

Modelo de objetos del documento en javascript.

Caso práctico

Antonio está haciendo muchos avances en su proyecto. Por ejemplo, ya ha realizado casi toda la validación de los formularios de las páginas web empleando JavaScript, y les ha añadido mensajes de errores de forma dinámica. Ahora se encuentra con un pequeño problema: necesita poder acceder a algunas partes de su documento y modificar contenidos, pero no sabe cómo hacerlo. Le interesaría hacer cosas, como borrar celdas en tablas, añadir o modificar atributos a elementos, modificar contenido textual de cualquier parte del documento, etc. y todo ello usando JavaScript.



Juan le informa que todo eso que solicita se puede hacer con JavaScript, pero tendrá que hacer uso del DOM de una forma más intensiva. El DOM organiza todo el documento en una especie de árbol de elementos, de tal forma que, a través de JavaScript, podrá acceder a cada uno de esos nodos y modificar su contenido, añadir nuevos nodos, eliminarlos, recorrerlos, etc. Cualquier cambio que realice en el árbol de nodos, es reflejado de forma automática por el navegador web, con lo que las modificaciones son instantáneas de cara al cliente.

Por último, **Antonio** pregunta si es posible detectar qué botones del ratón han sido pulsados, o si se está utilizando una combinación de teclas en el teclado, ya que, por ejemplo, una de las cosas que quiere hacer en sus formularios es que, cuando se esté dentro de un campo de texto y se pulse **Enter** se pase automáticamente al siguiente campo, o que cuando se pulse una combinación de teclas determinada, se realice una tarea que tenga programada en JavaScript.

Juan le responde que para hacer eso tiene que profundizar un poco más en los eventos y, en especial, en el objeto **Event**, que le permite acceder a nuevas propiedades que le proporcionarán esa información específica que busca. **Juan** además puntualiza que, ahora que se mete de lleno en eventos más específicos, tendrá que tener en cuenta las incompatibilidades entre navegadores, para que sus aplicaciones sean multi-cliente. Para conseguirlo le da unas indicaciones de cómo tiene que programar los eventos, y qué diferencias va a encontrar entre los distintos navegadores.

En esta unidad de trabajo se hace una introducción al Modelo de Objetos del Documento y los diferentes niveles propuestos por el W3C. A continuación se profundiza en el árbol del DOM viendo lo que son los nodos del árbol, cómo acceder a ellos, a los nodos de tipo texto y los nodos de tipo atributo, para terminar viendo cómo crear nuevos nodos o borrar los nodos ya existentes.

En el apartado 2 se verá la gestión de eventos en JavaScript. Se analizará el modelo de eventos, qué tipos de eventos hay y se terminará dicha sección viendo el objeto `Event`, sus propiedades y métodos más importantes y los eventos de teclado y de ratón, ya que éstos son los eventos más utilizados y más complejos de usar en JavaScript.



[Ministerio de Educación y Formación Profesional](#) (Dominio público)

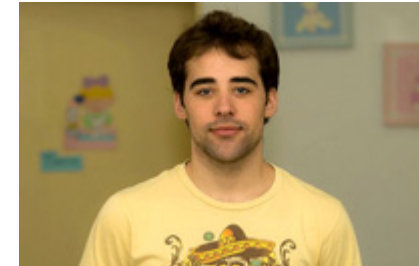
Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](#)

1.- Bases del Modelo de Objetos del Documento (DOM).

Caso práctico

El estudio más a fondo del DOM, va a permitir a **Antonio** llegar a conocer con muchísimo detalle cómo se construye una página web, ya que el DOM es la base de toda la estructura de cualquier documento. El conocer a fondo el DOM, qué tipos de nodos contiene, cómo acceder a ellos para recorrerlos, modificarlos o borrarlos, y ver las diferentes aproximaciones según los navegadores, supondrá un salto cualitativo en su programación con JavaScript. De esta forma, prácticamente cualquier cosa que se proponga dejará de tener secretos, porque prácticamente todo lo que se vea en la página Web, va a estar accesible a través de JavaScript empleando las instrucciones de manejo del DOM.



El DOM (Modelo de Objetos del Documento), es esencialmente una interfaz de programación de aplicaciones (API), que proporciona un conjunto estándar de objetos, para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML, que es para lo que se diseñó principalmente. El responsable del DOM es el W3C.

El DOM está separado en 3 partes / niveles:

- ✓ DOM Core – modelo estándar para cualquier documento estructurado.
- ✓ DOM XML – modelo estándar para documentos XML.
- ✓ DOM HTML – modelo estándar para documentos HTML.

EL DOM HTML es un estándar dónde se define cómo acceder, modificar, añadir o borrar elementos HTML.

En el DOM se definen los **objetos y propiedades** de todos los elementos del documento, y los métodos para acceder a ellos.

