

Gestión Avanzada de Eventos

5

En la primera parte de este manual aprendimos qué eran los eventos y como controlarlos a través de atributos HTML o, mejor, haciendo uso de manejadores de eventos semánticos. También se indicó una tercera forma aún más elegante para controlar los eventos pero no se desarrolló.

Es en este capítulo donde vamos a aprender a usar las *funciones asociadoras* de eventos, cuya ventaja más relevante es que, aparte de separar presentación y programación dado que no se incrustan en el código HTML haciendo así que nuestro código sea más limpio, claro y elegante, poder decidir en que orden se ejecutan los eventos de varios elementos.

También vamos a aprender a tratar el objeto `event`, el cual se crea cada vez que se ejecuta un evento y guarda mucha información acerca de él.

5.1 Flujo de eventos

Sabemos que cualquier elemento del árbol DOM puede tener asociado un manejador (sea del tipo que sea) para algún tipo de evento (o para varios).

También sabemos que la mayoría de elementos de nuestro árbol cuelgan de otros elementos (es decir, tenemos etiquetas dentro de otras etiquetas) y que la raíz del DOM es el `document`.

Pues bien, imaginemos el siguiente ejemplo:

```
<body>
  <div> <p> HAZ CLICK AQUI </p> </div>
</body>
```

Si hacemos 'click' sobre el párrafo estamos, a su vez, haciendo 'click' sobre el DIV (su padre) y sobre BODY (padre del DIV).

Si solo tenemos un manejador asignado al párrafo no hay problema, pero ¿qué pasa si tanto el párrafo, como el DIV, como BODY tienen asignados manejadores para el evento click? Al pulsar en el párrafo se están disparando tres eventos a la vez. ¿Quién se ejecuta primero?

```
<body onclick='alert('Soy BODY')'>
  <div onclick='alert('Soy DIV')'>
    <p onclick='alert('Soy P')'> HAZ CLICK AQUI </p>
  </div>
</body>
```

Esto es algo que se decide a través del **flujo de eventos**, que no es más que el orden en que se ejecutan los eventos asignados a cada elemento de la página.

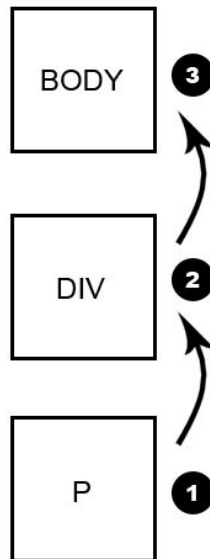
Este orden suele variar entre cada navegador (sobre todo entre los navegadores actuales y las versiones de Internet Explorer anteriores a la 9.0)

Existen dos modelos dentro del flujo de eventos: la fase de burbuja y la fase de captura.

5.1.1 Fase de Burbuja

En este modelo el orden de ejecución que se sigue es desde el elemento más específico al menos específico.

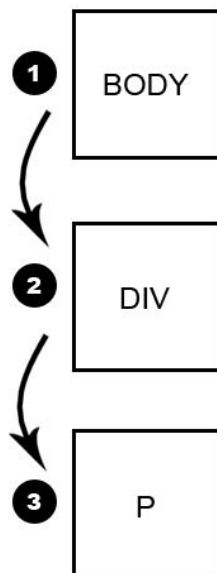
Para el ejemplo anterior, primero se ejecutaría el `alert` del párrafo, a continuación el del `div` y finalmente, el del `body`.



5.1.2 Fase de Captura

En este modelo el orden de ejecución es justo al contrario que el anterior. Es decir, va desde el elemento menos específico al más específico.

Para nuestro ejemplo, ahora primero se ejecutaría el `alert` del `body`, a continuación el del `div` y finalmente, el del párrafo.



5.1.3 Eventos DOM

Antiguamente, las diferentes versiones de Internet Explorer usaban sólo la fase de burbuja para la propagación de eventos mientras que el resto de navegadores usaban la fase de captura. Eso suponía un problema muy serio cuando se desarrollaba con Javascript.

