

UD5

Interacción con el usuario. Eventos.

Eventos

Eventos	1
Manejadores de Eventos	1
1. Manejadores como atributos de los elementos en línea	2
2. Manejadores como funciones JavaScript externas.	2
3. Manejadores Semánticos	4
4. Modelo de registro de eventos de la W3C.	5
5. Modelo de registro de eventos de Microsoft.	6
Conclusiones sobre los modelos de datos	¡Error! Marcador no definido.
El flujo de los eventos: captura y burbuja	8
Extraer información de los eventos - reutilización de código	10
Tipos de eventos	11
Eventos de ratón	11
Eventos de teclado	11
Eventos HTML	12
Eventos DOM	12

Eventos

¿Qué es un evento? Podemos definirlo como aquellos “**mecanismos**” que se accionan cuando el usuario realiza un cambio sobre la página web. Cada vez que se produce un evento, desde javascript debemos “capturar dicho evento” y programar una acción para que se lleve a cabo una tarea.

El encargado de gestionar los eventos es el DOM (Document Object Model).

El **nombre del evento** se forma con la palabra “**on**” seguido del **nombre de la acción**. En la página de la W3Schools podemos ver un buen resumen de los eventos relacionados con los elementos HTML:

https://www.w3schools.com/tags/ref_eventattributes.asp

Manejadores de Eventos

¿Qué es un manejador? Acción que se va a manejar. Por ejemplo, cuando el usuario hace click sobre algún elemento HTML **el manejador sería el código encargado de manejar el evento** “onClick”.

Las funciones o código JavaScript que se definen para cada evento se denominan **manejador de eventos** (event handlers en inglés) y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores, clasificados comúnmente en 5 categorías:

1. Manejadores como atributos de los elementos en línea.
2. Manejadores como funciones JavaScript externas.
3. Manejadores "semánticos".
4. Modelo de registro de eventos de la W3C.
5. Modelo de registro de eventos de Microsoft.

1. Manejadores como atributos de los elementos en línea

Se trata del método **más sencillo** y a la vez **menos profesional** de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento. En este caso, **el código se incluye en un atributo del propio elemento HTML**. Veamos algunos ejemplos:

- Incluir varios manejadores para varios eventos sobre un mismo elemento HTML:

```
<h3 id="txt1" onclick="this.innerHTML='Javascript en XHTML'"
      onmouseover="this.style.background='red'"
      onmouseout="this.style.background='white'">Pulsa aquí para ver lo que
se ejecuta</h3>
```

En el código anterior hemos aplicado una serie de acciones para cada uno de los 3 eventos que se manejarán sobre una etiqueta de texto (cuando se hace click, se cambia el texto por otro, y si se pasa el ratón por encima o pierde el foco se cambia el color del background de la etiqueta).

Importante: Variable this

JavaScript define una variable especial llamada **“this”** que se crea automáticamente. En los eventos, se puede utilizar la variable **“this” para referirse al elemento HTML que ha provocado el evento**. Si no se utilizara la variable **“this”** tendríamos que referenciar el elemento HTML de alguna otra forma como por ejemplo **“document.getElementById('txt1')”** en nuestro ejemplo anterior.

Resumiendo: el principal inconveniente de los manejadores como atributos de elementos en línea es que el código HTML se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para casos sencillos y puntuales.

2. Manejadores como funciones JavaScript externas.

Cuando el código de la función manejadora es más complejo, como por ejemplo la validación de un formulario, es aconsejable **agrupar todo el código JavaScript en una función externa** que se invoca desde el código HTML cuando se produce el evento. Por ejemplo, el código incluido en línea:

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

Se puede transformar en la siguiente composición de código HTML y Javascript:

```
<input type="button" value="Píñchame y verás" onclick="muestraMensaje()" />
<script>
function muestraMensaje() {
    console.log('Gracias por pinchar');
}
</script>
```

En las funciones externas no es posible utilizar la variable **this** de la misma forma que en los manejadores insertados en los atributos HTML. Por tanto, **es necesario pasar la variable “this” como parámetro a la función manejadora**:

```
<h3 onclick="cambiar(this)">Pulsa aquí para ver qué se ejecuta</h3>
<script>
    function cambiar(elem) {
        elem.innerHTML = "Javascript en HTML y función externa";
    }
</script>
```

