UD04 - Gestión de eventos

1. Introducción

Nos permiten detectar acciones que realiza el usuario o cambios que suceden en la página y reaccionar en respuesta a ellas. Existen muchos eventos diferentes (podéis ver la lista en w3schools) aunque nosotros nos centraremos en los más comunes.

Javascript nos permite ejecutar código cuando se produce un evento (por ejemplo el evento *click* del ratón) asociando al mismo una función. Hay varias formas de hacerlo.

2. Cómo escuchar un evento

La primera manera "estándar" de asociar código a un evento era añadiendo un atributo con el nombre del evento a escuchar (con 'on' delante) en el elemento HTML. Por ejemplo, para ejecutar código al producirse el evento 'click' sobre un botón se escribía:

```
<input type="button" id="boton1" onclick="alert('Se ha pulsado');" />
```

Una mejora era llamar a una función que contenía el código:

```
<input type="button" id="boton1" onclick="clicked()" />
```

```
function clicked() {
  alert('Se ha pulsado');
}
```

Esto "ensuciaba" con código la página HTML por lo que se creó el modelo de registro de eventos tradicional que permitía asociar a un elemento HTML una propiedad con el nombre del evento a escuchar (con 'on' delante). En el caso anterior:

```
document.getElementById('boton1').onclick = funnction () {
  alert('Se ha pulsado');
}
...
```

NOTA: hay que tener cuidado porque si se ejecuta el código antes de que se haya creado el botón estaremos asociando la función al evento *click* de un elemento que aún no existe así que no hará nada. Para evitarlo siempre es conveniente poner el código que atiende a los eventos dentro de una función que se ejecute al producirse el evento *load* de la ventana. Este evento se produce cuando se han cargado todos los elementos HTML de la página y se ha creado el árbol DOM. Lo mismo habría que hacer con cualquier código que modifique el árbol DOM. El código correcto sería:

```
window.onload = function() {
  document.getElementById('boton1').onclick = function() {
    alert('Se ha pulsado');
  }
}
```

2.1 Event listeners

La forma recomendada de hacerlo es usando el modelo avanzado de registro de eventos del W3C. Se usa el método `addEventListener` que recibe como primer parámetro el nombre del evento a escuchar (sin 'on') y como segundo parámetro la función a ejecutar (OJO, sin paréntesis) cuando se produzca:

```
document.getElementById('boton1').addEventListener('click', pulsado);
...
function pulsado() {
  alert('Se ha pulsado');
})
```

Habitualmente se usan funciones anónimas ya que no necesitan ser llamadas desde fuera del escuchador:

```
document.getElementById('boton1').addEventListener('click', function() {
  alert('Se ha pulsado');
});
```

Si queremos pasarle algún parámetro a la función escuchadora (cosa bastante poco usual) debemos usar funciones anónimas como escuchadores de eventos.

NOTA: igual que antes debemos estar seguros de que se ha creado el árbol DOM antes de poner un escuchador por lo que se recomienda ponerlos siempre dentro de la función asociada al evento `window.onload` (o mejor `window.addEventListener('load', ...)` como en el ejemplo anterior).

Una ventaja de este método es que podemos poner varios escuchadores para el mismo evento y se ejecutarán todos ellos. Para eliminar un escuchador se usa el método `removeEventListener`.

```
document.getElementById('acepto').removeEventListener('click', aceptado);
```

NOTA: no se puede quitar un escuchador si hemos usado una función anónima, para quitarlo debemos usar como escuchador una función con nombre.

3. Tipos de eventos

Según qué o dónde se produce un evento estos se clasifican en:

3.1 Eventos de página

Se producen en el documento HTML, normalmente en el BODY:

- **load**: se produce cuando termina de cargarse la página (cuando ya está construido el árbol DOM). Es útil para hacer acciones que requieran que el DOM esté cargado como modificar la página o poner escuchadores de eventos
- **unload**: al destruirse el documento (ej. cerrar)
- **beforeUnload**: antes de destruirse (podríamos mostrar un mensaje de confirmación)
- **resize**: si cambia el tamaño del documento (porque se redimensiona la ventana)

3.2 Eventos de ratón

Los produce el usuario con el ratón:

- **click** / **dblclick**: cuando se hace click/doble click sobre un elemento
- **mousedown** / **mouseup**: al pulsar/soltar cualquier botón del ratón
- **mouseenter** / **mouseleave**: cuando el puntero del ratón entra/sale del elemento (tb. podemos usar mouseover/mouseout)
- **mousemove**: se produce continuamente mientras el puntero se mueva dentro del elemento

NOTA: si hacemos doble click sobre un elemento la secuencia de eventos que se produciría es: *mousedown -> mouseup -> click -> mousedown -> mouseup -> click -> dblclick*

> **EJERCICIO 01_UD4**: Pon un escuchador desde la consola al botón 1 de la página de ejemplo de DOM para que al hacer click se muestre el un alert con 'Click sobre botón 1'. Ponle otro para que al pasar el ratón sobre él se muestre 'Entrando en botón 1'.

3.3 Eventos de teclado

Los produce el usuario al usar el teclado:

- **keydown**: se produce al presionar una tecla y se repite continuamente si la tecla se mantiene pulsada
- **keyup**: cuando se deja de presionar la tecla
- **keypress**: acción de pulsar y soltar (sólo se produce en las teclas alfanuméricas)

NOTA: el orden de secuencia de los eventos es:*keyDown -> keyPress -> keyUp*

3.4 Eventos de toque

Se producen al usar una pantalla táctil:

- **touchstart**: se produce cuando se detecta un toque en la pantalla táctil
- **touchend**: cuando se deja de pulsar la pantalla táctil
- **touchmove**: cuando un dedo es desplazado a través de la pantalla
- **touchcancel**: cuando se interrumpe un evento táctil.

3.5 Eventos de formulario

Se producen en los formularios:

- **focus** / **blur**: al obtener/perder el foco el elemento
- **change**: al perder el foco un <input> o <textarea> si ha cambiado su contenido o al cambiar de valor un <select> o un <checkbox>
- **input**: al cambiar el valor de un <imput> o <textarea> (se produce cada vez que escribimos una letra es estos elementos)
- **select**: al cambiar el valor de un <select> o al seleccionar texto de un <imput> o <textarea>
- **submit** / **reset**: al enviar/recargar un formulario

4. Los objetos *this* y *event*

Al producirse un evento se generan automáticamente en su función manejadora 2 objetos:

- **this**: siempre hace referencia al elemento que contiene el código en donde se encuentra la variable *this*. En el caso de una función escuchadora será el elemento que tiene el escuchador que ha recibido el evento
- **event**: es un objeto y la función escuchadora lo recibe como parámetro. Tiene propiedades y métodos que nos dan información sobre el evento, como:
  - **.type**: qué evento se ha producido (click, submit, keyDown, ...)
  - **.target**: el elemento donde se produjo el evento (puede ser *this* o un descendiente de *this*, como en el ejemplo siguiente)
  - **.currentTarget**: el elemento que contiene el escuchador del evento lanzado (normalmente el mismo que *this*). Por ejemplo si tenemos un

    al que le ponemos un escuchador de 'click' que dentro tiene un elemento , si hacemos *click* sobre el **event.target** será el que es donde hemos hecho click (está dentro de

) pero tanto como *event.currentTarget* será

    (que es quien tiene el escuchador que se está ejecutando).
  - **.relatedTarget**: en un evento 'mouseover' **event.target** es el elemento donde ha entrado el puntero del ratón y **event.relatedTarget** el elemento del que ha salido. En un evento 'mouseout' sería al revés.
  - **cancelable**: si el evento puede cancelarse. En caso afirmativo se puede llamar a **event.preventDefault()** para cancelarlo