Sin embargo, actualmente ambas fases están definidas en las especificaciones del estándar DOM, por lo que todos los navegadores actuales pueden ejecutar tanto la fase de burbuja como la de captura.

Si no indicamos nada, la propagación de eventos se hará usando la fase de burbuja. Si queremos 'mezclar' la propagación de eventos usando ambas fases, la mayoría de navegadores modernos ejecutarán primero la fase de captura y, a continuación, la fase de burbuja.

La pregunta es ¿Como podemos decidir nosotros que método de propagación de eventos queremos? Pues para ello necesitamos controlar los eventos con las *funciones asociadoras*.

5.2 Funciones asociadoras

Como su nombre indica, una función asociadora es una función propia del navegador que sirve para asignar un manejador a un evento, es decir, asocia un funciones a eventos de elementos de nuestra página. Además, estas funciones nos van a permitir decidir que modelo de propagación de eventos queremos para ese manejador.

La especificación DOM oficial define dos funciones para asociar y des-asociar manejadores de eventos a un elemento:

<code>addEventListener(evento, manejador, modelo)</code>	Asocia el evento del primer parámetro con su manejador indicado en el segundo parámetro. El tercer parámetro indica el modelo de propagación.
<code>removeEventListener(evento, manejador, modelo)</code>	Desasocia una asociación realizada previamente con <code>addEventListener</code> . Los argumentos deben coincidir con los del <code>addEventListener</code> que se quiere desasociar.

Los argumentos de ambas funciones son:

- **evento:** nombre del tipo de evento sin el prefijo ON (`onclick = click`, `onblur = blur`, `onload = load`, ...)
- **manejador:** aquí indicamos como vamos a controlar el evento. Podemos usar instrucciones Javascript, nombre de funciones o funciones anónimas.
- **modelo:** este parámetro es un boolean. El valor `false` indica propagación por burbuja y `true`, propagación por captura.

Importante recordar que estas funciones se ejecutan siempre sobre elementos de la forma:

```
//Referencia al elemento
var p = document.getElementById('parra');

//Asignacion de eventos
p.addEventListener(. . .);
```

Ejemplo:

```
<head>
  <title>Funciones Asociadoras</title>
  <script src="js/funciones.js"></script>
  <script>
    window.onload = function(){
      var p = document.getElementById('parra');
      var d = document.getElementById('salida');

      p.addEventListener("click",alert('soy el parrafo'),true);
      d.addEventListener("click",mostrar,true);
      document.addEventListener("click",function(){muestra('body');},true);
    }
  </script>
</head>
```

```
<body >
  <div id="salida">
    <p id="parra" > Pulsa aqui <p>
  </div>
</body>
```

Como muestra el ejemplo, hay varias formas de asignar un manejador (segundo parámetro) cuando usamos `addEventListener()`

1. Usando sentencias Javascript (separadas por comas).
2. Colocando el nombre de una función que no lleve argumentos. El nombre de la función no va entre comillas.
3. En el caso de necesitar una función con argumentos, usando funciones anónimas.

Si quisiéramos desasociar los eventos asociados en el ejemplo anterior, deberíamos hacer algo como esto:

```
. . .
p.removeEventListener("click",alert('soy el parrafo'),true);
d.removeEventListener("click",mostrar,true);
document.removeEventListener("click",function(){muestra('body');},true);
. . .
```

Importante: hay que tener en cuenta que para JS cada función anónima es diferente una de otra, independientemente de que tengan el mismo contenido. Esto es relevante porque este ejemplo de `removeEventListener` no funciona:

```
const parra = document.querySelector("p");
parra.addEventListener("click",
  function() {
    alert("Soy el P");
  }
);
```

```
document.body.addEventListener("click", function(){
    alert("Soy el body add");
    //Quito el evento del parrafo
    parra.removeEventListener("click",
        function(){
            alert("Soy el P");
        }
    );
});
```

Si queremos que el evento se elimine correctamente, no podemos usar funciones anónimas.

```
const parra = document.querySelector("p");
function eliminar(){
    alert("Soy el P");
}

parra.addEventListener("click", eliminar);

document.body.addEventListener("click", function(){
    alert("Soy el body add");
    //Quito el evento del parrafo
    parra.removeEventListener("click",eliminar);
});
```

Como se comentó en la primera parte de este manual, la mejor forma de asociar eventos a los elementos de una página es usando las funciones asociadoras que hemos visto en este apartado.

