sobre el DOM al desencadenarse eventos

Una de las grandes ventajas que nos aporta DOM es poder actuar sobre la estructura del árbol, de forma inmediata. Los eventos por otro lado nos permiten definir el momento en el que queremos que se realice una acción. Estas dos características combinadas, convierten la arquitectura web en el lado del cliente, en algo dinámico, rápido y versátil, para conseguir cualquier objetivo.

A continuación, vamos a ver cómo podemos actuar sobre la estructura del árbol, cuando se desencadena un evento. Una etiqueta div, tiene un valor por defecto. Cuando pasamos el ratón por encima cambia su valor, cuando el ratón no está por encima, el div toma otro valor diferente.

```
<html>
  <head>
    <title>Titulo DOM</title>
    <script>
      function ratonEncima () {
        document.getElementsByTagName("div")[0].childNodes[0].nodeValue="EL RATÓN ESTA ENCIMA";
      }

      function ratonFuera () {
       document.getElementsByTagName("div")[0].childNodes[0].nodeVale="NO  ESTA  EL  RATÓN ENCIMA";
      }
    </script>
  </head>
  <body>
    <div onmouseover="ratonEncima();"onmouseout="ratonFuera();"> VALOR   POR   DEFECTO </div>
  </body>
</html>
```

Ejemplo de código con DOM

En el código anterior hay dos funciones que modifican el texto que mostramos en la etiqueta div. Estas funciones actúan directamente a través de la estructura DOM, sobre el valor de la etiqueta div. En la etiqueta div se llama a dos eventos, el primero, para que

se accione cuando el ratón esta encima de la etiqueta div. El segundo, para que se accione cuando el ratón no se sitúa encima del texto de la etiqueta div. Cuando la página se carga por primera vez, el texto que aparece, es el texto por defecto.

9. Diferencias en las implementaciones del modelo

Una de las principales dificultades que nos encontramos a la hora de utilizar DOM, es que no todos los navegadores hacen una misma interpretación. W3C hace unas recomendaciones y crea unos estándares, que los navegadores van adoptando, con arreglo a la capacidad que tienen en el desarrollo del navegador. Los navegadores han de respetar los estándares que existían antes y adoptar los nuevos que van saliendo. Solo de esta forma es posible que la gran cantidad de páginas web que existen y no están adaptadas a las recomendaciones W3C puedan sobrevivir entre las que si cumplen los nuevos estándares. Las altas expectativas en generar los estándares, los caminos paralelos, los intereses de las empresas que están detrás de los diferentes navegadores, hacen que esta tarea sea lenta y exija un consenso entre los principales navegadores y la asociación estandarizadora W3C.

La guerra entre los navegadores a la hora de generar sus propios estándares, ha generado muchos problemas a los programadores de páginas web. Todos los navegadores utilizan JavaScript como uno de los lenguajes de programación en el entorno cliente, pero los objetos y eventos no se comportan de la misma forma en todos ellos. Esto obliga a generar diferente código dependiendo del navegador. Otra opción es limitar el uso de una aplicación web a uno o dos navegadores concretos. Los principales navegadores Internet Explorer, Microsoft Edge, Firefox, Google Crome, Opera, Safari. Sus navegadores intentando adaptar la interpretación del código a los estándares. Además, dentro de un mismo navegador existen distintas versiones, de tal forma que unas soportan unos mecanismos y otras no.

9.1. Aplicaciones cross-browser

Cuando hablamos de aplicaciones cross-browser, nos estamos refiriendo a aplicaciones que se vean exactamente igual en cualquier navegador. Los navegadores son desarrollados por diferentes empresas de software, cada una con sus propios intereses y, desde siempre, han sido patentes las diferencias entre unos y otros. El W3C define estándares para HTML, CSS y JavaScript, pero muchas veces estas empresas interpretan el estándar de forma distinta, o incluso, a veces, agregan funcionalidades o etiquetas que no están contempladas ni permitidas en el estándar.

El W3C ha ido mejorando y actualizando los estándares, definiendo nuevos niveles del DOM, y por el otro lado, las empresas desarrolladoras de software también se van adaptando, cada vez más, a los estándares propuestos por el W3C.

La historia de cross-browser comenzó con la "guerra de navegadores" al final de 1990 entre Netscape Navigator y Microsoft Internet Explorer y, por lo tanto, también entre

JavaScript y JScript (los primeros lenguajes de scripting implementados en estos navegadores respectivamente). Netscape Navigator era el navegador web más usado en ese momento, y Microsoft había sacado Mosaic para crear Internet Explorer 1.0. Nuevas versiones de estos navegadores fueron surgiendo rápidamente, y debido a la feroz competencia entre ellos, muchas veces se añadieron características nuevas, sin ningún tipo de coordinación o control entre fabricantes. La introducción de estas nuevas características a menudo tuvo prioridad sobre la corrección de errores, dando como resultado navegadores inestables, bloqueos, navegadores que no cumplen el estándar y fallos de ejecución, llegando incluso a provocar cierres accidentales de las aplicaciones o del navegador.

Durante todo ese tiempo los programadores de páginas web han sido los encargados de ir parcheando estas diferencias para conseguir que sus aplicaciones se ejecuten de la misma forma en unos u otros navegadores, independientemente de la versión o fabricante utilizado. Estas soluciones que se adaptan a cualquier tipo de navegador son las que se conocen como "**soluciones cross-browser**".