Hay **acciones que desencadenan varios eventos**. Por ejemplo, cada vez que en apretamos un botón esto desencadena una serie de eventos como: onmousedown, onclick, onmouseup, onsubmit, o que el propio objeto desencadena una acción por defecto. **Para evitar que el navegador ejecute la acción por defecto necesitamos añadir "return false;"** Veámoslo con el siguiente ejemplo el cual cuando se hace click en el enlace, debemos controlar que no se dirija al enlace correspondiente sin haber ejecutado correctamente la función javascript:

```
<a href="http://www.google.com" onclick="alertar(); return false;">Pulsa aquí para
ver qué se ejecuta</a>
<script>
    function alertar() {
        alert("Vamos a Google");
    }
</script>
```

Pregunta: ¿Qué pasaría si en vez de hacer un “return false;” hiciésemos un “return true;”?

Es más, si quisiéramos controlar mediante una ventana de confirmación si nos dirigimos al enlace o no podríamos hacerlo de la siguiente forma:

```
<a href="http://www.google.com" onclick="return preguntar();">Pulsa aquí para ver
qué se ejecuta</a>
<script>
    function preguntar() {
        return confirm("¿Deseas ir a Google?");
    }
</script>
```

3. Manejadores Semánticos

Como es conocido, al crear páginas web se recomienda separar los contenidos HTML de la presentación (CSS). En lo posible, también se recomienda separar los contenidos HTML de la lógica (JavaScript). Mezclar JavaScript y HTML complica excesivamente el código fuente de la página, dificulta su mantenimiento y reduce la semántica del documento final producido.

Existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica consiste en asignar las funciones externas a los eventos de los elementos HTML en el código javascript, haciendo de alguna forma (porque hay varias) referencia a los elementos HTML a los que se le debe asignar estos eventos. Una solución es separar completamente el código HTML del código JavaScript. Para ello debemos ejecutar algún método para acceder al elemento que queremos manejar desde el código. Veamos un ejemplo:

```
<h3 id="tradicional">Pulsa aquí para ver lo que se ejecuta</h3>
<script>
  //Accedemos al elemento y a su evento onclick le asignamos una función
  document.getElementById("tradicional").onclick = cambiar; //¡¡Sin paréntesis!!
  function cambiar() {
    alert("Entramos en cambiar");
    document.getElementById("tradicional").innerHTML = "Modelo Semántico";
    //Hace que solo funcione una vez ya que desactiva el evento poniéndolo a null.
    document.getElementById("tradicional").onclick = null;
  }
</script>
```

Inconveniente: Esta forma de acceder a los eventos de los elementos HTML tiene un gran inconveniente y es que para acceder a un elemento de esta forma “document.getElementById(“tradicional”)” el árbol DOM tiene que haber sido construido por completo y esto se hace una vez que solo se haya cargado la página al completo. Esto nos puede dar algún problema ya que en mitad de la carga de la página intentamos acceder a estos elementos y aún son nulos.

Una forma de solventar este problemilla es que el código que deseamos se ejecute una vez el árbol DOM se haya construido y para esto podemos usar la función **window.onload** de tal forma que el código anterior quedaría de la siguiente forma:

```
<h3 id="tradicional">Pulsa aquí para ver lo que se ejecuta</h3>
<script>
  window.onload = function {
    document.getElementById("tradicional").onclick = cambiar; //¡¡Sin
paréntesis!!
    function cambiar() {
      alert("Entramos en cambiar");
      document.getElementById("tradicional").innerHTML = "Modelo Semántico";
      document.getElementById("tradicional").onclick = null;
    }
  }
</script>
```

```

        //window.onload = cargaEventos();
        //cargaEventos = function () { ----- }
    }
</script>

```

→ **window.onload = function { }** : Nos asegura que el código ejecutado entre sus llaves será ejecutado una vez que la página se haya cargado por completo.

