Gestión de eventos en JavaScript

1. Modelos de eventos

Debido a la incompatibilidad existente entre los navegadores, existen diferentes modelos para manejar los eventos.

a) Modelo básico de eventos

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

b) Modelo de eventos estándar

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

c) Modelo de eventos de Internet Explorer

Internet Explorer utiliza su propio modelo de eventos, que es similar pero incompatible con el modelo estándar. Se utilizó por primera vez en Internet Explorer 4 y Microsoft decidió seguir utilizándolo en el resto de versiones, a pesar de que la empresa había participado en la creación del estándar de DOM que define el modelo de eventos estándar.

Tipos de eventos

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

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

Algunos eventos de la tabla anterior (`onclick`, `onkeydown`, `onkeypress`, `onreset`, `onsubmit`) permiten evitar la "acción por defecto" de ese evento.

Las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar por ejemplo sobre un botón de tipo <input type="submit"> se desencadenan los eventos `onmousedown`, `onclick`, `onmouseup` y `onsubmit` de forma consecutiva.

Manejadores de eventos

Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede *responder* ante cualquier evento que se produzca durante su ejecución.

Existen varias formas diferentes de indicar los manejadores:

- Manejadores como atributos de los elementos HTML.
- Manejadores como funciones JavaScript externas.
- Manejadores *"semánticos"*.
- Función addEventListener()

Manejadores como atributos de elementos HTML

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.

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

También podemos hacer uso de la variable this.

```
<div id="contenidos" style="width:150px; height:60px; border:thin solid silver"
onmouseover = "this.style.borderColor='black';" onmouseout =
"this.style.borderColor= 'silver';">
  Sección de contenidos...
</div>
```

Manejadores como funciones JavaScript externas

Esta técnica consiste en extraer todas las instrucciones de JavaScript y agruparlas en una función externa. Una vez definida la función, en el atributo del elemento HTML se incluye el nombre de la función, para indicar que es la función que se ejecuta cuando se produce el evento.

La llamada a la función se realiza de la forma habitual, indicando su nombre seguido de los paréntesis y de forma opcional, incluyendo todos los argumentos y parámetros que se necesiten.

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

```
function resalta(elemento) {
  switch(elemento.style.borderColor) {
    case 'silver':
    case 'silver silver silver silver':
    case '#c0c0c0':
      elemento.style.borderColor = 'black';
      break;
    case 'black':
    case 'black black black black':
    case '#000000':
      elemento.style.borderColor = 'silver';
      break;
  }
}
```

```
<div style="width:150px; height:60px; border:thin solid silver"
onmouseover="resalta(this)" onmouseout="resalta(this)">
  Sección de contenidos...
</div>
```

Manejadores "semánticos"

Los métodos que se han visto para añadir manejadores de eventos (como atributos HTML y como funciones externas) tienen un grave inconveniente: *"ensucian"* el código HTML de la página.

Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica es una evolución del método de las funciones externas, ya que se basa en utilizar las propiedades DOM de los elementos HTML para asignar todas las funciones externas que actúan de manejadores de eventos. Así, el siguiente ejemplo:

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

Se puede transformar en:

```
// Función externa
function muestraMensaje() {
  alert('Gracias por pinchar');
}

// Asignar la función externa al elemento
document.getElementById("pinchable").onclick = muestraMensaje;

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

La técnica de los manejadores semánticos consiste en:

1. Asignar un identificador único al elemento HTML mediante el atributo `id`.
2. Crear una función de JavaScript encargada de manejar el evento.
3. Asignar la función externa al evento correspondiente en el elemento deseado (sin paréntesis).

La gran ventaja de este método es que el código HTML resultante es muy *"limpio"*, ya que no se mezcla con el código JavaScript. Además, dentro de las funciones externas asignadas sí que se puede utilizar la variable `this` para referirse al elemento que provoca el evento.

El único inconveniente de este método es que la página se debe cargar completamente antes de que se puedan utilizar las funciones DOM que asignan los manejadores a los elementos HTML. Una de las formas más sencillas de asegurar que cierto código se va a ejecutar después de que la página se cargue por completo es utilizar el evento `onload`:

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

Función addEventListener()

Otra forma de gestionar eventos manteniendo limpio código HTML es escuchando.

En el nivel 2 del DOM se define el método `addEventListener` que nos permite indicar al agente de usuario que permanezca atento a la interacción de un usuario sobre un elemento en concreto.

La sintaxis de `addEventListener` es muy sencilla:

```
elemento_que_se_escucha.addEventListener('evento',función_a_lanzar,booleano);
```

Veámoslo con más detalle:

- *elemento_que_se_escucha* es cualquier elemento presente en un documento, al que accedemos por el medio que elijamos, bien por su id, por su etiqueta o las propiedades de otro nodo.
- *evento* es el suceso ocurrido sobre el elemento, con o sin interacción del usuario.
- función_a_lanzar es cualquier función definida que queramos que se ejecute cuando ocurra el evento.
- booleano es un valor que define el orden del flujo de eventos, algo que veremos un poco más abajo.

Pongamos un ejemplo. Si tuviéramos un formulario y quisiéramos que éste se validase antes de ser enviado al servidor, la forma errónea de hacerlo sería ésta:

```
<input type="submit" onclick="validar()">Enviar formulario</button>
```

Mediante una escucha, en el marcado tendríamos:

```
<input type="submit" id="enviar">Enviar formulario</button>
```

y en el script:

```
document.getElementById('enviar').addEventListener('click',validar,false);
```

¿Y qué ocurre si necesito escuchar un evento sólo una vez? Para ese caso existe un método complementario de `addEventListener`, que es `removeEventListener`:

```
elemento_que_escuchaba.removeEventListener('evento',función_anular,booleano);
```

Esta función no está soportada por IE8 y versiones anteriores. Este navegador trabaja con los métodos:

```
element.attachEvent('manejador_evento', function);
element.detachEvent('manejador_evento', function);
```

Luego una forma más genérica de trabajar con los eventos sería:

```
var x = document.getElementById("myBtn");
if (x.addEventListener) {  // For all major browsers, except IE 8 and earlier
    x.addEventListener("click", myFunction);
} else if (x.attachEvent) {    // For IE 8 and earlier versions
    x.attachEvent("onclick", myFunction);
}
```

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.

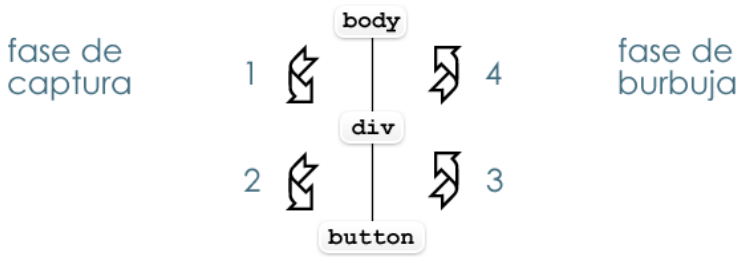
Supongamos que tenemos este marcado:

```
<body>
    <div>
        <button>HAZME CLIC</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. Según éste, cuando se hace clic sobre un elemento, el evento se propaga en dos fases:

- *Captura* :el evento comienza en el nivel superior del documento y recorre los elementos de padres a hijos
- *Burbuja*: el orden inverso, ascendiendo de hijos a padres



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`.

2. Obtener información de los eventos

Normalmente, los manejadores de eventos requieren información adicional para procesar sus tareas. Si una función por ejemplo se encarga de procesar el evento asociado al teclado, normalmente, es muy importante conocer la tecla que se ha pulsado, por ejemplo para diferenciar las teclas normales de las teclas especiales (`ENTER`, tabulador, `Alt`, `Ctrl.`, etc.).

JavaScript permite obtener información sobre el ratón y el teclado mediante un objeto especial llamado `event`. Desafortunadamente, los diferentes navegadores presentan diferencias muy notables en el tratamiento de la información sobre los eventos.

La principal diferencia reside en la forma en la que se obtiene el objeto `event`. Internet Explorer considera que este objeto forma parte del objeto `window` y el resto de navegadores lo consideran como el único argumento que tienen las funciones manejadoras de eventos.

Aunque es un comportamiento que resulta muy extraño al principio, todos los navegadores modernos excepto Internet Explorer crean *mágicamente* y de forma automática un argumento que se pasa a la función manejadora, por lo que no es necesario incluirlo en la llamada a la función manejadora. De esta forma, para utilizar este *"argumento mágico"*, sólo es necesario asignarle un nombre, ya que los navegadores lo crean automáticamente.