Es importante citar que en el DOM del W3C, no se especifican todas las características especiales de los modelos de objeto de exploración. Muchas de las funciones de Internet Explorer 4 (y posteriores) del modelo de objetos, no forman parte de la especificación DOM W3C.

Debes tener en cuenta que, mientras que los navegadores basados en Mozilla están haciendo grandes esfuerzos para poner en práctica todos los niveles del DOM 1 y la mayoría de Nivel 2 del W3C, Microsoft (por la razón que sea) sólo realiza una aplicación parcial del DOM a sus navegadores, aunque con las versiones más modernas se están adaptando poco a poco al estándar. Otros navegadores modernos como Chrome, Safari, Opera, soportan de forma extensiva el DOM del W3C.

Niveles del DOM

Al igual que otras muchas especificaciones del W3C, una versión no suele ser suficiente, por lo que la especificación del DOM sigue el mismo camino. El DOM está en continua evolución. Las fechas propuestas de las diferentes versiones aportadas por el W3C, raramente coinciden con las versiones de los navegadores. Por lo que suele ser muy común que muchas versiones de los navegadores incluyan solamente, algunos detalles de las versiones más recientes del W3C. La primera especificación DOM nivel 1, fue liberada después de Netscape 4 e Internet Explorer 4. La parte de HTML cubierta por la especificación de nivel 1 incluye el llamado DOM de nivel 0 (aunque no hay ningún estándar publicado con ese nombre). Esta especificación es esencialmente el modelo de objetos implementado en Navigator 3 (y en parte de Internet Explorer 3 incluyendo el objeto image). Quizás la parte omitida que podamos destacar de este modelo de nivel 1, ha sido una especificación del modelo de eventos.

El DOM de nivel 2 trabaja sobre los desarrollos de nivel 1. Se han añadido nuevas secciones, estilos, formas de inspección de la jerarquía del documento, y se han publicado como módulos separados. Algunos módulos del nivel 3 del DOM han alcanzado el estado de "Recomendación". Aunque Internet Explorer sigue sin implementar una gran mayoría de opciones de los módulos, otros navegadores sí que implementan algunos de los módulos, incluso de los que están en estado de "Borrador".

1.1.- Objetos del DOM HTML, propiedades y métodos.

En otras unidades ya has trabajado con muchos de los objetos que te presentamos en esta lista. Aquí se muestra la referencia completa de objetos, que puedes encontrar en el Modelo de Objetos del Documento para HTML. Y, al final de la lista, hay un hiperenlace que debes visitar para ampliar información sobre las propiedades y métodos de aquellos objetos que no hayas utilizado hasta este momento.

Te recuerdo aquí la sintaxis para acceder a las propiedades o métodos de aquellos objetos que estén dentro de nuestro documento:

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

NOTA : Los datos entre corchetes son opcionales.

Listado de objetos del DOM en HTML:

- ✓ Document
- ✓ HTMLElement
- ✓ Anchor
- ✓ Area
- ✓ Base
- ✓ Body
- ✓ Button
- ✓ Event
- ✓ Form
- ✓ Frame/IFrame
- ✓ Frameset
- ✓ Image
- ✓ Input Button
- ✓ Input Checkbox
- ✓ Input File
- ✓ Input Hidden
- ✓ Input Password
- ✓ Input Radio



[GDJ](#) (Pixabay License)

- ✓ Input Reset
- ✓ Input Submit
- ✓ Input Text
- ✓ Link
- ✓ Meta
- ✓ Object
- ✓ Option
- ✓ Select
- ✓ Style
- ✓ Table
- ✓ TableCell
- ✓ TableRow
- ✓ Textarea

Debes conocer

En el siguiente enlace puedes ampliar detalles sobre el modelo de objetos del documento.

[Introducción al DOM.](#)

Citas para pensar

"Por muy alto que sea un árbol, sus hojas siempre caen hacia la raíz."

Anónimo

1.2.- El árbol del DOM y tipos de nodos.

La tarea más habitual en programación web suele ser la manipulación de páginas web, para acceder a su contenido, crear nuevos elementos, hacer animaciones, modificar valores, etc.

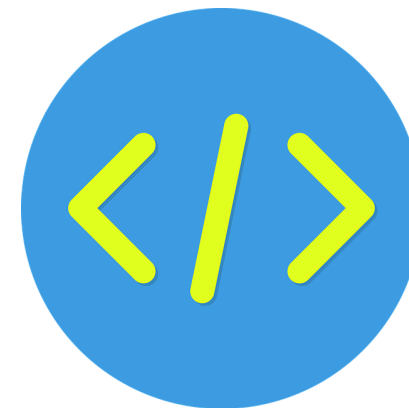
Todas estas tareas se pueden realizar de una forma más sencilla gracias al DOM. Los navegadores web son los encargados de realizar la transformación de nuestro documento, en una estructura jerárquica de objetos, para que podamos acceder con métodos más estructurados a su contenido.

