Gestión de eventos

1. Eventos y programación tradicional

Los programas tradicionales como los que hemos hecho hasta ahora se limitan a ejecutar una secuencia concreta de instrucciones, una tras otra. Pero para ciertos tipos de aplicaciones esto puede no ser suficiente. Por ejemplo, podemos necesitar que nuestra aplicación haga algo cuando pulsamos un botón, o cuando pasamos el ratón por encima de una imagen. Es aquí donde debemos acudir a la programación dirigida por eventos (*event driven programming*).

Un evento, en general, es algo que sucede y que permite, a raíz de él, desencadenar una serie de acciones. Por ejemplo, cuando pulsamos el botón de cerrar en una ventana del ordenador, se produce un evento que ocasiona el cierre del programa en el que estamos. Cuando elegimos el menú Archivo > Guardar como... de una aplicación, se produce un evento que permite mostrar un cuadro de diálogo para elegir dónde queremos guardar lo que estamos haciendo.

Los eventos de JavaScript están a la espera de que suceda algo en la página web. Por ejemplo, que el usuario haga clic sobre algún elemento, o que pase el ratón por encima de alguna zona, o incluso eventos que no tienen que ver con acciones del usuario, como que el contenido de la página termine de cargarse. Una vez se produce un evento, podemos preparar a nuestro programa o aplicacion web para que responda con una acción determinada. Para ello, JavaScript permite asignar una función a cada uno de los posibles eventos que pueden producirse en una página, sobre cada elemento.

2. Principales tipos de eventos

Sobre una página web se pueden producir diferentes tipos de eventos. Veremos a continuación algunos de los más importantes, indicando sobre qué elementos se pueden producir.

2.1. Eventos sobre el contenido de la página (*body*)

Los siguientes eventos se producen sobre el contenido general de la página:

- **load**: se produce cuando la página termina de cargarse. Podemos emplearlo, por ejemplo, para hacer ciertas operaciones sobre el DOM, una vez sabemos que el contenido ya está cargado.
- **unload**: se produce cuando se cierra la página.
- **beforeunload**: se produce antes de cerrar la página. Es típico utilizarlo en páginas donde se pregunta al usuario si está seguro que quiere salir, porque hay cambios que no se hayan guardado.
- **resize**: se produce cuando cambia el tamaño de la ventana donde está el contenido

2.2. Eventos de teclado

Los siguientes eventos se producen con acciones desde el teclado sobre la página:

- **keydown** / **keyup**: se producen cuando se pulsa y suelta una tecla, respectivamente
- **keypress**: se produce cuando se pulsa y levanta una tecla (ambas acciones juntas), aunque está desaconsejado (*deprecated*) en versiones recientes de JavaScript, y se recomienda usar uno de los dos anteriores.

2.3. Eventos de ratón

Estos eventos se producen utilizando el ratón sobre la página:

- **click** / **dblclick**: cuando hacemos clic o doble clic con cualquier botón del ratón, respectivamente
- **mousedown** / **mouseup**: cuando pulsamos o levantamos un botón del ratón, respectivamente
- **mouseenter**: cuando entramos con el ratón en una zona
- **mouseleave**: cuando salimos con el ratón de una zona
- **mousemove**: cuando movemos el ratón dentro de una zona

2.4. Eventos de formulario

Estos eventos se aplican sobre elementos de formularios:

- **focus**: cuando un elemento del formulario tiene el foco. Por ejemplo, un cuadro de texto en el que acabamos de entrar
- **blur**: cuando un elemento del formulario pierde el foco
- **change**: cuando el contenido del elemento ha cambiado. Por ejemplo, al cambiar el texto de un cuadro de texto, o la opción seleccionada en una lista. En el caso de cuadros de texto también se puede usar el evento **input**.
- **submit**: cuando se envía el formulario

2.5. Otros eventos

Existen otros muchos eventos a los que podemos responder: abortar la carga de una imagen, eventos sobre pantallas táctiles, etc. Podéis consultar una lista más detallada aquí.

