DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Eventos

Un evento es una señal de que algo ha sucedido. Todos los nodos DOM generan tales señales (pero los eventos no se limitan a DOM).
Aquí hay una lista de los eventos DOM más útiles:

Eventos del mouse:

- **click** - cuando se hace clic en un elemento (los dispositivos con pantalla táctil lo generan con un toque).
- **contextmenu** - cuando se hace clic derecho en un elemento.
- **mouseover/ mouseout**- cuando el ratón esta sobre un elemento o sale de él.
- **mousedown/ mouseup**- cuando se presiona / suelta el botón del ratón sobre un elemento.
- **mousemove** - cuando se mueve el ratón.

Eventos de teclado:

- **Keydowny** y **keyup**- cuando se presiona y suelta una tecla del teclado.

Eventos de elementos de formulario:

- **submit-** cuando el visitante envía a <form>.
- **reset** cuando borramos el contenido de un formulario
- **focus**- cuando el visitante se centra en un elemento, por ejemplo, en un <input>.

Documentar eventos:

- **DOMContentLoaded (load**) - cuando se carga y procesa el HTML, DOM está completamente construido.

Eventos CSS:

- **transitionend** - cuando termina una animación CSS.

Hay muchos otros eventos. Entraremos en más detalles de eventos particulares en los próximos capítulos.

2.- Controladores de eventos

Para reaccionar ante los eventos, podemos asignar un **controlador**, una función que se ejecuta en caso de un evento. Los controladores son una forma de ejecutar código JavaScript en función de las acciones del usuario. Hay varias formas de asignar un controlador.

2.1.- Atributo HTML

Se puede establecer un controlador en HTML con un atributo denominado **on<event>.**

Por ejemplo, para asignar un controlador click para un input, podemos usar onclick, como aquí:

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

Al hacer clic con el ratón, se ejecuta el código interno .
En el interior de onclick usamos comillas simples, porque el atributo en sí está entre comillas dobles. Si olvidamos que el código está dentro del atributo y usamos comillas dobles, como este: `onclick="alert("Click!")"` entonces no funcionará correctamente.

Un atributo HTML no es un lugar conveniente para escribir mucho código, por lo que es mejor que creemos una función de JavaScript y la llamemos allí.

En este ejemplo llamamos a la función countRabbits():

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Rabbit number " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">
```

Como sabemos, los nombres de atributos HTML no distinguen entre mayúsculas y minúsculas, por lo que ONCLICK funciona tan bien como onClick e onCLICK... Pero por lo general serán minúsculas atributos: onclick.

2.2.- Propiedad DOM

Podemos asignar un controlador utilizando la propiedad DOM **on<event>.**

Por ejemplo elem.onclick:

```
<input id="elem" type="button" value="Click me">
<script>
  elem.onclick = function() {
    alert('Thank you');
  };
</script>
```

Si el controlador se asigna utilizando un atributo HTML, el navegador lo lee, crea una nueva función a partir del contenido del atributo y lo escribe en la propiedad DOM. Entonces, esta forma es realmente la misma que la anterior.

Estas dos ejemplos funcionan igual:

Solo HTML:

```
<input type="button" onclick="alert('Click!')" value="Button">
```

HTML + JS:

```
<input type="button" id="button" value="Button">
<script>
  button.onclick = function() {
    alert('Click!');
  };
</script>
```

En el primer ejemplo, el atributo HTML se usa para inicializar button.onclick, mientras que en el segundo ejemplo, se hace en el script, esa es la diferencia.

Se recomienda utilizar la opción DOM porque de esta manera estamos separando el código HTML de código JS, de esta manera podemos trabajar en paralelo HTML (diseñadores) y Javascript (programadores)

Como solo hay una propiedad onclick, no podemos asignar más de un controlador de eventos.

En el siguiente ejemplo, agregar un controlador con JavaScript sobrescribe el controlador existente:

```
<input type="button" id="elem" onclick="alert('Before')" value="Click me">
<script>
  elem.onclick = function() { // overwrites the existing handler
    alert('After'); // only this will be shown
  };
</script>
```

Para eliminar un controlador, asignamos

```
elem.onclick = null.
```

Uso de this en el controlador

El valor de this dentro de un controlador es el elemento. Cuando asignemos una función como controlador de evento de un elemento, dentro del controlador this hace referencia al elemento sobre el que se ha producido el evento.

Por ejemplo podríamos tener varios botones con el mismo controlador, al hacer click sobre ellos todos ejecutarían el mismo código. Para saber que botón es el que ha hecho click utilizaremos this, de esa manera podemos ejecutar el mismo controlador para todos los botones y a la vez saber cuál ha hecho la llamada.