4. Modelo de registro de eventos de la W3C.

Este modelo es el más utilizado a día de hoy y soportado por todos los navegadores con más cuota de mercado como Chrome, Firefox, Opera y Edge. La sintaxis básica para añadir un manejador a un evento indicado se realiza con la sintaxis:

elemento.addEventListener("<evento_sin_on>",<función>,<false|true>);

y podemos eliminar o cancelar el evento con el siguiente método:

elemento.removeEventListener("<evento_sin_on>",<función>);

IMPORTANTE: Si te has fijado el método **addEventListener** tiene como tercer parámetro un booleano el cual está relacionado con la fase de captura y burbujeo la cual está explicada aquí: [Flujo de Eventos](#)

Veamos un ejemplo:

```

<h1>Modelo de eventos avanzados del W3C</h1>
<h3 id="w3c">Modelo del W3C</h3>
<script>

    document.getElementById("w3c").addEventListener("click", saludarUnaVez,
false);
    document.getElementById("w3c").addEventListener("click", colorearse, false);
    document.getElementById("w3c").addEventListener("mouseover",fondo, false);
    function saludarUnaVez() {
        alert("¡Hola, caracola!");
        document.getElementById("w3c").removeEventListener("click",
saludarUnaVez);
    }
    function colorearse() {
        document.getElementById("w3c").style.color = "red";
    }
    function fondo() {
        document.getElementById("w3c").style.background = "blue";
    }
</script>

```

Este modelo también nos permite aplicar **funciones anónimas** las cuales son eventos que no tienen un nombre determinado:

```
<h3 id="w3canonima">Modelo del W3C con funciones anónimas</h3>
<script>
    document.getElementById("w3canonima").addEventListener("click", function () {
        this.style.background = "#C0C0C0";
    }, false);
</script>
```

Una posibilidad muy importante que nos ofrecen las funciones anónimas es **la posibilidad de llamar a funciones a los que tengamos que pasarle parámetros**:

```
<h3 id="msanonima">Modelo de W3C</h3>
<script>
    document.getElementById("msanonima").addEventListener("click", function () {
        funcionConParametros( parametro1 );
    });
    function funcionConParametros ( parametro1 ) { //Hacemos algo };
</script>
```

Como vemos una de las **ventajas** que ofrece este modelo es que **podemos añadir todos los manejadores que deseemos a los mismos eventos de los mismos elementos HTML**.

5. Modelo de registro de eventos de Microsoft.

Hemos visto que con el modelo de registro de eventos de W3C podemos dar soporte para navegadores como FireFox, Chrome, Opera y las versiones más novedosas de Microsoft como Edge. Pero si quisiéramos dar soporte y manejar eventos correctamente en navegadores Microsoft de versiones más antiguas deberíamos usar este modelo de eventos. Es muy parecido al W3C pero usando unos métodos diferentes.

La sintaxis básica de este modelo para añadir un manejador al evento es:

elemento.attachEvent("<evento_con_on>",función);

y podemos eliminarlo con:

elemento.detachEvent("<evento_con_on>, función")

Lo vemos con un ejemplo:

```
<h3 id="ms">Modelo de Microsoft</h3>
<script>
    document.getElementById("ms").attachEvent("onclick", saludarUnaVez);
    document.getElementById("ms").attachEvent("onclick", colorearse);
    document.getElementById("ms").attachEvent("onmouseover", fondo);
    function saludaUnaVez() {
        alert("¡Hola, caracola!");
        document.getElementById("ms").detachEvent("onclick", saludarUnaVez);
    }
</script>
```

```

    }
    function colorearse() {
        document.getElementById("ms").style.color = "red";
    }
    function fondo() {
        document.getElementById("ms").style.backgroundColor = "blue";
    }
}
</script>

```

También como hacíamos en el modelo W3C podemos añadir una **función anónima** de la siguiente forma:

elemento.attachEvent("<evento_con_on>",function(){...});

```

<h3 id="msanonima">Modelo de Microsoft</h3>
<script>
    /* Funciones anónimas */
    document.getElementById("msanonima").attachEvent("onclick", function () {
        this.style.backgroundColor = "#C0C0C0";
    });
</script>

```

Una posibilidad muy importante que nos ofrecen las funciones anónimas es la posibilidad de llamar a funciones a los que tengamos que pasarle parámetros:

```

<h3 id="msanonima">Modelo de Microsoft</h3>
<script>
    document.getElementById("msanonima").attachEvent("click", function () {
        funcionConParametros( parametro1 );
    });
    function funcionConParametros ( parametro1 ) { //Hacemos algo };
</script>