3. Uso básico de eventos

Como hemos visto, cada etiqueta o elemento de la página puede admitir un conjunto determinado de eventos, y responder ante ellos. Los eventos debemos asociarlos a funciones JavaScript para que puedan responder al mismo, y estas funciones se llaman manejadores de eventos (*event handlers*). Podemos definir estos manejadores de distintas formas.

3.1. Manejadores de eventos mediante atributos

Una primera forma de definir los manejadores de eventos consiste en utilizar una serie de atributos XHTML asociados a cada evento, y definir dentro el código JavaScript que queramos. Estos atributos tienen el mismo

nombre del evento, con el prefijo "on" delante. Por ejemplo, así advertiríamos de que el usuario sale de un formulario:

```
<form action="pagina.php" onmouseleave="alert('Saliendo del formulario');">
```

Es una alternativa poco recomendable, porque se mezcla demasiado el código JavaScript y el HTML, y es difícilmente mantenible. Como valor añadido, podemos utilizar el elemento **this** para acceder al elemento que ha provocado el evento. Este código cambia el color de fondo de un *div* a amarillo cuando entramos en él:

```
<div onmouseenter="this.style.backgroundColor='yellow';">...</div>
```

3.2. Manejadores de eventos mediante atributos y funciones

Una segunda forma algo más cómoda con respecto a la anterior consiste en utilizar el atributo XHTML para invocar a una función externa que haga el trabajo. De este modo, si el evento requiere de varios pasos complejos, no tenemos que ponerlos todos dentro del atributo, y el código JavaScript es más manejable. El ejemplo anterior de salida del formulario podríamos definirlo así: por un lado, definiríamos nuestra función JavaScript (en la propia página HTML o en un archivo aparte):

```
function mensajeSalida()
{
    alert("Saliendo del formulario");
}
```

Por otro lado, definiríamos así el atributo con el evento:

```
<form action="pagina.php" onmouseleave="mensajeSalida()">
```

En este caso, desde la función no podemos acceder al elemento *this* que provocó el evento, tendríamos que enviárselo como parámetro. Por ejemplo, así cambiaríamos el color de fondo de un div al entrar en él, usando funciones:

```
function cambiarColor(elemento)
{
    elemento.style.backgroundColor= "yellow";
}
```

```
<div onmouseenter="cambiarColor(this)">...</div>
```

3.3. Manejadores de eventos usando el DOM

En lugar de utilizar los atributos XHTML vistos antes, podemos acceder a los diferentes elementos del DOM y definir los eventos sobre ellos. Tiene la ventaja de que el código HTML queda intacto (no tenemos que alterarlo) y además podemos hacer referencia a la variable *this* para acceder al elemento que provocó el evento. Por ejemplo, de este modo definiríamos el evento de cambio de color sobre el *div* anterior. En primer lugar, debemos identificar el elemento de algún modo (típicamente con un *id*):

```
<div id="div1">...</div>
```

Después, accedemos al elemento desde JavaScript (por ejemplo, con `document.getElementById`), y añadimos el evento sobre el propio elemento:

```
let div = document.getElementById("div1");
div.onmouseenter = function() {
    this.style.backgroundColor = "yellow";
}
```

3.4. Manejadores de eventos con *event listeners*

Como cuarta y última forma de definir manejadores de eventos (y la más recomendable de todas) podemos definir *listeners* sobre los elementos de la página. Es algo parecido a la forma anterior usando el DOM, pero con algunas ventajas, como por ejemplo la posibilidad de definir distintos manejadores para un mismo evento.

Para utilizar esta opción, debemos emplear la instrucción `addEventListener`, pasándole 2 parámetros entre paréntesis:

- El nombre del evento (sin prefijo "on")
- La función que se va a ejecutar

El mismo evento anterior de entrada de ratón sobre el *div* lo podríamos definir así:

```
let div = document.getElementById("div1");
div.addEventListener('mouseenter', function() {
    this.style.backgroundColor = "yellow";
});
```

Como decíamos, podemos definir varias funciones manejadoras sobre el mismo evento. En este ejemplo cambiamos el color de fondo desde una función, y la anchura desde otra (aunque podríamos hacer las dos cosas desde la misma función también):

```
let div = document.getElementById("div1");
div.addEventListener('mouseenter', function() {
    this.style.backgroundColor = "yellow";
});
div.addEventListener('mouseenter', function() {
    this.style.width="50%";
});
```