Las ventajas son obvias:

- Separan la presentación de la programación, por lo que nos queda un código más claro.
- Permiten asignar varios manejadores a un mismo evento. Esto es algo que no podíamos hacer usando manejadores como atributos HTML o manejadores semánticos.
- Podemos controlar qué modelo de propagación debe usarse en cada evento.

5.3 El Objeto event

Cuando se produce un evento, aparte de ejecutar el manejador correspondiente si estuviera definido, el navegador, de forma automática, crea un objeto de un tipo especial llamado `event` el cual guarda información relativa al evento que se ha ejecutado (Pej, tecla que se ha pulsado, posición del ratón, elemento que produce el evento, fase en la que está...) y permite hacer uso de métodos para trabajar con él.

Este objeto se crea automáticamente cuando se produce el evento y se destruye también de forma automática cuando finalizan todas las funciones asignadas al evento.

Ejemplo:

```
button.addEventListener("click", function(event) {  
    console.log(event);  
});
```

Cuando se pulsa el botón, se imprime por consola un objeto con muchas propiedades que hace referencia al evento click.

El DOM especifica que el objeto `event` es el único argumento que se puede pasar a las funciones anónimas encargadas de manejar los eventos (estén definidas usando manejadores semánticos o funciones asociadoras). El nombre del argumento puede ser el que queramos.

Ejemplo:

```
parrafo.onclick = function(e){  
    //Dentro de esta función anónima,  
    //el argumento 'e' será de tipo event  
}
```

```
parrafo.addEventListener ('click', function(ev){  
    //Dentro de esta función anónima,  
    //el argumento 'ev' será de tipo event  
}, false);
```

```
elemento.addEventListener("keyup", (event) => {  
    //dentro de esta función flecha  
    //el argumento 'event' será de tipo event  
});
```

5.3.1 Propiedades y métodos de event

Como cualquier objeto, `event` tiene una serie de propiedades y métodos que nos van a facilitar trabajar con él. Las más interesantes para nosotros son:

Propiedad/Método	Devuelve	Descripción
<code>target</code>	Elemento HTML	El elemento que origina el evento.
<code>currentTarget</code>	Elemento HTML	El elemento por el que se está propagando el evento.
<code>type</code>	String	El nombre del evento producido.

<code>preventDefault()</code>	Función	Elimina la acción predefinida del evento de un elemento (<i>pej: botones submit</i>)
<code>stopPropagation()</code>	Función	Detiene el flujo de propagación del evento desde ese momento.

Ejemplo:

```
var enla = document.getElementById('enlace');
enla.onclick = function(evento) {
    evento.preventDefault();
    alert(evento.type)
}
. . .
<a href="http://terra.es" id="enlace"> Pulsa </a>
```

En la función manejadora del ejemplo hemos impedido que el enlace haga su función normal si se pulsa (ir a la URL indicada) y en su lugar mostramos el tipo de evento que se dispara (click)

5.4 Acciones por defecto

Hay muchos elementos HTML que tienen asociado un evento por defecto, el cual “define” a dicho elemento, *pej: enlaces, botones de formulario, flechas de scroll...*

Si asociamos un manejador de eventos a estos elementos y hacemos saltar el evento, el código Javascript se ejecutará antes que el comportamiento por defecto (y luego se ejecutará este).

Podemos evitar que se produzca el evento por defecto usando el método `preventDefault` del objeto `event`:

```
let enlace = document.querySelector("a");
enlace.addEventListener("click", e => {
    console.log("No vas a ningún lado.");
    e.preventDefault();
});
```

En el código anterior, tomamos el primer enlace de la página, hacemos que muestre un mensaje por consola y evitamos que vaya a la url que posea (su comportamiento habitual).