Las soluciones cross-browser no sólo se aplican a JavaScript, sino que también se pueden aplicar a otras tecnologías como CSS o, incluso, HTML. Lo que se busca por lo tanto es que, esas incompatibilidades o diferencias entre navegadores no sean apreciables por el cliente, y que la página web o aplicación funcione indistintamente en cualquier navegador sin producir fallos o efectos indeseados.

9.2. Métodos para programar aplicaciones cross-browser

A la hora de realizar aplicaciones multi-cliente con JavaScript deberemos tener en cuenta el tipo de navegador que estamos utilizando para que el código se ejecute correctamente. Por ejemplo, si quisiéramos acceder a los nombres de clases CSS empleados por un determinado elemento, dependiendo de si es IE u otro navegador, tendríamos que usar **className** o **classList** respectivamente:

```
if (navigator.appName.indexOf("Explorer") != -1) // Es un navegador IE
{     // Usaremos className en lugar de classList
   this.parentNode.childNodes[i].className = this.parentNode.childNodes[i].className.replace(/\bseleccionado\b/,'');
}
else     // Es un navegador W3C
   this.parentNode.childNodes[i].classList.remove("seleccionado");
```

Comprobación de navegdor

En JavaScript, podemos ejecutar bloques de código dependiendo de una condición determinada. En nuestro caso en el tema de cross-browsing, podríamos comprobar el tipo de navegador que estamos utilizando para ejecutar nuestro código de JavaScript, y dependiendo de eso, ejecutaríamos el código compatible con ese navegador. Por ejemplo, aquí te muestro una función para crear un evento:

```
function crearEvento(elemento, evento, funcion)
{
  if (typeof elemento.addEventListener !== 'undefined')
  {           // evento compatible con W3C
  }
  else if (typeof elem.attachEvent !== 'undefined')
  {           // evento compatible con Internet Explorer
  }
}
```

Crear evento

La función anterior es una función **cross-browser** muy simplificada para crear eventos. Pero ¿qué pasaría en el siguiente caso?

```
var elementos = document.getElementsByTagName('div');
// supongamos que nos devuelve 5000 divs
var i, longitud = elementos.length;
for (i = 0; i < longitud; i++)
{
  crearEvento(elementos[i], 'click', function()     {
      alert('Saludos !');       });
}
```

Código para crear eventos en etiquetas div

En este caso el navegador estaría comprobando 5000 veces **if (typeof elemento.addEventListener !== 'undefined')** …. - una vez para cada elemento de la colección elementos, lo cual supone una gran pérdida de tiempo. Estaría mejor si, de alguna manera, pudiéramos decirle al navegador: "Cuando sepas si addEventListener() está soportado por este navegador, no continúes comprobando para las otras 4999 iteraciones".

Para evitar el hacer la comprobación que citamos anteriormente 4999 veces, debemos crear funciones separadas que contengan la lógica de cross-browser, y luego envolviendo esas funciones con otra mayor que devuelva la función apropiada a ejecutar. La parte más ingeniosa de este código es que la parte externa del código de la función (la decisión del tipo de navegador que estamos utilizando) será ejecutada solamente una vez, independientemente del número de llamadas que hagamos; eso es, en cierto modo, la "parte condicional de compilación (técnica para mejorar el rendimiento de sistemas de programación, de tal forma que se obtiene código máquina a partir del código fuente con lo que se mejora la ejecución de las aplicaciones. Los compiladores son los encargados de realizar este proceso. El lenguaje JavaScript es un

lenguaje interpretado, no compilado)". Un ejemplo de una función cross-browser para crear eventos podría ser:

```
var crearEvento = function(){
function w3c_crearEvento(elemento, evento, mifuncion)     {
    elemento.addEventListener(evento, mifuncion, false);
}

function ie_crearEvento(elemento, evento, mifuncion)     {
    var fx = function(){
        mifuncion.call(elemento);
};
// Cuando usamos attachEvent dejamos de tener acceso
// al objeto this en mifuncion. Para solucionar eso
// usaremos el método call() del objeto Function, que nos permitirá
// asignar el puntero this para su uso dentro de la función.
// El primer parámetro que pongamos en call será la referencia que
// se usará como objeto this dentro de nuestra función mifuncion. De
// esta manera solucionamos el problema  de acceder a this usando
// attachEvent en Internet Explorer.

elemento.attachEvent('on' + evento, fx);
}
      if (typeof window.addEventListener !== 'undefined'){
            return w3c_crearEvento;
}else if (typeof window.attachEvent !== 'undefined')     {
    return ie_crearEvento;
}
}();     // <= Esta es la parte más crítica - tiene que terminar en ()
```

Ejemplo de creación de eventos.

En este código se ha separado la lógica para IE y Edge y los navegadores W3C. Se han desarrollado dos funciones, una para navegadores Internet Explorer, y otra para compatibles W3C. Según el tipo de navegador se devolverá una u otra función. La parte más crítica está en el uso de }(); al final del código. Es importante darse cuenta lo que está pasando aquí: estamos declarando una función crearEvento, con el código: var crearEvento= function() {, e inmediatamente después, estamos ejecutando esa función al final de su declaración con } (); . De esta forma aunque llamemos a crearEvento múltiples veces en nuestro código, sólo se comprobará el tipo de navegador una sola vez con lo que se acelera la ejecución del código. Puedes comprobar el código de la función que se devolvería en tu navegador con: alert(crearEvento.toString()).