También podemos definir las funciones aparte y referenciarlas desde *addEventListener*:

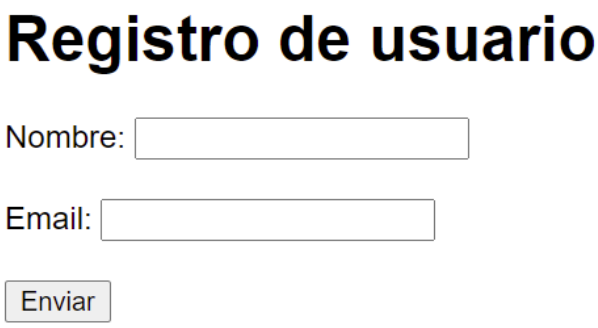
```
let cambiarColorFondo = function() {
    this.style.backgroundColor = "yellow";
}

let div = document.getElementById("div1");
div.addEventListener('mouseenter', cambiarColorFondo);
```

Puedes probar aquí un ejemplo de uso de eventos con cada una de las cuatro estrategias comentadas.

> **Ejercicio 1:**
>
> Crea una página llamada **eventos_imagenes.html**. Coloca en ella tres imágenes centradas, con un tamaño de 100 x 100 píxeles (mediante CSS). Haz que, al pasar el ratón por encima de cualquiera de ellas, se cambie su tamaño a 200 x 200 y se le ponga un borde rojo de 3 píxeles de grosor. Haz que este efecto desaparezca al salir el ratón de la imagen.

> **Ejercicio 2:**
>
> Crea una página llamada **evento_form.html** con un formulario como este:



> Define eventos JavaScript sobre los dos cuadros de texto para que:

- Si salimos del cuadro de texto (evento *blur*) y no tiene valor (propiedad *value* está vacía), se le ponga el color de fondo rojo al cuadro de texto. Si tiene valor, dejaremos el color de fondo blanco.
- Cuando entramos al cuadro de texto (evento *focus*), se le ponga el color de fondo amarillo.

4. Uso avanzado de eventos

4.1. Propagación de eventos

Cuando interactuamos con una página web, se producen una serie de eventos sobre los elementos de dicha página, como hemos podido ver en esta sesión. Cada vez que hacemos clic en un elemento, o escribimos en un control de formulario, o simplemente pasamos el ratón sobre una zona de la página, se desencadena un evento al que podemos responder.

Sin embargo, hemos de tener en cuenta que el contenido de una página está jerarquizado: los elementos que la componen están unos dentro de otros. Por ejemplo, una imagen puede encontrarse dentro de un párrafo, que a su vez se encuentre en un *div* o contenedor, que a su vez forme parte de una rejilla con otros contenedores. Esta jerarquía permite que varios elementos involucrados puedan responder a un evento; por ejemplo, si tenemos una imagen dentro de un párrafo, y a su vez dentro de un *div*, podemos hacer que, al pasar el ratón por una imagen, todos ellos (*div*, párrafo y/o imagen) puedan responder a este evento. Esto se puede afrontar desde dos puntos de vista:

- *Bubbling*: se responde al evento desde el elemento más específico (la imagen, en el ejemplo anterior) al menos específico (el *div*)
- *Capturing*: se responde al evento desde el elemento menos específico (el *div* en el ejemplo anterior) al más específico (la imagen)

Ejemplo de *bubbling*

Veamos el ejemplo anterior más concretamente. Imaginemos la siguiente estructura HTML:

```
<div id="div1">
    <p id="p1">
        <img id="img1" src="foto.png">
    </p>
</div>
```

Definimos en JavaScript el siguiente gestor de eventos `eventoClic`, y lo aplicamos por igual a los tres elementos:

```
let eventoClic = function() {
    alert("Has pulsado " + this.id);
}

let div = document.getElementById("div1");
let p = document.getElementById("p1");
let img = document.getElementById("img1");

div.addEventListener("click", eventoClic);
p.addEventListener("click", eventoClic);
img.addEventListener("click", eventoClic);
```