```
<button value="Boton 1">Boton 1</button>
<button value="Boton 2">Boton 2</button>
<button value="Boton 3">Boton 3</button>
<script>
  function hacerClick(){
    alert("Has pulsado: "+ this.value);
  }

  let misBotones=document.getElementsByTagName("button");
  botonesArray=Array.from(misBotones);
  botonesArray.forEach(b=>b.onclick=hacerClick);
</script>
```

En el ejemplo anterior todos los botones tienen asignado el mismo controlador y podemos saber con this quien está ejecutándose por si hiciera falta realizar acciones diferentes en función del botón.

En el siguiente código button muestra su contenido utilizando this.innerHTML:

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

Posibles errores

Si comenzamos a trabajar con eventos tenemos que prestar a tención a un par de detalles muy importantes:

1. **Asignación desde DOM de la función como controlador del evento**

   Podemos establecer una función existente como controlador:

   ```
   function sayThanks() {
     alert('Thanks!');
   }
   elem.onclick = sayThanks;
   ```

   La función debe asignarse como sayThanks, no sayThanks().

   // right

   ```
   button.onclick = sayThanks;
   ```

   // wrong

   ```
   button.onclick = sayThanks();
   ```

   Si agregamos paréntesis, sayThanks() se convierte en una llamada de función. Entonces, la última línea realmente toma el resultado de la ejecución de la función, es decir undefined (ya que la función no devuelve nada), y se la asigna onclick. Eso no funciona

2. **Asignación dentro de la etiqueta HTML**

   En este caso si que debemos incluir el nombre de la función y los paréntesis. Ambos además entre comillas

   ```
   <input type="button" id="button" onclick="sayThanks()">
   ```

   La diferencia con el caso anterior es que cuando el navegador lee el atributo de la etiqueta HTML, crea una función de controlador con cuerpo a partir del contenido del atributo.

Entonces el HTML genera esta propiedad:

```
button.onclick = function() {
  sayThanks(); // <-- the attribute content goes here
};
```

2.3.- Listener de eventos

El problema fundamental de las formas que hemos visto de asignar controladores es que no podemos asignar varios controladores a un evento.
Digamos que una parte de nuestro código quiere resaltar un botón al hacer clic y otra quiere mostrar un mensaje con el mismo clic.
Nos gustaría asignar dos controladores de eventos para eso. Pero una nueva propiedad DOM sobrescribirá la existente:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // replaces the previous handler
```

Los desarrolladores de estándares web lo entendieron hace mucho tiempo y sugirieron una forma alternativa de administrar manejadores utilizando métodos especiales **addEventListener** y **removeEventListener**.

La sintaxis **para agregar un controlador**:

```
element.addEventListener(event, handler, [useCapture]);
```

- **event**
  Nombre del evento, por ej "click".
- **handler**
  La función del controlador.
- **useCapture**
  La fase donde manejar el evento, es un booleano que indica que tipo de propagación se realizara del evento entre los componentes del DOM que estén afectados (**bubbling** and **capturing**).

La sintaxis **para eliminar un controlador**:

```
element.removeEventListener(event, handler, [useCapture]);
```

Para eliminar un controlador, debemos pasar exactamente la misma función que se le asignó.

Esto no funciona:

```
elem.addEventListener( "click" , () => alert('Thanks!'));
// ....
elem.removeEventListener( "click", () => alert('Thanks!'));
```

El controlador no se eliminará porque removeEventListener obtiene otra función, con el mismo código, pero eso no importa, ya que es un objeto de función diferente.