El DOM transforma todos los documentos HTML en un conjunto de elementos, a los que llama **nodos**. En el HTML DOM **cada nodo es un objeto**. Estos nodos están conectados entre sí y representan los contenidos de la página web, y la relación que hay entre ellos. Cuando unimos todos estos nodos de forma jerárquica, obtenemos una estructura similar a un árbol, por lo que muchas veces se suele referenciar como árbol DOM, "**árbol de nodos**", etc.

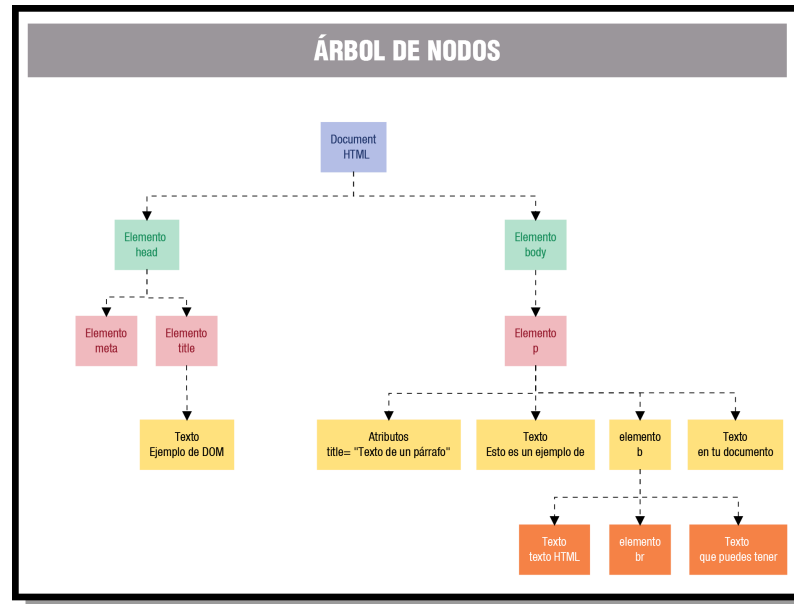
Veamos el siguiente ejemplo de código:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Ejemplo de DOM</title>
  </head>
  <body>
    <p title="Texto de un párrafo">Esto es un ejemplo de <b>texto HTML <br />que puedes tener</b> en tu documento.</p>
  </body>
</html>
```

Ese código se transformaría en el siguiente árbol de nodos:



[janjf93](#) (Pixabay License)



Cada rectángulo del gráfico representa un nodo del DOM, y las líneas indican cómo se relacionan los nodos entre sí. La raíz del árbol de nodos es un nodo especial, denominado "document". A partir de ese nodo, cada etiqueta HTML se transformará en nodos de tipo "**elemento**" o "**texto**". Los nodos de tipo "texto", contendrán el texto encerrado para esa etiqueta HTML. Esta conversión se realiza de forma jerárquica. El nodo inmediatamente superior será el **nodo padre** y todos los nodos que están por debajo serán **nodos hijos**.

Tipos de nodos.

La especificación del DOM define 12 tipos de nodos, aunque generalmente nosotros emplearemos solamente cuatro o cinco tipos de nodos:

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

Los otros tipos de nodos pueden ser: **CdataSection**, **DocumentFragment**, **DocumentType**, **EntityReference**, **Entity**, **Notation** Y **ProcessingInstruction**.

1.3.- Acceso a los nodos.

Cuando ya se ha construido automáticamente el árbol de nodos del DOM, ya podemos comenzar a utilizar sus funciones para acceder a cualquier nodo del árbol. El acceder a un nodo del árbol, es lo equivalente a acceder a un fragmento de la página de nuestro documento. Así que, una vez que hemos accedido a esa parte del documento, ya podemos modificar valores, crear y añadir nuevos elementos, moverlos de sitio, etc.

Para acceder a un nodo específico lo podemos hacer empleando dos métodos: o bien a través de los nodos padre, o bien usando un método de acceso directo. A través de los nodos padre partiremos del nodo raíz e iremos accediendo a los nodos hijo, y así sucesivamente hasta llegar al elemento que deseemos. Y para el método de acceso directo, que por cierto es el método más utilizado, emplearemos funciones del DOM, que nos permiten ir directamente a un elemento sin tener que atravesar nodo a nodo.

Algo muy importante que tenemos que destacar es, que para que podamos acceder a todos los nodos de un árbol, el árbol tiene que estar completamente construido, es decir, cuando la página haya sido cargada por completo, en ese momento es cuando podremos acceder a cualquier elemento de dicha página.

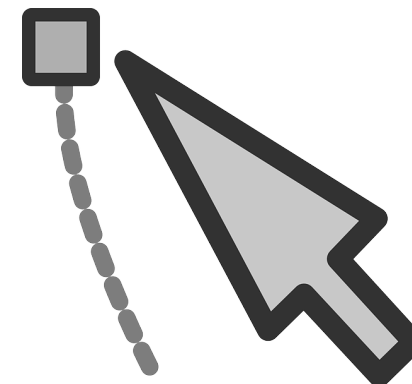
Consideremos el siguiente ejemplo y veamos las formas de acceso:

```
<input type="text" id="apellidos" name="apellidos" />
```

getElementsByTagName()

Esta función obtiene una colección, que contiene todos los elementos de la página cuyo atributo `name` coincida con el indicado como parámetro.

```
let elementos = document.getElementsByTagName("apellidos");
```



[Clicer-Free-Vector-Images](#) ([Pixabay License](#))

Una colección no es un array, aunque se le parezca mucho, ya que aunque puedas recorrerla y referenciar a sus elementos como un array, no se pueden usar métodos de array, como `push` o `pop`, en la colección.

Si sólo tenemos un elemento con `name="apellidos"` para acceder a él haremos:

```
let elemento = document.getElementsByName("apellidos")[0];
```

Por ejemplo, si tuviéramos 3 elementos con el atributo `name="apellidos"` para acceder al segundo elemento haríamos:

```
let segundo = document.getElementsByName("apellidos")[1]; // recordarte que los arrays comienzan en la posición 0.
```

`getElementsByTagName()`

Esta función es muy similar a la anterior y también devuelve una colección de elementos cuya etiqueta coincida con la que se pasa como parámetro. Por ejemplo:

```
let elementos = document.getElementsByTagName("input");  
let cuarto = document.getElementsByTagName("input")[3];
```

`getElementById()`

Esta función es la más utilizada, ya que nos permite acceder directamente al elemento por el ID. Entre paréntesis escribiremos la cadena de texto con el ID. Es muy importante que el ID sea único para cada elemento de una misma página. La función nos devolverá únicamente el nodo buscado. Por ejemplo:

```
let elemento= document.getElementById("apellidos");
```

Si tenemos por ejemplo una tabla con `id="datos"` y queremos acceder a todas las celdas de esa tabla, tendríamos que combinar `getElementById` con `getElementsByTagName`. Por ejemplo:

```
let miTabla= document.getElementById("datos");  
let filas= miTabla.getElementsByTagName("td");
```

1.4.- Acceso a los nodos de tipo atributo.

Una vez que ya hemos visto cómo acceder a los nodos en un documento, vamos a ver cómo podemos acceder a los nodos de tipo atributo. Para referenciar un atributo, como por ejemplo el atributo `type="text"` del campo "apellidos", emplearemos la colección `attributes`. Dependiendo del navegador, esta colección se podrá cubrir de diferentes maneras y podrán existir muchos pares en la colección, tantos como atributos tenga el elemento. Para buscar el par correcto emplearemos la propiedad `nodeName`, que nos devolverá el nombre del atributo y para acceder a su valor usaremos `nodeValue`.

En el ejemplo:

```
<input type="text" id="apellidos" name="apellidos" />
```



[Ciker-Free-Vector-Images](#) ([Pixabay License](#))

Para imprimir todos los atributos del elemento "apellidos", podríamos hacer un bucle que recorriera todos esos atributos imprimiendo su valor:

```
document.write("<br/>El elemento <b>apellidos</b> contiene los pares atributo -> valor: <br/>");  
for( let x = 0; x < document.getElementById("apellidos").attributes.length; x++) {  
    var atributo = document.getElementById("apellidos").attributes[x];  
    document.write(atributo.nodeName+ " -> "+atributo.nodeValue+"<br/>");  
}
```

También podemos **modificar los valores de un atributo** de un nodo manualmente, por ejemplo:

```
document.getElementById("apellidos").attributes[0].nodeValue="password";  
// En este caso hemos modificado el type del campo apellidos y lo hemos puesto de tipo "password".
```

O también:

```
document.getElementById("apellidos").attributes["type"].nodeValue="password";  
// hemos puesto el nombre del atributo como referencia en el array de atributos.
```

O también:

```
document.getElementById("apellidos").type="password";  
// hemos puesto el atributo como una propiedad del objeto apellidos y lo hemos modificado.
```

El método `setAttribute()` nos permitirá **crear o modificar atributos** de un elemento. Por ejemplo, para ponerle de nuevo al campo "apellidos" `type='text'` y un `value='Cid Blanco'`, haríamos:

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

Si lo que quieres realmente es chequear el valor del atributo y no modificarlo, se puede utilizar `getAttribute()`:

```
let valor = document.getElementById("apellidos").getAttribute('type');  
// o también  
let valor= document.getElementById("apellidos").type;
```

Y si lo que quieres es eliminar un atributo, lo podemos hacer con `removeAttribute()`:

```
// <div id="contenedor" align="left" width="200px">  
document.getElementById("contenedor").removeAttribute("align");  
// Obtendremos como resultado: <div id="contenedor" width="200px">
```

1.5.- 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:

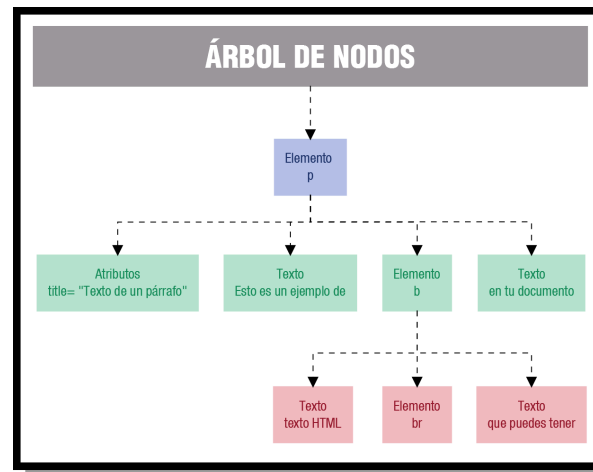
```
<p title="Texto de un párrafo">Esto es un ejemplo de <b>texto HTML<br /> que puedes tener</b> en tu documento.
```



[Ciker-Free-Vector-Images](#) (Pixabay License)

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.

Aquí puedes ver una imagen del árbol para ese elemento en cuestión:



Y el código de JavaScript para mostrar una alerta, con el contenido "texto HTML", sería:

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

`childNodes[1]` : selecciona el segundo hijo de `<p>` que sería el elemento `` (el primer hijo es un nodo de tipo Texto "Esto es un...").

`childNodes[0]` : selecciona el primer hijo del elemento `` que es el nodo de texto "texto HTML"

En lugar de `childNodes[0]` también podríamos haber utilizado `firstChild`, el cuál nos devuelve el primer hijo de un nodo.

Por ejemplo:

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

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.

1.6.- Creación y borrado de nodos.

La creación y borrado de nodos fue uno de los objetivos para los que se creó el DOM. Podremos crear elementos y luego insertarlos en el DOM, y la actualización quedará reflejada automáticamente por el navegador. También podremos mover nodos ya existentes (como el párrafo del punto 1.4) simplemente insertándolo en cualquier otro lugar del árbol del DOM.

Usaremos los métodos `createElement()`, `createTextNode()` y `appendChild()`, que nos permitirán crear un elemento, crear un nodo de texto y añadir un nuevo nodo hijo.

Ejemplo de creación de un nuevo párrafo, suponiendo que partimos del siguiente código:

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



[Peggy_Marco](#) (Pixabay License)

Para crear el nuevo párrafo haremos:

```
let nuevoParrafo = document.createElement('p');  
let nuevoTexto = document.createTextNode('Contenido añadido al párrafo.');
```

nuevoParrafo.appendChild(nuevoTexto);
document.getElementById('parrafo').appendChild(nuevoParrafo);

Y obtendremos como resultado:

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


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

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
let elementoP = document.createElement('p');
let elementoB = document.createElement('b');
let elementoBR = document.createElement('br');

//Le asignamos un nuevo atributo title al elementoP que hemos creado.
elementoP.setAttribute('title', 'Parrafo creado desde JavaScript');

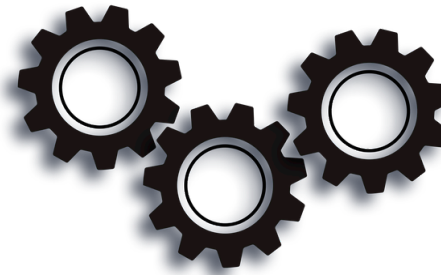
//Preparamos los nodos de texto
let texto1 = document.createTextNode('Con JavaScript se ');
let texto2 = document.createTextNode('pueden realizar ');
let texto3 = document.createTextNode('un monton');
let texto4 = document.createTextNode(' de cosas sobre el documento.');
```

```
//Añadimos al elemento B los nodos de texto2, elemento BR y texto3.
elementoB.appendChild(texto2);
elementoB.appendChild(elementoBR);
elementoB.appendChild(texto3);

//Añadimos al elemento P los nodos de texto1, elemento B y texto 4.
elementoP.appendChild(texto1);
elementoP.appendChild(elementoB);
elementoP.appendChild(texto4);

//insertamos el nuevo párrafo como un nuevo hijo de nuestro parrafo
document.getElementById('parrafo').appendChild(elementoP);
```

1.7.- Propiedades y métodos de los objetos nodo (DOM nivel 2 W3C).