```

Conclusiones sobre los modelos de datos

La pregunta del millón después de haber visto estos 5 tipos/modelos de manejar eventos sería: ¿Y qué modelo debo de usar? Pues la respuesta es simple:

- Si es para algo muy sencillo, en una página muy simple, ya sea por ejemplo un popup de nuestra página principal, podríamos usar los manejadores en línea.
- Para soluciones o webs más complejas o grandes proyectos web, lo mejor es tener las funciones almacenadas en javascripts externos, bien organizados, y utilizar el modelo recomendado por la W3C.
- Si además quisiéramos dar soporte a antiguos navegadores de Internet Explorer deberíamos de combinar el modelo de W3C con el de Microsoft de la siguiente manera:

Comprobar si existe el **addEventListener**; y, en caso contrario, es que estás utilizando Explorer anterior a la versión 9, y utiliza **attachEvent**. Te dejo una posible solución:

```
<script>
var x = document.getElementById("elementoHTML");
if (x.addEventListener) { // Si existe el AddEventListener, es decir, para la
    mayoría de los exploradores
    x.addEventListener("click", miFuncion, false); }
else if (x.attachEvent) { // Si no existe, es que hablamos de Explorer anterior a
    9, y usa attachEvent
    x.attachEvent("onclick", miFuncion); }
</script>
```

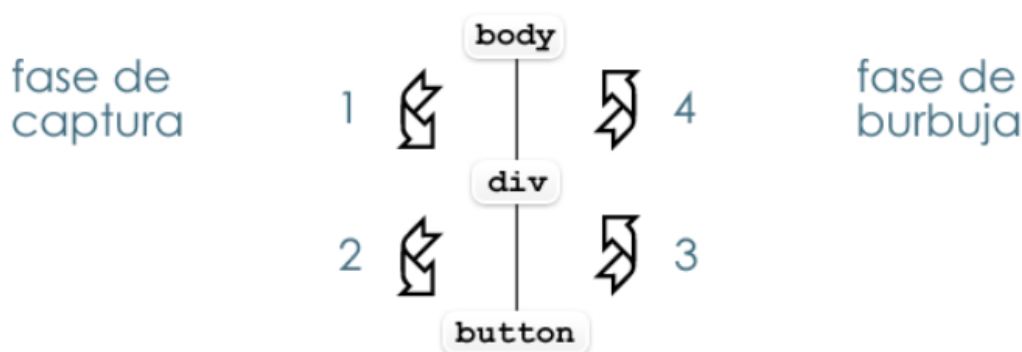
El flujo de los eventos: captura y burbuja

Como hemos visto, hay un parámetro el **addEventListener** que es un **booleano**. ¿Para qué sirve? Bueno, para entenderlo primero hemos de saber lo que es el **flujo de eventos**. Supongamos que tenemos este código HTML:

```
<body>
  <div>
    <button>HAZME CLICK</button>
  </div>
</body>
```

Cuando hacemos clic en el botón no sólo lo estamos haciendo sobre él, sino sobre los elementos que lo contienen en el árbol del DOM, es decir, hemos hecho clic además sobre el elemento “body” y sobre el elemento “div”. Si sólo hay una función asignada a una escucha para el botón no hay mayor problema, pero si además hay una para el body y otra para el div, **¿cuál es el orden en que se deben lanzar las tres funciones?**

Para contestar a esa pregunta existe un modelo de comportamiento, [el flujo de eventos \(inglés\)](#). Según éste, cuando se hace clic sobre un elemento, el evento se propaga en **dos fases**, una que es la **captura (el evento comienza en el nivel superior del documento y recorre los elementos de padres a hijos)** y la otra la **burbuja (el orden inverso, ascendiendo de hijos a padres)**. En un diagrama sería algo así:



Así, el orden por defecto de lanzamiento de las supuestas funciones sería **body-div-button**. Una vez visto esto, podemos comprender el tercer parámetro de **addEventListener**, que lo que hace es permitirnos escoger el orden de propagación:

- **true**: El flujo de eventos es como el representado, y la fase de captura ocurre al lanzarse el evento. El orden de propagación para el ejemplo sería, por tanto, el indicado, **body-div-button**.
- **false**: Permite saltar la fase de captura, y la propagación seguiría sólo la burbuja. Así, el orden sería **button-div-body**.