Aquí está la manera correcta:

```
function handler() {
  alert( 'Thanks!' );
}

input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

Si no almacenamos la función en una variable, no podemos eliminarla. No hay forma de "leer de nuevo" los controladores asignados por addEventListener.

Múltiples llamadas para addEventListenerpermitir agregar múltiples controladores, como este:

```
<input id="elem" type="button" value="Click me"/>

<script>
  function handler1() {
    alert('Thanks!');
  };

  function handler2() {
    alert('Thanks again!');
  }
```

```
  elem.onclick = () => alert("Hello");
  elem.addEventListener("click", handler1); // Thanks!
  elem.addEventListener("click", handler2); // Thanks again!
</script>
```

Como podemos ver en el ejemplo anterior, podemos establecer los manipuladores de ambos utilizando una propiedad DOM y addEventListener. Pero generalmente usamos solo una de estas formas. Para algunos eventos, los controladores solo funcionan con addEventListener. Existen eventos que no se pueden asignar a través de una propiedad DOM. Sólo con addEventListener.

Con lo cual **addEventListener es más universal y generalmente es el más utilizado**, dejando el controlador de eventos mediante propiedad DOM para elementos creados dinámicamente y poder asignarles en el momento de su creación su controlador de evento

3.- Propagación de eventos: Bubbling and capturing

Hay dos formas de propagación de eventos en HTML DOM, **bubbling** y **capturing**.

La propagación de eventos es una forma de definir el orden de los elementos cuando ocurre un evento. Si tenemos un elemento <p> dentro de un elemento y el usuario hace clic en el elemento <p>, ¿qué evento "clic" de elemento debe manejarse primero?

- Con **bubbling** el evento del elemento más interno (el hijo) se maneja primero y luego el externo (el padre o contenedor): el evento de clic del elemento <p> se maneja primero, luego el evento de clic del elemento .
- Con **capturing** el evento del elemento más externo (el padre o contenedor) se maneja primero y luego el interno (el hijo): el evento de clic del elemento se manejará primero, luego el evento de clic del elemento <p>.

Con el método addEventListener () podemos especificar el tipo de propagación utilizando el parámetro "useCapture":

```
addEventListener(event, function, useCapture);
```

El valor predeterminado es falso, que utilizará la propagación bubbling, cuando el valor se establece en verdadero, el evento utiliza la propagación capturing.

```
<style>
 #myDiv1, #myDiv2 {background-color: coral;padding: 50px;}
 #myP1, #myP2 {background-color: white; font-size: 20px;
                        border: 1px solid; padding: 20px;}
</style>
<body>
 <h2>JavaScript addEventListener()</h2>

 <div id="myDiv1">
   <h2>Bubbling:</h2>
   <p id="myP1">Click me!</p>
 </div><br>

 <div id="myDiv2">
   <h2>Capturing:</h2>
   <p id="myP2">Click me!</p>
 </div>

<script>
function clickOnP(){
 alert("Click en Parrafo blanco");
}
function clickOnDiv(){
 alert("Click en DIV naranja");
}
```

```
document.getElementById("myP1").addEventListener("click",clickOnP,false);
document.getElementById("myDiv1").addEventListener("click", clickOnDiv,false);

document.getElementById("myP2").addEventListener("click", clickOnP, true);
document.getElementById("myDiv2").addEventListener("click", clickOnDiv, true);
</script>