[TheDigitalArtist](#) (Pixabay License)

Propiedades del objeto nodo según el DOM nivel 2 de W3C

Propiedad	Valor	Descripción	IE6Win+	IE5Mac+	Mozilla	Safari
nodeName	String	Varía según el tipo de nodo.	Sí.	Sí.	Sí.	Sí.
nodeValue	String	Varía según el tipo de nodo.	Sí.	Sí.	Sí.	Sí.
nodeType	Integer	Constante que representa cada tipo.	Sí.	Sí.	Sí.	Sí.
parentNode	Object	Referencia al siguiente contenedor más externo.	Sí.	Sí.	Sí.	Sí.
childNodes	Array	Todos los nodos hijos en orden.	Sí.	Sí.	Sí.	Sí.
firstChild	Object	Referencia al primer nodo hijo.	Sí.	Sí.	Sí.	Sí.
lastChild	Object	Referencia al último nodo hijo.	Sí.	Sí.	Sí.	Sí.

Propiedad	Valor	Descripción	IE6Win+	IE5Mac+	Mozilla	Safari
previousSibling	Object	Referencia al hermano anterior según su orden en el código fuente.	Sí.	Sí.	Sí.	Sí.
nextSibling	Object	Referencia al hermano siguiente según su orden en el código fuente.	Sí.	Sí.	Sí.	Sí.
attributes	NodeMap	Array de atributos de los nodos.	Sí.	Algunos.	Sí.	Sí.
ownerDocument	Object	Contiene el objeto <code>document</code> .	Sí.	Sí.	Sí.	Sí.
namespaceURI	String	URI a la definición de namespace.	Sí.	No.	Sí.	Sí.
Prefix	String	Prefijo del namespace.	Sí.	No.	Sí.	Sí.
localName	String	Aplicable a los nodos afectados en el namespace.	Sí.	No.	Sí.	Sí.

Métodos del objeto nodo según el DOM nivel 2 de W3C

Método	Descripción	IE5++	Mozilla	Safari
appendChild(newChild)	Añade un hijo al final del nodo actual.	Sí.	Sí.	Sí.
cloneNode(deep)	Realiza una copia del nodo actual (opcionalmente con todos sus hijos).	Sí.	Sí.	Sí.
hasChildNodes()	Determina si el nodo actual tiene o no hijos (valor boolean).	Sí.	Sí.	Sí.
insertBefore(new, ref)	Inserta un nuevo hijo antes de otro hijo.	Sí.	Sí.	Sí.
removeChild(old)	Borra un hijo.	Sí.	Sí.	Sí.
replaceChild(new, old)	Reemplaza un hijo viejo con el nuevo viejo.	Sí.	Sí.	Sí.
isSupported(feature, version)	Determina cuando el nodo soporta una característica especial.	No.	Sí.	Sí.

2.- Gestión de eventos.

Caso práctico

Antonio ya ha visto en anteriores unidades el modelo de registro de Eventos, y los modelos de disparo de eventos. Ha llegado el momento de profundizar un poco más en los eventos, y ver toda la información y posibilidades que nos dan: cómo detener un evento, o su propagación, incompatibilidades entre navegadores en la gestión de eventos, etc.

Juan le comenta a **Antonio** que se centre, sobre todo, en el estudio de los eventos del ratón y del teclado, ya que son los más utilizados, y los que pueden suponer diferencias de gestión según el tipo de navegador web utilizado.



Como ya te comentábamos en la unidad anterior 5, 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 de alguna forma detectar las acciones del usuario, para saber cuando reaccionar. También necesita saber las funciones a ejecutar cuando se producen 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 disparará automáticamente cuando un documento ha sido cargado en el navegador.

También comentábamos que hay diferencias, en lo que es la gestión de eventos, por unos navegadores u otros. Esas diferencias provocan que los programadores de páginas web, tengan que tener mucha precaución con los métodos y propiedades que usan, dependiendo del navegador que ejecutará la página de JavaScript.

Un ejemplo de solución cross-browser para asignar un evento, independientemente del navegador utilizado podría ser:

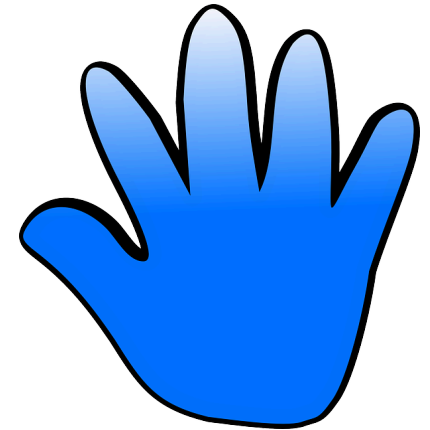
```
const crearEvento = (elemento, tipoEvento, funcion) => {  
  if (elemento.addEventListener) {  
    elemento.addEventListener(tipoEvento, funcion, false);  
  }  
  else if (elemento.attachEvent) {  
    elemento.attachEvent("on" + tipoEvento, funcion);  
  }  
  else {  
    elemento["on" + tipoEvento] = funcion;  
  }  
}  
let miparrafo = document.getElementById("parrafo");  
crearEvento(miparrafo, 'click', function(){alert("hola")});
```