Con un ejemplo podemos ver más claro el comportamiento descrito:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Captura y burbuja</title>
  </head>

  <body>
    <div id="capt1">
      <span id="capt2">Pulsa aquí para la fase de captura.</span>
    </div>

    <div id="burbuja1">
      <span id="burbuja2">Pulsa aquí para la fase de burbuja.</span>
    </div>

    <script>
      var capt1 = document.getElementById('capt1'),
          capt2 = document.getElementById('capt2'),
          burbuja1 = document.getElementById('burbuja1'),
          burbuja2 = document.getElementById('burbuja2');

      capt1.addEventListener('click', function() {
        alert("El evento de div se acaba de desencadenar.");
      }, true);

      capt2.addEventListener('click', function() {
        alert("El evento de span se acaba de desencadenar.");
      }, true);

      burbuja1.addEventListener('click', function() {
        alert("El evento de div se acaba de desencadenar.");
      }, false);

      burbuja2.addEventListener('click', function() {
        alert("El evento de span se acaba de desencadenar.");
      }, false);
    </script>
  </body>
```

</html>

Extraer información de los eventos - reutilización de código

Cuando trabajamos con eventos y manejadores nos resultará fundamental el poder identificar el tipo de evento producido o incluso qué elemento ha sido el generador del evento, entre otros datos, y así tener la posibilidad de poder llevar a cabo la reutilización de funciones de nuestro código javascript. En otras palabras, hay veces que nos interesa reusar funciones para diferentes eventos producidos y/o para diferentes elementos generadores, por lo que a la hora de realizar la acción existen mecanismos para obtener quién ha provocado dicho evento o el tipo de evento generado.

La propiedad **"window.event"** nos resultará fundamental ya que gracias a esta podemos obtener información sobre los eventos, y entre las propiedades que podemos usar destacaremos 2:

- **window.event.type:** nos da información sobre el tipo de evento generado (onclick, mouseover, etc).
- **window.event.target.id:** nos devuelve el "id" del elemento HTML que ha generado el evento.

En el siguiente enlace podemos ver todas las propiedades y métodos del objeto **window.event**:

[HTML DOM Event Properties and Methods](#)

Veamos un ejemplo de su utilización:

```
<h1 id="eventos">Obtener información de un evento</h1>
<h2 id="parrafo1">Párrafo 1</h2>
<h2 id="parrafo2">Párrafo 2</h2>
<script>
    //Asignamos misma función a diferentes tipos de eventos:
    document.getElementById("eventos").addEventListener("mouseover", manejador);
    document.getElementById("eventos").addEventListener("mouseout", manejador);
    //Asignamos misma función a mismo eventos de elementos HTML diferentes:
    document.getElementById("parrafo1").addEventListener("click", saludo);
    document.getElementById("parrafo2").addEventListener("click", saludo);

    function manejador(e) {
        //Obtenemos el window.event
        if (!e)
            e = window.event;
        switch (e.type) { //Obtenemos el tipo de evento, y base, actuamos:
            case "mouseover":
                this.style.color = "purple";
                break;
            case "mouseout":
                this.style.color = "yellow";
                break;
        }
    }

    function saludo(e) {
        //Obtenemos el window.event con el fin de obtener el target.id
        if (!e) e = window.event;
```

```
        if (e.target.id == "parrafo1")
            alert("Has pulsado el primer párrafo");
        else if (e.target.id == "parrafo2")
            alert("Has pulsado el segundo párrafo");