En resumen, en los navegadores tipo Internet Explorer, el objeto `event` se obtiene directamente mediante:

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

Por otra parte, en el resto de navegadores, el objeto `event` se obtiene *mágicamente* a partir del argumento que el navegador crea automáticamente:

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

Si se quiere programar una aplicación que funcione correctamente en todos los navegadores, es necesario obtener el objeto `event` de forma correcta según cada navegador. El siguiente código muestra la forma correcta de obtener el objeto `event` en cualquier navegador:

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

Una vez obtenido el objeto `event`, ya se puede acceder a toda la información relacionada con el evento, que depende del tipo de evento producido.

Información sobre el evento

La propiedad `type` indica el tipo de evento producido, lo que es útil cuando una misma función se utiliza para manejar varios eventos:

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

La propiedad `type` devuelve el tipo de evento producido, que es igual al nombre del evento pero sin el prefijo `on`.

Mediante esta propiedad, se puede rehacer de forma más sencilla el ejemplo anterior en el que se resaltaba una sección de contenidos al pasar el ratón por encima:

```
function resalta(elEvento) {
  var evento = elEvento || window.event;
  switch(evento.type) {
    case 'mouseover':
      this.style.borderColor = 'black';
      break;
    case 'mouseout':
      this.style.borderColor = 'silver';
      break;
  }
}

window.onload = function() {
  document.getElementById("seccion").onmouseover = resalta;
  document.getElementById("seccion").onmouseout = resalta;
}

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

PROPIEDADES DEFINIDAS POR DOM- EVENT

Propiedad/Método	Devuelve	Descripción
`altKey`	Boolean	Devuelve `true` si se ha pulsado la tecla `ALT` y `false` en otro caso
`bubbles`	Boolean	Indica si el evento pertenece al flujo de eventos de bubbling
`button`	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: `0` – Ningún botón pulsado `1` – Se ha pulsado el botón `izquierdo2- Se ha pulsado el botón derecho3- Se pulsan a la vez el botón izquierdo y el derecho4- Se ha pulsado el botón central5- Se pulsan a la vez el botón izquierdo y el central6- Se pulsan a la vez el botón derecho y el central7` – Se pulsan a la vez los 3 botones
`cancelable`	Boolean	Indica si el evento se puede cancelar
`cancelBubble`	Boolean	Indica si se ha detenido el flujo de eventos de tipo *bubbling*
`charCode`	Número entero	El código unicode del carácter correspondiente a la tecla pulsada
`clientX`	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
`clientY`	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
`ctrlKey`	Boolean	Devuelve `true` si se ha pulsado la tecla `CTRL` y `false` en otro caso
`currentTarget`	Element	El elemento que es el objetivo del evento
`detail`	Número entero	El número de veces que se han pulsado los botones del ratón
`eventPhase`	Número entero	La fase a la que pertenece el evento: `0` – Fase capturing `1` – En el elemento destino `2` – Fase bubbling
`isChar`	Boolean	Indica si la tecla pulsada corresponde a un carácter
`keyCode`	Número entero	Indica el código numérico de la tecla pulsada
`metaKey`	Número entero	Devuelve `true` si se ha pulsado la tecla `META` y `false` en otro caso
`pageX`	Número entero	Coordenada X de la posición del ratón respecto de la página
`pageY`	Número entero	Coordenada Y de la posición del ratón respecto de la página
`preventDefault()`	Función	Se emplea para cancelar la acción predefinida del evento
`relatedTarget`	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)

`screenX`	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
`screenY`	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
`shiftKey`	Boolean	Devuelve `true` si se ha pulsado la tecla `SHIFT` y `false` en otro caso
`stopPropagation()`	Función	Se emplea para detener el flujo de eventos de tipo bubbling
`target`	Element	El elemento que origina el evento
`timeStamp`	Número	La fecha y hora en la que se ha producido el evento
`type`	Cadena de texto	El nombre del evento

Al contrario de lo que sucede con Internet Explorer, la mayoría de propiedades del objeto `event` de DOM son de sólo lectura. En concreto, solamente las siguientes propiedades son de lectura y escritura: `altKey`, `button` y `keyCode`.

La tecla `META` es una tecla especial que se encuentra en algunos teclados de ordenadores muy antiguos. Actualmente, en los ordenadores tipo PC se asimila a la tecla `Alt` o a la tecla de Windows, mientras que en los ordenadores tipo Mac se asimila a la tecla `Command`.