La función `crearEvento`, lo primero que intenta hacer es asignar el evento con el método `addEventListener()` del W3C, y que soportan los navegadores más modernos; en el caso de que esa opción falle, intenta asignar el evento usando el método de Internet Explorer; y, si por último esta opción tampoco va, intenta asignar el evento en línea, como un atributo más del objeto.

2.1.- Modelos de eventos.

Vamos a hacer un pequeño repaso de los modelos de eventos, los cuáles fueron detallados en la unidad 5, apartados 3.1 al 3.4 y la fase de disparo de eventos apartado 3.5.

Veíamos que tenemos **4 modelos de registro de eventos**:



[Clicer-Free-Vector-Images](#) ([Pixabay License](#))

- ✓ 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: `Pulsa aqui`
- ✓ 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, pero si utilizado ampliamente por Netscape y Microsoft.
 - 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 burbujan, 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 **3 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()`.

2.2.- 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 el W3C, 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).



[janjf93 \(Pixabay License\)](#)

Algunas categorías y tipos de eventos en el modelo W3C

Categoría	Tipo de Evento	Descripción	Burbujea	Se puede cancelar
Ratón	click	Al hacer click sobre un elemento. Un click se define como <code>mousedown</code> y <code>mouseup</code> sobre la misma localización en pantalla.	Sí.	Sí.
	dblclick	Al hacer doble click sobre un elemento.	Sí.	Sí.
	mousedown	Al mantener presionado el botón del ratón sobre un elemento.	Sí.	Sí.

Categoría	Tipo de Evento	Descripción	Burbujea	Se puede cancelar
	mousedownmouseup	Al soltar el botón del ratón que estaba sobre un elemento.	Sí.	Sí.
	mouseover	Al pasar el ratón justo sobre un elemento.	Sí.	No.
	mousemove	Cuando el ratón se mueve mientras está sobre un elemento.	Sí.	Sí.
	mouseout	Cuando el ratón sale fuera de un elemento.	Sí.	Sí.
Teclado	keydown	Este evento se dispara justo antes del evento keypress al presionar una tecla.	Sí.	Sí.
	keypress	Este evento se dispara después de keydown al presionar una tecla.	Sí.	Sí.
	keyup	Al soltar una tecla.	Sí.	Sí.
Frame HTML	load	Se dispara cuando se ha terminado de cargar todo el contenido de un documento, incluyendo ventanas, frames, objetos e imágenes.	No.	No.
	unload	Al salir de un documento y modificar el contenido de una ventana.	No.	No.
	abort	Cuando se detiene la carga de un objeto/imagen antes de que esté completamente cargado.	Sí.	No.
	error	Cuando se detiene la carga de un objeto/imagen antes de que esté completamente cargado.	Sí.	No.
	resize	Cuando se redimensiona un documento.	Sí.	No.
	scroll	Cuando nos desplazamos por el documento con scroll.	Sí.	No.

2.3.- 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 a mayores alguna tecla especial pulsada como Alt, Control, etc.