        alert("Has pulsado el " + e.target.id);
    }
</script>
```

Tipos de eventos

La lista completa de eventos que se pueden generar en un navegador se puede dividir en cuatro grandes grupos. La especificación de DOM define los siguientes grupos:

- **Eventos de ratón:** se originan cuando el usuario emplea el ratón para realizar algunas acciones.
- **Eventos de teclado:** se originan cuando el usuario pulsa sobre cualquier tecla de su teclado.
- **Eventos HTML:** se originan cuando se producen cambios en la ventana del navegador o cuando se producen ciertas interacciones entre el cliente y el servidor.
- **Eventos DOM:** se originan cuando se produce un cambio en la estructura DOM de la página. También se denominan "eventos de mutación".

Eventos de ratón

Los eventos de ratón son, con mucha diferencia, los más empleados en las aplicaciones web. Los eventos que se incluyen en esta clasificación son los siguientes:

- **click:** Se produce cuando se pulsa el botón izquierdo del ratón. También se produce cuando el foco de la aplicación está situado en un botón y se pulsa la tecla ENTER.
- **dblclick:** Se produce cuando se pulsa dos veces el botón izquierdo del ratón.
- **mousedown:** Se produce cuando se pulsa cualquier botón del ratón.
- **mouseout:** Se produce cuando el puntero del ratón se encuentra en el interior de un elemento y el usuario mueve el puntero a un lugar fuera de ese elemento.
- **mouseover:** Se produce cuando el puntero del ratón se encuentra fuera de un elemento y el usuario mueve el puntero hacia un lugar en el interior del elemento.
- **mouseup:** Se produce cuando se suelta cualquier botón del ratón que haya sido pulsado.
- **mousemove:** Se produce (de forma continua) cuando el puntero del ratón se encuentra sobre un elemento.

Eventos de teclado

Los eventos que se incluyen en esta clasificación son los siguientes:

- **keydown:** Se produce cuando se pulsa cualquier tecla del teclado. También se produce de forma continua si se mantiene pulsada la tecla.
- **keypress:** Se produce cuando se pulsa una tecla correspondiente a un carácter alfanumérico (no se tienen en cuenta teclas como SHIFT, ALT, etc.). También se produce de forma continua si se mantiene pulsada la tecla.
- **keyup:** Se produce cuando se suelta cualquier tecla pulsada.

Eventos HTML

Los eventos HTML definidos se recogen en la siguiente tabla:

- **load:** Se produce en el objeto window cuando la página se carga por completo. En el elemento cuando se carga por completo la imagen. En el elemento <object> cuando se carga el objeto.
- **unload:** Se produce en el objeto window cuando la página desaparece por completo (al cerrar la ventana del navegador por ejemplo). En el elemento <object> cuando desaparece el objeto.
- **abort:** Se produce en un elemento <object> cuando el usuario detiene la descarga del elemento antes de que haya terminado.
- **error:** Se produce en el objeto window cuando se produce un error de JavaScript. En el elemento cuando la imagen no se ha podido cargar por completo y en el elemento <object> cuando el elemento no se carga correctamente.
- **select:** Se produce cuando se seleccionan varios caracteres de un cuadro de texto (<input> y <textarea>).
- **change:** Se produce cuando un cuadro de texto (<input> y <textarea>) pierde el foco y su contenido ha variado. También se produce cuando varía el valor de un elemento <select>.
- **submit:** Se produce cuando se pulsa sobre un botón de tipo submit (<input type="submit">).
- **reset:** Se produce cuando se pulsa sobre un botón de tipo reset (<input type="reset">).
- **resize:** Se produce en el objeto window cuando se redimensiona la ventana del navegador.
- **scroll:** Se produce en cualquier elemento que tenga una barra de scroll, cuando el usuario la utiliza. El elemento <body> contiene la barra de scroll de la página completa.
- **focus:** Se produce en cualquier elemento (incluido el objeto window) cuando el elemento obtiene el foco.
- **blur:** Se produce en cualquier elemento (incluido el objeto window) cuando el elemento pierde el foco.

Eventos DOM

<https://youtu.be/Mq8Pp8Dqy5Y> → **MutationObserver**

<https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>

<https://developer.mozilla.org/en-US/docs/Web/API/MutationRecord>

Aunque los eventos de este tipo son parte de la especificación oficial de DOM, aún no han sido implementados en todos los navegadores. La siguiente tabla recoge los eventos más importantes de este tipo:

Importante: Estos métodos han sido “deprecated” en las últimas versiones de navegadores. En su defecto se usa un objeto llamado “MutationEvent”. Debido a su complejidad de uso NO vamos a abarcarlo en nuestro temario. Pero que conozcáis su existencia.

- **DOMSubtreeModified:** Se produce cuando se añaden o eliminan nodos en el subárbol de un documento o elemento
- **DOMNodeInserted:** Se produce cuando se añade un nodo como hijo de otro nodo
- **DOMNodeRemoved:** Se produce cuando se elimina un nodo que es hijo de otro nodo
- **DOMNodeRemovedFromDocument:** Se produce cuando se elimina un nodo del documento
- **DOMNodeInsertedIntoDocument:** Se produce cuando se añade un nodo al documento.