Con esta estructura, ocurre lo siguiente:

- Si hacemos clic en el *div* (fuera del párrafo y la imagen), saca el mensaje *Has pulsado div1*
- Si hacemos clic en el párrafo (fuera de la imagen), saca los mensajes *Has pulsado p1* y *Has pulsado div1*
- Si hacemos clic en la imagen, saca los tres mensajes: *Has pulsado img1*, *Has pulsado p1* y *Has pulsado div1*

Este es el comportamiento por defecto de los eventos, que se corresponde con el *bubbling*: se propagan desde el más específico donde se ha producido hasta el más general que lo contiene.

Ejemplo de *capturing*

Si queremos ofrecer el comportamiento opuesto, es decir, que los eventos se propaguen desde el más general al más específico, debemos proporcionar un tercer parámetro *true* al *event listener*. Nuestro ejemplo anterior quedaría así:

```
div.addEventListener("click", eventoClic, true);
p.addEventListener("click", eventoClic, true);
img.addEventListener("click", eventoClic, true);
```

Ahora, si hacemos clic en la imagen los mensajes se muestran en orden inverso: desde el div hasta la imagen. Lo mismo ocurre si hacemos clic en el párrafo (primero se muestra el mensaje del div y luego el del párrafo).

> **Ejercicio 3**
>
> Crea una página llamada **bubbling_capturing.html** y prueba el código de los ejemplos anteriores con la imagen que quieras. Define un color de fondo y un *padding* para el *div* y para el párrafo, de forma que te ayude a localizarlos mejor en la página y ver donde tienes que hacer clic en cada caso.

4.2. Obtener información del evento

En algunas ocasiones nos puede ser útil obtener información del evento que se produce. Por ejemplo, las coordenadas donde hemos hecho clic con el ratón, o qué tecla hemos pulsado en un evento de teclado. Para recoger información del evento podemos pasarlo como parámetro a la función manejadora.

```
let evento = function(event)
{
    ...
}
```

En la variable donde hemos recogido el evento (`event` en el ejemplo anterior) tendremos disponibles una serie de propiedades según el tipo de evento que haya sido. Algunas de las más útiles son:

- Si ha sido un evento de teclado (pulsar una tecla, por ejemplo), en la propiedad `key` tendremos el nombre de la tecla pulsada, y en `keyCode` tendremos el código interno de la tecla en Unicode, aunque esta última propiedad puede no funcionar adecuadamente en ciertos navegadores con *keyup* o *keydown*. También podemos detectar con algunos booleanos especiales como `shiftKey` , `ctrlKey` , etc, si se ha pulsado alguna tecla especial (como *Mayúsculas* o *Control*, por ejemplo):

```
document.addEventListener("keydown", function(evento)
{
    if (evento.ctrlKey && evento.key == 'z')
    {
        alert("Has pulsado Ctrl + Z");
    }
    else if (evento.shiftKey)
    {
        alert("Mayúsculas pulsadas");
    }
});
```

- Si ha sido un evento de ratón, en las propiedades `clientX` y `clientY` tendremos las coordenadas del ratón con respecto a la ventana del navegador, y en `screenX` y `screenY` tendremos las coordenadas respecto a la pantalla. El siguiente ejemplo recoge en qué coordenadas se hace clic con el ratón, en un documento.

```
document.addEventListener("click", function(event)
{
    let mouseX = event.clientX;
    let mouseY = event.clientY;
    alert(mouseX + ", " + mouseY);
}
```

> **Ejercicio 4:**
>
> Crea una página llamada **info_eventos.html**. Define en ella un div con un borde negro, y que dentro de ese div se informe de las coordenadas actuales del ratón dentro de la ventana. Si pulsamos cualquier tecla, deberemos mostrar un alert con su código.