Para gestionar toda esa información disponemos del objeto `Event`, el cuál 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);
const gestionar = (miEvento) => {
    alert (miEvento.type); // Mostrará una alerta con el tipo de evento que en este caso es 'click'.
}
```

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.



[OpenClipart-Vectors](#) ([Pixabay License](#))

2.3.1.- Propiedades y métodos del objeto Event.



[OpenClipart-Vectors](#) ([Pixabay License](#))

Veamos una lista de propiedades del objeto `Event`:

Propiedades del objeto Event

Propiedades	Descripción
<code>altKey</code> , <code>ctrlKey</code> , <code>metaKey</code> , <code>shiftKey</code>	Valor booleano que indica si están presionadas alguna de las teclas <code>Alt</code> , <code>Ctrl</code> , <code>Meta</code> o <code>Shift</code> en el momento del evento.
<code>bubbles</code>	Valor booleano que indica si el evento burbujea o no.
<code>button</code>	Valor <code>integer</code> que indica que botón del ratón ha sido presionado o soltado, 0=izquierdo, 2=derecho, 1=medio.
<code>cancelable</code>	Valor booleano que indica si el evento se puede cancelar.

Propiedades	Descripción
charCode	Indica el carácter <u>Unicode</u> 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. Por ejemplo si tenemos un evento de click en un <code>divA</code> que contiene un hijo <code>divB</code> . Si hacemos click en <code>divB</code> , <code>currentTarget</code> referenciará a <code>divA</code> (el elemento dónde se asignó el evento) mientras que <code>target</code> devolverá <code>divB</code> , el elemento dónde ocurrió el evento.
eventPhase	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" indica el nodo que ha abandonado el ratón. En un evento de "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 disparó el evento.
target	El elemento dónde se originó el evento, que puede diferir del elemento que tenga asignado el evento. Véase <code>currentTarget</code> .
timestamp	Devuelve la hora (en milisegundos desde <code>epoch</code>) a la que se creó el evento. Por ejemplo cuando se presionó una tecla. No todos los eventos devuelven <code>timestamp</code> .
type	Una cadena de texto que indica el tipo de evento "click", "mouseout", "mouseover", etc.
which	Indica el Unicode de la tecla presionada. Idéntico a <code>charCode</code> , excepto que esta propiedad también funciona en Netscape 4.

Veamos una lista de métodos del objeto `Event`:

Métodos del objeto Event

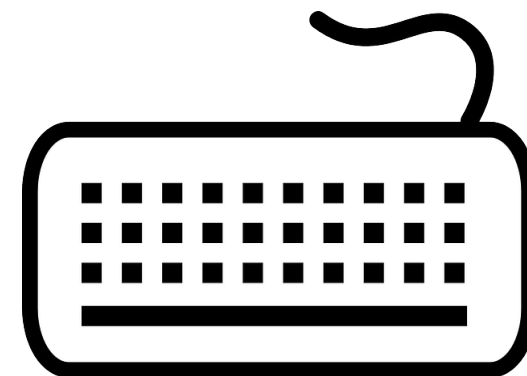
Métodos	Descripción
<code>preventDefault()</code>	Cancela cualquier acción asociada por defecto a un evento.
<code>stopPropagation()</code>	Evita que un evento burbujee. Por ejemplo si tenemos un <code>divA</code> que contiene un <code>divB</code> hijo. Cuando asignamos un evento de click a <code>divA</code> , si hacemos click en <code>divB</code> , por defecto se dispararía también el evento en <code>divA</code> en la fase de burbujeo. Para evitar esto se puede llamar a <code>stopPropagation()</code> en <code>divB</code> . Para ello creamos un evento de click en <code>divB</code> y le hacemos <code>stopPropagation()</code> .

2.3.2.- Eventos del teclado en JavaScript.

Uno de los eventos más complicados de gestionar en JavaScript son los eventos de teclado, debido a que suele haber bastantes incompatibilidades entre navegadores, teclados, idiomas, etc.

Para el teclado disponemos de 3 tipos de eventos: `keydown`, `keypress` y `keyup`. Y además disponemos de dos tipos de teclas: las especiales (`Shift`, `Alt`, `AltGr`, `Enter`, etc.) y las teclas **normales**, que contienen letras, números, y símbolos.

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`.



[Ciker-Free-Vector-Images](#) (Pixabay License)

- ✓ `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 `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:

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

```
<input type="button" id="enviar" value="Enviar" />
</form>
.....
<script type="text/javascript">
  const cambiar = (evt) => {
    if (evt.keyCode == 13) // Código de la tecla Enter
      if (this.nextSibling.nextSibling.type == "text")
        this.nextSibling.nextSibling.focus();
  }
  let inputs = document.getElementsByTagName("input");
  for (let i = 0; i < inputs.length; i++) {
    inputs[i].addEventListener("keypress", cambiar, false);
  }
</script>
```

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

- ✓ **this.nextSibling** - hace referencia al siguiente hermano al actual (la siguiente etiqueta `label` del siguiente campo).
- ✓ **this.nextSibling.nextSibling** - hermano siguiente, al hermano del elemento actual. (será otro elemento `input`. Si pusiéramos un salto de línea entre campos `input` entonces ya ese `this.nextSibling.nextSibling` ya no sería un campo `input` y sería un nodo de texto con el carácter `\n` del salto de línea que hemos puesto como separador de los campos `input`).

2.3.3.- Eventos del ratón en JavaScript.

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`. Ésto 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 ésto 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 el `divA` y nos vamos moviendo hacia el contenedor interno, veremos que ese evento sigue disparándose cuando estemos sobre `divB` o entremos en `divC`. Ésta 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`.



[Ciker-Free-Vector-Images \(Pixabay License\)](#)

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 type="text/javascript">
    const mostrarCoordenadas = (elEvento) => {
        document.getElementById("coordenadas").value=elEvento.clientX+" : "+elEvento.clientY;
    }
    document.addEventListener('mousemove',mostrarCoordenadas,false);
</script>
```

Para saber más

En el siguiente enlace podrás ampliar conocimientos sobre las propiedades de este evento.

[Tabla de propiedades de posicionamiento del ratón.](#)