- **.preventDefault()**: si un evento tiene un escuchador asociado se ejecuta el código de dicho escuchador y después el navegador realiza la acción que correspondería por defecto al evento si no tuviera escuchador (por ejemplo un escuchador del evento *click* sobre un hiperenlace hará que se ejecute su código y después saltará a la página indicada en el *href* del hiperenlace). Este método cancela la acción por defecto del navegador para el evento. Por ejemplo si el evento era el *submit* de un formulario éste no se enviará o si era un *click* sobre un hiperenlace no se irá a la página indicada en él.
- **.stopPropagation**: un evento se produce sobre un elemento y todos su padres. Por ejemplo si hacemos click en un que está en un <p> que está en un que está en el BODY el evento se va propagando por todos estos elementos y saltarían los escuchadores asociados a todos ellos (si los hubiera). Si alguno llama a este método el evento no se propagará a los demás elementos padre.
- dependiento del tipo de evento tendrá más propiedades:
  - eventos de ratón:
    - **.button**: qué botón del ratón se ha pulsado (0: izq, 1: rueda; 2: dcho).
    - **.screenX** / **.screenY**: las corrdenadas del ratón respecto a la pantalla
    - **.clientX** / **.clientY**: las coordenadas del ratón respecto a la ventana cuando se produjo el evento
    - **.pageX** / **.pageY**: las coordenadas del ratón respecto al documento (si se ha hecho un scroll será el clientX/Y más el scroll)
    - **.offsetX** / **.offsetY**: las coordenadas del ratón respecto al elemento sobre el que se produce el evento
    - **.detail**: si se ha hecho click, doble click o triple click
  - eventos de teclado: son los más incompatibles entre diferentes navegadores. En el teclado hay teclas normales y especiales (Alt, Ctrl, Shift, Enter, Tab, flechas, Supr, ...). En la información del teclado hay que distinguir entre el código del carácter pulsado (e=101, E=69, €=8364) y el código de la tecla pulsada (para los 3 caracteres es el 69 ya que se pulsa la misma tecla). Las principales propiedades de *event* son:
    - **.key**: devuelve el nombre de la tecla pulsada
    - **.which**: devuelve el código de la tecla pulsada
    - **.keyCode** / **.charCode**: código de la tecla pulsada y del carácter pulsado (según navegadores)
    - **.shiftKey** / **.ctrlKey** / **.altKey** / **.metaKey**: si está o no pulsada la tecla SHIFT / CTRL / ALT / META. Esta propiedad también la tienen los eventos de ratón

    > NOTA: a la hora de saber qué tecla ha pulsado el usuario es conveniente tener en cuenta:
    >
    > - para saber qué carácter se ha pulsado lo mejor usar la propiedad *key* o *charCode* de *keyPress*, pero varía entre navegadores

- para saber la tecla especial pulsada mejor usar el *key* o el *keyCode* de *keyUp*
- captura sólo lo que sea necesario, se producen muchos eventos de teclado
- para obtener el carácter a partir del código: `String fromCharCode(codigo);`

Lo mejor para familiarizarse con los diferentes eventos es consultar los ejemplos de w3schools.

EJERCICIO 02_UD4: Pon desde la consola un escuchador al BODY de la página de ejemplo para que al mover el ratón en cualquier punto de la ventana del navegador, se muestre en algún sitio (añade un DIV o un P al HTML) la posición del puntero respecto del navegador y respecto de la página.

EJERCICIO 03_UD4: Pon desde la consola un escuchador al BODY de la página de ejemplo para que al pulsar cualquier tecla nos muestre en un alert el *key* y el *keyCode* de la tecla pulsada. Pruébalo con diferentes teclas

4.1 "Bindeo" del objeto *this*

En ocasiones no queremos que *this* sea el elemento sobre quien se produce el evento sino que queremos conservar el valor que tenía antes de entrar a la función escuchadora. Por ejemplo la función escuchadora es un método de una clase en *this* tenemos el objeto de la clase sobre el que estamos actuando pero al entrar en la función perdemos esa referencia.

El método *.bind()* nos permite pasarle a una función el valor que queremos darle a la variable *this* dentro de dicha función. Por defecto a una función escuchadora de eventos se le *bindea* le valor de **event.currentTarget**. Si queremos que tenga otro valor se lo indicamos con **.bind()**:

```
1  document.getElementById('acepto').removeEventListener('click',
   aceptado.bind(variable));
```

En este ejemplo el valor de *this* dentro de la función *aceptado* será *variable*. En el ejemplo que habíamos comentado de un escuchador dentro de una clase, para mantener el valor de *this* y que haga referencia al objeto sobre el que estamos actuando haríamos:

```
1  document.getElementById('acepto').removeEventListener('click',
   aceptado.bind(this));
```

por lo que el valor de *this* dentro de la función *aceptado* será el mismo que tenía fuera, es decir, el objeto.