</body>
</html>
```

En este ejemplo hemos puesto dos div con fondo naranja y dentro de cada div un párrafo con color blanco. Hemos programado los eventos click de los div y los dos párrafos mediante un listener y las funciones controladoras cilckOnP para los párrafos y clickOnDiv para los divs.
En el primer div y párrafo usamos en el listener useCapture con el valor de false, con lo cual estamos usando la propagación de eventos de bubbling, de esta manera cuando hagamos click sobre el párrafo se ejecutará el controlador del evento click del párrafo y después el controlador del evento click del div.
En el segundo div y párrafo usamos en el listener useCapture con el valor de true, con lo cual estamos usando la propagación de eventos de capturing, de esta manera cuando hagamos click sobre el párrafo se ejecutará el controlador del evento click del div (aunque hemos hecho click primero sobre el párrafo) y después el controlador del evento click del párrafo.

4.- El objeto Event

Hay una serie de eventos que además de controlar que han sucedido y ejecutar las acciones correspondientes desde su controlador necesitamos saber información acerca del propio evento. Este tipo de situación está asociada principalmente a los eventos de teclado y ratón.

Para poder manejar adecuadamente un evento, a veces queremos saber más sobre lo que sucedió. No solo un "clic" o un "keydown", sino ¿cuáles eran las coordenadas del puntero?, ¿Qué tecla fue presionada?, ¿Con que botón del ratón se hizo click?...

Cuando ocurre un evento, el navegador crea un objeto de tipo evento, le pone toda la información detallada del evento y lo pasa como un argumento al controlador.

Aquí hay un ejemplo de cómo obtener coordenadas de puntero del objeto de evento:

```
<input type="button" value="Click me" id="elem">

<script>
  elem.onclick = function(event) {
    // show event type, element and coordinates of the click
    alert(event.type + " at " + event.currentTarget);
    alert("Coordinates: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Algunas propiedades del objeto event:

- **event.type**
  Tipo de evento, en el ejemplo "click".
- **event.currentTarget**
  Elemento que manejó el evento. Eso es exactamente lo mismo que this, a menos que el controlador sea una función de flecha, o this esté vinculado a otra cosa, entonces podemos obtener el elemento event.currentTarget.
- **event.clientX / event.clientY**
  Coordenadas relativas a la ventana del cursor, para eventos de puntero.

Hay más propiedades. Muchos de ellos dependen del tipo de evento: los eventos de teclado tienen un conjunto de propiedades. Así para cada tipo

4.1.- El objeto event en controladores HTML

Si asignamos un controlador en HTML, también podemos usar el objeto event, así:

```
<input type="button" onclick="alert(event.type)" value="Event type">
```

Eso es posible porque cuando el navegador lee el atributo, se crea un controlador de la siguiente manera:

```
function(event) { alert(event.type) }.
```

Es decir: se crea una función para el controlador del evento con un argumento event y el código que se pone en la etiqueta pasa al cuerpo de esa función. De esta manera el objeto event está disponible para nuestro código

4.2.-El objeto event en listeners: handleEvent

Podemos asignar **no solo una función, sino un objeto** como controlador de eventos mediante addEventListener. Cuando ocurre un evento, se llama a su método handleEvent.

Por ejemplo:

```
<button id="elem">Click me</button>

<script>
  let obj = {
    handleEvent(event) {
      alert(event.type + " at " + event.currentTarget);
    }
  };

  elem.addEventListener('click', obj);
</script>
```

Como podemos ver, cuando addEventListener recibe un objeto como controlador, llama a obj.handleEvent(event) en caso de un evento.

No es necesario realizar la llamada con un objeto, podemos utilizar un listener de eventos y como controlador una función como hemos visto anteriormente, y de forma similar a como se hace en la asignación de un controlador desde el HTML el objeto event pasara a estar disponible para el código de la función.

```
<button id="elem">Click me</button>

<script>

 function handler(){
   alert(event.type + " at " + event.currentTarget);
 }

 elem.addEventListener('click', handler);
</script>
```

En funciones que gestionan eventos, el objeto event está disponible aunque no se pase como parámetro, ya que JavaScript se encarga de que esté disponible para el código del controlador.

