addEventListener

Introducción

Todo manual moderno de JavaScript que merezca la pena comienza diciendo que escribir líneas como estas:

```
<a href="javascript:función_que_sea()">Un vínculo</a>
<a href="función_que_sea()">Un vínculo</a>
<a href="#" onclick="función_que_sea()">Un vínculo</a>
```

es una pésima idea. ¿Por qué? Bueno, pues porque si el agente de usuario de nuestra visita no soporta JavaScript o no lo tiene activado, la funcionalidad añadida por medio del *script* simplemente no funciona.

Un paso más allá es crear un marcado como el siguiente:

```
<a href="página_con_funcionalidad_alternativa_no_dependiente_de_JS.htm"
onclick="función_que_sea();return false;">Un vínculo</a>
```

Sin embargo, si queremos acercarnos lo más posible al ideal de separar las capas de contenido, presentación y comportaiento de un documento, deberíamos mantener el marcado limpio de atributos como `onclick`, que pertenece propiamente al comportamiento.

La pregunta es: ¿cómo hacer esto de manera que las mejoras para la experiencia del usuario que permite JavaScript sigan siendo efectivas? Y la respuesta es: escuchando.

Escuchar eventos

En el nivel 2 del DOM («Document Object Model», Modelo de Objeto de Documento) se define el método `addEventListener` (inglés) [http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/events.html#Events-EventTarget-addEventListener] que nos permite precisamente lo que hemos dicho arriba: indicar al agente de usuario que permanezca atento a la interacción de un usuario sobre un elemento en concreto, sin necesidad de tocar un sólo caracter de nuestro marcado.

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` [curso_4_3_a.php#gEBI], por su etiqueta [curso_4_3_a.php#gEBTN] o las propiedades de otro nodo [curso_4_3_b.php].
- *evento* es el suceso ocurrido sobre el elemento, con o sin interacción del usuario, como vimos en la sección de sintaxis de JavaScript relativa a los eventos [curso_4_2_g.php].
- *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:

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

Mediante una escucha, en el marcado tendríamos:

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

y en el *script*:

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

Mucho más limpio, y además nos aseguramos de que si JavaScript no está disponible, el formulario podrá ser enviado.

Aquí dejo un ejemplo funcional [js/ejemplo_addEventListener_01.php].

¿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_se_escuchaba.removeEventListener('evento',función_a_anular,booleano);
```

Veámoslo [js/ejemplo_addEventListener_02.php].

En los ejemplos he escuchado el evento `click`, pero la mecánica es la misma para cualquier otro, siempre que el elemento acepte el evento. Y los eventos que acepta cada elemento ya los vimos en la descripción de los mismos en XHTML («eXtensible HyperText Markup Language», Lenguaje de Marcado de HiperTexto eXtensible) 1.1 [curso_2_3.php].

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 («Document Object Model», Modelo de Objeto de Documento), es decir, hemos hecho clic además sobre el elemento `body` y sobre el elemento `div`[1]. Sí 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) [http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/events.html#Events-flow]. 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:

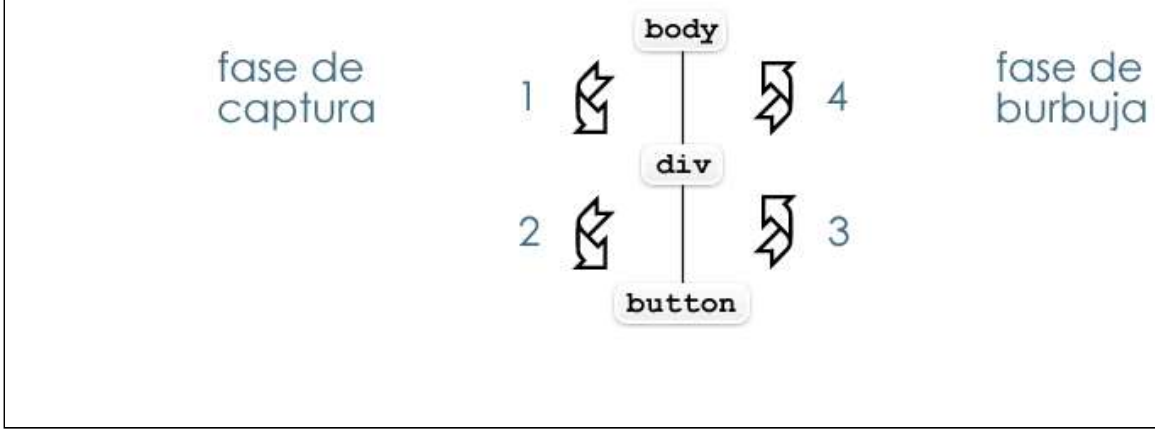

Ilustración del párrafo anterior

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`.

Un ejemplo con el valor `true` [js/prueba_burbuja_captura_01.php], y otro con el valor `false` [js/prueba_burbuja_captura_02.php].

En este punto ya podemos crear comportamientos sin ensuciar nuestro marcado con atributos relativos al comportamiento. Genial, ¿no? Bueno, no del todo; como siempre, hay un factor alterador a tener en cuenta...

¿Y qué pasa con Explorer?

En el momento en que escribo, ni Internet Explorer 7 ni ninguno de sus predecesores soportan `addEventListener`. Lo que sí soportan las versiones 6 y 7 —ahora mismo no estoy seguro de si lo hacen la 4 y la 5— son dos métodos propietarios similares: `attachEvent` y `detachEvent`:

```
elemento_que_se_escucha.attachEvent('manejador_de_evento',función_a_lanzar);

elemento_que_se_escuchaba.detachEvent('manejador_de_evento',función_a_anular);
```

Como se ve, su sintaxis es muy similar a las de `addEventListener` y `removeEventListener`, salvo por dos detalles:

- No se hace referencia al evento, sino al *manejador de evento*, que sin meternos en cuestiones técnicas es lo mismo pero con un perfijo `on`; así, al evento `click` le corresponde el manejador `onclick`, a `mouseover` `onmouseover`, y así sucesivamente.

- No hay un tercer parámetro: en Explorer el flujo se identifica con la burbuja, y no hay fase previa de captura.

Así, siempre que se quieran establecer escuchas, habrá que emplear unos métodos para navegadores con un soporte de los métodos de escucha estándar, y los otros para Explorer. Éste es el motivo por el que todos antes o después todos empezamos a utilizar una función para establecer escuchas:

```
function ev (x,y,z) {
    if (document.addEventListener){
        x.addEventListener(y,z,false);
    } else {
        x.attachEvent('on'+y,z);
```

```
        }
    }
```

---

donde `x` es el elemento sobre el que se quiere establecer la escucha, `y` el evento y `z` la función a ejecutar ;) (guiño).

Y con esto, terminamos esta pequeña introducción al DOM.

Notas

1. Estrictamente también hemos hecho clic sobre el objeto `window`, el objeto `document` y el elemento `html`, pero para esta explicación y el ejemplo subsiguiente no es necesario remontarnos a tanto.