Podemos *bindear*, es decir, pasarle a la función escuchadora más variables declarándolas como parámetros de *bind*. El primer parámetro será el valor de *this* y los demás serán parámetros que recibirá la función antes de recibir el parámetro *event* que será el último. Por ejemplo:

```
document.getElementById('acepto').removeEventListener('click',
aceptado.bind(var1, var2, var3));
...
function aceptado(param1, param2, event) {
  // Aquí dentro tendremos los valores
  // this = var1
  // param1 = var2
  // param2 = var3
  // event es el objeto con la información del evento producido
}
```

5. Propagación de eventos (bubbling)

Normalmente en una página web los elementos HTML se solapan unos con otros, por ejemplo, un está en un <p> que está en un que está en el <body>. Si ponemos un escuchador del evento *click* a todos ellos se ejecutarán todos ellos, pero ¿en qué orden?.

Pues el W3C estableció un modelo en el que primero se disparan los eventos de fuera hacia dentro (primero el <body>) y al llegar al más interno (el) se vuelven a disparar de nuevo pero de dentro hacia afuera. La primera fase se conoce como **fase de captura** y la segunda como **fase de burbujeo**. Cuando ponemos un escuchador con `addEventListener` el tercer parámetro indica en qué fase debe dispararse:

- **true**: en fase de captura
- **false** (valor por defecto): en fase de burbujeo

Sin embargo si al método `.addEventListener` le pasamos un tercer parámetro con el valor *true* el comportamiento será el contrario, lo que se conoce como *captura* y el primer escuchador que se ejecutará es el del <body> y el último el del (podéis probarlo añadiendo ese parámetro a los escuchadores del ejemplo anterior).

En cualquier momento podemos evitar que se siga propagando el evento ejecutando el método `.stopPropagation()` en el código de cualquiera de los escuchadores.

Podéis ver las distintas fases de un evento en la página domevents.dev.

6. innerHTML y escuchadores de eventos

Si cambiamos la propiedad *innerHTML* de un elemento del árbol DOM todos sus escuchadores de eventos desaparecen ya que es como si se volviera a crear ese elemento (y los escuchadores deben ponerse después de crearse).

Por ejemplo, tenemos una tabla de datos y queremos que al hacer doble click en cada fila se muestre su id. La función que añade una nueva fila podría ser:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = `<tr id="${data.id}"><td>${data.dato1}</td>
<td>${data.dato2}...</td></tr>`;
  miTabla.innerHTML += nuevaFila;
  document.getElementById(data.id).addEventListener('dblclick', event =>
alert('Id: '+ event.target.id));
```

Sin embargo esto sólo funcionaría para la última fila añadida ya que la línea `miTabla.innerHTML += nuevaFila` equivale a `miTabla.innerHTML = miTabla.innerHTML + nuevaFila`. Por tanto estamos asignando a *miTabla* un código HTML que ya no contiene escuchadores, excepto el de *nuevaFila* que lo ponemos después de hacer la asignación.

La forma correcta de hacerlo sería:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = document.createElement('tr');
  nuevaFila.id = data.id;
  nuevaFila.innerHTML = `<td>${data.dato1}</td><td>${data.dato2}...</td>`;
  nuevaFila.addEventListener('dblclick', event => alert('Id: ' +
event.target.id) );
  miTabla.appendChild(nuevaFila);
```

De esta forma además mejoramos el rendimiento ya que el navegador sólo tiene que renderizar el nodo correspondiente a la nuevaFila. Si lo hacemos como estaba al principio se deben volver a crear y a renderizar todas las filas de la tabla (todo lo que hay dentro de miTabla).

7. Eventos personalizados

También podemos mediante código lanzar manualmente cualquier evento sobre un elemento con el método `dispatchEvent()` e incluso crear eventos personalizados, por ejemplo:

```
const event = new Event('build');

// Listen for the event.
elem.addEventListener('build', (e) => { /* ... */ }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

Incluso podemos añadir datos al objeto *event* si creamos el evento con `new CustomEvent()`. Podéis obtener más información en la página de MDN.