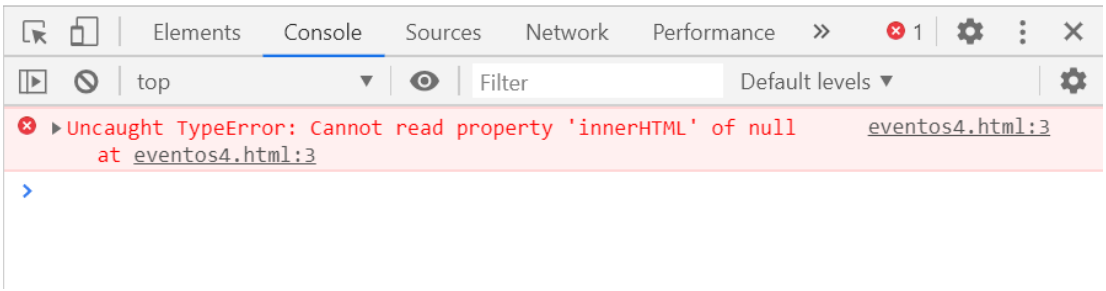
5.- El evento onload

En el tema anterior vimos el DOM y vimos que para poder acceder a los elementos del DOM es necesario que se construya el árbol, para ello se debe de cargar la página web, en ese momento estarán disponibles todos los elementos para poder acceder a ellos mediante código.

En el tema anterior debíamos poner los scripts debajo del body, porque al ser un lenguaje interpretado si ponemos el script antes del body e intentamos utilizar el DOM, como no se ha cargado la página aún, los elementos no existen y cualquier operación sobre ellos nos devolverá un error por null.

```
<script>
 let myP1=document.getElementById("myP1");
 alert(myP1.innerHTML);
</script>

<body>
 <p id="myP1">Soy el parrafo1</p>
</body>
</html>
```



La solución a este problema era poner los scripts después de body.

Para evitar esta situación disponemos del evento onload que indica que ya ha terminado la carga del objeto sobre el que se realiza el evento. En nuestro caso al querer estar seguros de que la página se ha cargado completamente deberíamos realizar el control sobre el elemento **body** o sobre el objeto **window**. Podemos hacerlo de cualquiera de las formas vistas para asignar controladores de eventos.

5.1.- Atributo HTML

```
<script>
  function inicio(){
   let myP1=document.getElementById("myP1");
   alert(myP1.innerHTML);
  }
</script>

<body onload="inicio()">
  <p id="myP1">Soy el parrafo1</p>
</body>
```

En este caso el el atributo onclick de la etiqueta body llamamos a la función inicio que ya podrá acceder a los elementos del DOM

5.2.- Propiedad DOM

En este caso no podemos utilizar onload sobre body porque tendríamos el problema que body aún no se habría cargado con lo cual nos daría un error de null por body. Para esta situación utilizaremos el objeto **window** que define el contenedor donde se carga nuestra página web.

Podemos hacerlo de todas las formas que hemos visto y usando la sintaxis de implementación de funciones vista

Usando una función existente

```
<script>
  window.onload=inicio;

  function inicio(){
    let myP1=document.getElementById("myP1");
    alert(myP1.innerHTML);
  }
</script>

<body>
  <p id="myP1">Soy el parrafo1</p>
</body>
```

Usando una función anónima

```
<script>
  window.onload=function(){
    let myP1=document.getElementById("myP1");
    alert(myP1.innerHTML);
  }
</script>

<body>
  <p id="myP1">Soy el parrafo1</p>
</body>
```

Con sintaxis fat arrow

```
<script>
  window.onload=()=>{
    let myP1=document.getElementById("myP1");
    alert(myP1.innerHTML);
  }
</script>

<body>
  <p id="myP1">Soy el parrafo1</p>
</body>
```

5.3.- Listener de eventos

En este caso también gestionaremos el evento onload sobre el objeto window. Podemos hacerlo de todas las formas que hemos visto y usando la sintaxis de implementación de funciones vista

Usando una función existente como controlador

```
<script>
 window.addEventListener("load",inicio);

 function inicio(){
  let myP1=document.getElementById("myP1");
  alert(myP1.innerHTML);
 }
</script>

<body>
 <p id="myP1">Soy el parrafo1</p>
</body>
```

Usando una función anónima en addEventListener

```
<script>
 window.addEventListener("load",function(){
                                    let myP1=document.getElementById("myP1");
                                    alert(myP1.innerHTML);
                                   });
</script>

<body>
 <p id="myP1">Soy el parrafo1</p>
</body>
```

Con sintaxis fat arrow

```
<script>
 window.addEventListener("load",()=>{
                                    let myP1=document.getElementById("myP1");
                                    alert(myP1.innerHTML);
                                  });
</script>

<body>
 <p id="myP1">Soy el parrafo1</p>
</body>
```