4.2.1. Otras propiedades de los eventos

Además de las propiedades particulares de cada tipo de evento (coordenadas del ratón, tecla pulsada, etc) existen una serie de propiedades y operaciones generales para cualquier evento. Aquí mostramos algunas de las más importantes:

- **target**: hace referencia al elemento HTML sobre el que se ha producido el evento
- **type**: tipo de evento producido. Se tiene el nombre del evento, tal y como los hemos visto hasta ahora: *click*, *mouseenter*, etc.
- **preventDefault()**: esta operación es útil para eventos que desencadenan una acción por defecto. Por ejemplo, el evento *submit* cuando pulsamos el botón de un formulario, o el evento de cargar un enlace al hacer clic sobre él. Al llamar a esta instrucción evitamos que se ejecute esa acción predefinida. Será útil, como veremos más adelante, para prevenir que se envíen formularios incorrectos, entre otras cosas.
- **stopPropagation()**: evita que el evento se siga propagando a niveles superiores o inferiores (según si estamos en modo *bubbling* o *capturing*, respectivamente).

El siguiente manejador muestra el contenido HTML del elemento sobre el que se ha producido el evento, y el tipo de evento que se ha producido (*click* en este caso):

```
let informacion = function(event)
{
    alert(event.target.innerHTML);
    alert(event.type);
}
```

4.3. El evento de "arrastrar y soltar" (*drag & drop*)

El hecho de arrastrar elementos de una página de una zona a otra para ubicarlos en ciertos contenedores es algo habitual. Hasta hace no mucho, esto requería del uso de ciertas librerías externas como *jQuery* para simplificar la tarea, pero se ha convertido en algo tan habitual que la propia API de JavaScript ha incorporado esta característica al lenguaje, junto con HTML5.

El proceso se basa en tres eventos que debemos gestionar:

- `dragstart`: cuando comenzamos a arrastrar un componente. Típicamente lo vamos a utilizar para guardarnos la referencia del elemento que estamos arrastrando, para recuperarla después al soltar. Esta referencia suele ser el *id* del elemento.
- `dragover`: cuando estamos arrastrando el elemento por encima del componente donde lo podemos soltar. En este caso debemos evitar que se active la acción por defecto, que es rechazar el elemento,

usando `evento.preventDefault()`.

- `drop`: cuando finalmente se suelta el elemento. Si lo soltamos sobre un componente que no hemos configurado, el propio componente rechazará al elemento, que volverá a su lugar de origen. Pero si hemos configurado adecuadamente el evento *dragover* sobre ese elemento, podemos simplemente usar este otro evento para que ese componente "acoja" al elemento que hemos soltado, añadiéndolo como hijo (*appendChild*).
- Todo esto va unido a un atributo HTML llamado `draggable`, que debemos poner a *true* en los elementos que queramos poder arrastrar.

Aquí vemos un sencillo ejemplo donde podemos arrastrar el logo de JavaScript a una caja. El HTML puede quedar así:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo Drag & Drop</title>
    <style>
        #contenedor
        {
            display: grid;
            grid-template-columns: 1fr 1fr;
            gap: 20px;
        }
        #receptor
        {
            border: 1px solid black;
            border-radius: 10px;
            height: 300px;
            text-align: center;
            vertical-align: middle;
        }
    </style>
</head>
<body>
    <div id="contenedor">
        <div>
            <img id="logo" draggable="true"
                src="javascript_logo.png">
        </div>
        <div id="receptor">
        </div>
    </div>
</body>
</html>
```

Y el código JavaScript asociado sería este:

```
let imagen = document.getElementById("logo");
let contenedor = document.getElementById("receptor");

