Eventos (05-10-23)

Que son los eventos

En Javascript existe un concepto llamado evento, que no es más que una notificación de que alguna característica interesante acaba de ocurrir, generalmente relacionada con el usuario que navega por la página.

Como desarrolladores, nuestro objetivo es preparar nuestro código para que cuando ocurra un determinado evento, se lleve a cabo una funcionalidad asociada. De esta forma, podemos preparar nuestra página o aplicación para que cuando ocurran ciertos eventos (que no podemos predecir de otra forma), reaccionen a ellos.

Dichas características pueden ser muy variadas:

- Click de ratón del usuario sobre un elemento de la página
- Pulsación de una tecla específica del teclado
- Reproducción de un archivo de audio/video
- Scroll de ratón sobre un elemento de la página
- El usuario ha activado la opción "Imprimir página"

Formas de manejar eventos

Existen varias formas diferentes de manejar eventos en Javascript. Cada una de estas opciones se puede utilizar para gestionar eventos en Javascript de forma equivalente, pero cada una de ellas tiene sus ventajas y sus desventajas.

A continuación, utilizaremos el evento clic para demostrar las formas de manejar los eventos.

1. **Mediante atributos HTML:**

Suponiendo el HTML para el elemento botón:

```
<button>Saludar</button>
```

Como cualquier otro elemento, podemos modificar sus atributos directamente en la etiqueta:

```
<button onClick="alert('Hello!')">Saludar</button>
```

De este modo, cuando el usuario haga click con el ratón en el botón "Saludar", se disparará el evento click en ese elemento HTML. Dicho botón, al tener un atributo `onClick` (cuando hagas click), ejecutará el código que tenemos asociado en el valor del atributo HTML.

Otra manera de lograr este mismo resultado es reorganizando la funcionalidad de manera que el código que contendrá la función a ejecutar esté separado del llamado de dicha función en la propiedad HTML. Así, el código es más limpio al tener una función con más de una línea de código:

```
<script>
  function doTask() {
    alert("Hello!");
  }
</script>

<button onClick="doTask()">Saludar</button>
```

Ahora sí, todo está un poco mejor organizado. Sin embargo, no es muy habitual tener bloques `<script>` de código Javascript en nuestro HTML, sino que lo habitual suele ser externalizarlo en ficheros `.js` para dividir y organizar mejor nuestro código:

```
<script src="tasks.js"></script>
<button onClick="doTask()">Saludar</button>
```

Ahora aparece un nuevo problema que quizás puede que aún no sea muy evidente. En nuestro `<button>` estamos haciendo referencia a una función llamada `doTask()` que, aparentemente, confiaremos en que se encuentra declarada dentro del fichero `tasks.js`.

Esto podría convertirse en un problema, si posteriormente, o dentro de cierto tiempo, nos encontramos modificando código en el fichero `tasks.js` y le cambiamos el nombre a la función `doTask()`, ya que podríamos olvidar que hay una llamada a una función Javascript en uno (o varios) ficheros `.html`.

---

2. **Mediante propiedades JavaScript:**

La idea es similar a lo comentado anteriormente, prefiriendo la externalización del código a ficheros `.js`.

Suponiendo el mismo botón como elemento HTML:

```
<button>Saludar</button>

<script>
const button = document.querySelector("button");
button.onclick = function() {
  alert("Hello!");
}
</script>
```

Observa que en este caso, en lugar de añadir el atributo onClick a nuestro `<button>`, lo que hacemos es localizarlo mediante `querySelector()`, que supone una alternativa a `getElementById()`.

> Nota: La propiedad `.onclick` (o del evento en cuestión) siempre irá en minúsculas, ya que se trata de una propiedad Javascript, y Javascript es sensible a mayúsculas y minúsculas.

Otra forma de hacer uso de las propiedades de Javascript, es utilizar el método `setAttribute()` para el nodo del DOM que deseamos agregarle el evento:

```
<button>Saludar</button>

<script>
const button = document.querySelector("button");
const doTask = () => alert("Hello!");
button.setAttribute("onclick", "doTask()");
</script>
```

Este caso es similar al del punto 1, ya que se le asigna directamente al atributo `onclick` del elemento HTML una función Javascript.

> Nota: `const doTask = () => alert("Hello!");` es una forma de declarar funciones conocida como *Arrow Function*. Entre los paréntesis, se asignan los parámetros de la función (en este caso, ninguno), y seguido a la flecha "`=>`" las líneas de código de la función (que, si es una, no necesita llaves; si son más de una, se coloca entre llaves).

---

3. **Mediante escuchadores de eventos:**

Con el método .addEventListener() permite añadir una escucha del evento indicado (primer parámetro), y en el caso de que ocurra, se ejecutará la función asociada indicada (segundo parámetro).

```
button.addEventListener(event, func)
```

De este modo, se escucha el evento `event`, y si ocurre, ejecuta `func`.

Continuando con el mismo ejemplo:

```
const button = document.querySelector("button");
button.addEventListener("click", function() {
  alert("Hello!");
});
```

Aunque es muy habitual escribir los eventos de esta forma, es posible que veas mucho más organizado este código si sacamos la función y la guardamos en una constante previamente, para luego hacer referencia a ella desde el `.addEventListener()`:

```
const button = document.querySelector("button");
function action() {
  alert("Hello!");
};
button.addEventListener("click", action);
```

Si prefieres utilizar las funciones flecha de Javascript, quedaría incluso más legible:

```
const button = document.querySelector("button");
const action = () => alert("Hello!");
button.addEventListener("click", action);
```

Sin embargo, una de las características más cómodas de utilizar `.addEventListener()` es que puedes añadir múltiples listeners de una forma muy sencilla.

Dicho método, permite asociar múltiples funciones a un mismo evento, algo que, aunque no es imposible, es menos sencillo e intuitivo en las modalidades de gestionar eventos que vimos anteriormente:

```
<button>Saludar</button>

<style>
  .red { background: red }
</style>

<script>
const button = document.querySelector("button");
const action = () => alert("Hello!");
const toggle = () => button.classList.toggle("red");

button.addEventListener("click", action); // Hello message
button.addEventListener("click", toggle); // Add/remove red CSS
</script>
```

Aparte del evento a escuchar y la función a ejecutar, existe un tercer parámetro considerado opcional para el método `addEventListener()`.

Este es un parámetro de **opciones**, el cual es un objeto donde se indica una de las siguientes opciones para modificar alguna característica del listener en cuestión:

- `capture`. El evento se dispara al inicio (`capture`) en lugar de al final (`bubble`).
- `once`. Solo ejecuta la función la primera vez. Luego, el listener es eliminado.
- `passive`. La función nunca llama a `.preventDefault()` (mejora el rendimiento).

> Nota: `.preventDefault()` Cancela el evento si este es cancelable, sin detener el resto del funcionamiento del evento, es decir, puede ser llamado de nuevo.