imagen.addEventListener("dragstart", function(evento) {
    evento.dataTransfer.setData("elemento", evento.target.id);
});
contenedor.addEventListener("dragover", function(evento) {
    evento.preventDefault();
});
contenedor.addEventListener("drop", function(evento) {
    evento.preventDefault();
    let datos = evento.dataTransfer.getData("elemento");
    evento.target.appendChild(document.getElementById(datos));
});
```

Explicamos el código que hemos definido antes:

- Definimos el atributo `draggable` sobre la imagen para poderla arrastrar
- Definimos el evento `dragstart` sobre la imagen, y guardamos en el evento el *id* de la imagen, en su propiedad *dataTransfer*, asignándole el nombre *elemento*, por ejemplo. Dicha propiedad se utiliza para almacenar y recuperar información utilizada durante el proceso de arrastrar y soltar.
- Definimos el evento `dragover` sobre la caja para que, cuando algo esté "sobrevolándola" no lo rechace (*preventDefault*)
- Definimos el evento `drop` también sobre la caja para anular el evento por defecto (*preventDefault*), acceder al elemento que hemos estado arrastrando (recuperamos la información del *dataTransfer*) y añadirlo (*appendChild*) al contenido de la caja. Puede resultar extraño usar aquí *preventDefault* también, pero es que en algunos navegadores la acción por defecto cuando se arrastra un componente en alguna parte de la página es mostrar ese componente (en este caso, mostrar la imagen) a pantalla completa.

> **Ejercicio 5**
>
> Crea una página llamada **drag_drop_js.html** y utiliza esta plantilla como base. Se pide que añadas el código JavaScript necesario para poder arrastrar los logos de los navegadores cada uno a su casilla correspondiente. En principio puedes arrastrar cualquier logo a cualquier casilla, pero al pulsar el botón de *Comprobar* se debe mostrar un alert indicando si has acertado o no.

4.4. Encapsulando el código JavaScript

El uso de JavaScript en páginas HTML es una herramienta muy potente, que nos permite enriquecer el comportamiento de las páginas añadiendo elementos dinámicamente: modificar contenidos, responder a eventos de usuario, etc. Sin embargo, en ocasiones puede ser conveniente proteger nuestro código JavaScript, de forma que quede aislado. Esto puede resultar particularmente útil cuando accedemos a distintos archivos JavaScript, para así evitar colisiones en nombres de variables o funciones. Para ello, tenemos dos alternativas.

Evento *load*

Como primera alternativa, podemos hacer que nuestro código se cargue dentro del evento *load* de la página. Esto hará, por un lado, que ciertas instrucciones no se ejecuten hasta que los contenidos estén cargados (útil si queremos acceder a partes del DOM), y por otro lado, que todo el código quede encapsulado en el evento en sí, y no sea accesible desde fuera.

El ejemplo anterior de *bubbling* y *capturing* lo podríamos definir así dentro del evento *load* de la ventana (y por tanto, este código lo podríamos poner tanto en la cabecera como en el cuerpo de la página):

```
window.addEventListener('load', function()
{
    let eventoClic = function() {
        alert("Has pulsado " + this.id);
    }

    let div = document.getElementById("div1");
    let p = document.getElementById("p1");
    let img = document.getElementById("img1");

    div.addEventListener("click", eventoClic);
    p.addEventListener("click", eventoClic);
    img.addEventListener("click", eventoClic);
});
```

IIFE

Como alternativa a la opción anterior, también podemos emplear una estructura normalmente conocida como **IIFE** (*Immediately Invoked Function Expression*). Básicamente consiste en encapsular todo nuestro código en una función que se auto-ejecuta. De este modo, el código interno a la función no es visible desde fuera, y evitamos así que colisione con otros fragmentos de código JavaScript que podamos querer incorporar.

Así quedaría el ejemplo anterior de *bubbling* y *capturing* encapsulado con IIFE:

```
(function()
{
    let eventoClic = function() {
        alert("Has pulsado " + this.id);
    }

    let div = document.getElementById("div1");
    let p = document.getElementById("p1");
    let img = document.getElementById("img1");

    div.addEventListener("click", eventoClic);
    p.addEventListener("click", eventoClic);
    img.addEventListener("click", eventoClic);
})();
```

Observemos que hemos encapsulado el código en una función que se auto-ejecuta. Esto hace que los elementos que hemos definido (función *eventoClic*, variables *div*, *p*, *img*...) no puedan ser accesibles desde fuera de este código. Además, este código se ejecuta *inmediatamente*, con lo que, en este caso, convendría colocarlo después de cargar la página (para poder acceder a los elementos del DOM).

> **Ejercicio 6**
>
> Crea una versión del ejercicio 4 en otro archivo llamado **info_eventos_iife.html** y encapsula el código JavaScript utilizando IIFE.