DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Eventos

Un **evento** en JavaScript es una acción u ocurrencia que sucede en un documento HTML, como hacer clic en un botón, mover el ratón, presionar una tecla o cargar una página. Estos eventos permiten que nuestras páginas web sean interactivas y respondan a las acciones del usuario.

Podríamos hacer una analogía de una página web con una casa

- **Elementos HTML:** Son las habitaciones, muebles y objetos de la casa.
- **Eventos:** Son las acciones que ocurren dentro de la casa, como encender una luz, abrir una puerta o tocar un timbre.
- **JavaScript:** Es el dueño de la casa que responde a estas acciones.

Existen muchos tipos de eventos, cada uno asociado a una acción específica, algunos de los más comunes son:

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:

- **keydown:** cuando se presiona una tecla.
- **keyup:** cuando se suelta una tecla.
- **keypress:** mientras se mantiene una tecla presionada

Eventos de elementos de formulario:

- **submit:** cuando el usuario 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>.
- **change:** cuando cambia el valor de algún elemento del formulario

Eventos DOM:

- **load:** cuando se carga y procesa el HTML, DOM está completamente construido.
- **resize:** cuando cambia el tamaño de la ventana del navegador.
- **scroll:** cuando se desplaza la página.

2.- Controladores de eventos

Para reaccionar ante los eventos, podemos asignar un **controlador**, una función que se ejecuta cuando sucede 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:

- Atributo HTML
- Propiedad DOM
- Listener de eventos

Atributo HTML

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

Por ejemplo, para asignar un controlador click para un botón

```
<button class="btn btn-primary"
        onclick="alert('Hola...')">Saludar</button>
```

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.

```
<body>
  <h3>Demo de eventos</h3>
```

```
  <button class="btn btn-primary" onclick="saludar()">Saludar</button>
</body>

<script>
  function saludar(){
    alert ("Hola...")
  }
</script>
```

Al hacer clic con el ratón, se ejecuta el código de la función indicada.

Propiedad DOM

En este caso será a través del DOM, Modelo de Objetos de Documento, la manera en la que asignaremos el controlador del evento a nuestro elemento. Para ello deberemos seleccionar el elemento al que queremos asignar el controlador y asignarle el controlador utilizando la propiedad DOM **on<event>.**

Vamos a probar el ejemplo anterior utilizando la asignación del controlador con el DOM

```
<body>

  <button class="btn btn-primary" id="miBoton">Saludar</button>

</body>
<script>
  let miBoton=document.getElementById("miBoton")
  miBoton.onclick=saludar()

  function saludar(){
    alert ("Hola...")
  }
</script>
```

Aparentemente arece que todo está bien, pero no es así. **Este ejemplo no va a realizar lo que queremos**.

Si lo probamos veremos que el mensaje de “Hola…” se muestra nada más cargarse la página, no cuando pulsemos el botón que es lo que nosotros queremos.

Esto es debido a que estamos asignando a un elemento del DOM, en concreto a miBoton la función saludar(). Como lo estamos haciendo, **le estamos diciendo a JavaScript que ejecute la función saludar() y guarde su valor en la variable miBoton.** Como la ejecución de la función saludar muestra el mensaje, al ejecutarse en la asignación muestra el alert. **Nosotros lo que queremos realmente es que el elemento miBoton (que es un objeto con propiedades y métodos) tenga en su evento onclick (que es un método) el código de la función saludar**.

Para realizar esto deberemos asignar a nuestro elemento la función y no la ejecución de la función, para ello deberemos eliminar en la asignación los paréntesis

```
miBoton.onclick=saludar
```

Otra alternativa con el DOM es asignar el evento con una función anónima

```
<body>
  <button class="btn btn-primary" id="miBoton2">Saludar</button>
</body>

<script>
  let miBoton2=document.getElementById("miBoton2")
  miBoton2.onclick=function(){
    alert ("Hola...")
  }
</script>
```

También podemos usar la sintaxis **fatArrow**

```
miBoton2.onclick=()=>alert ("Hola...")
```

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. Si queremos ejecutar dos acciones diferentes con funciones diferentes no podemos asignar los dos controladores al mismo elemento, ya que el segundo reemplazara al primero. **Nos gustaría asignar dos controladores de eventos para eso. Pero una nueva propiedad DOM sobrescribirá la existente**

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

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**).

```
let miBoton3=document.getElementById("miBoton3")
miBoton3.addEventListener( "click" , () => alert('Saludar...'));
```

Esta forma no es la más adecuada si queremos poder eliminar más adelante el controlador.

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**:

```
let miBoton3=document.getElementById("miBoton3")
miBoton3.addEventListener( "click" , () => alert('Saludar...'));
miBoton3.removeEventListener( "click", () => alert('Saludar...'));
```

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.

La forma correcta sería utilizar controladores con un nombre asignado, es decir, llamar a funciones.

```
function handler() {
    alert( 'Hola...' );
  }
let miBoton3=document.getElementById("miBoton3")
```

```
miBoton3.addEventListener("click", handler);
miBoton3.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.

Como hemos dicho antes, la gran ventaja de usar los listeners es que podemos asignar varios controladores al mismo elemento y todos se ejecutaran.

```
function handler1() {
    alert( 'Hola primera vez...' );
}

function handler2() {
    alert( 'Hola segunda vez...' );
}

let miBoton4=document.getElementById("miBoton4")

miBoton4.addEventListener("click", handler1);
miBoton4.addEventListener("click", handler2);
```

Como conclusión:

- Establecer controladores de eventos mediante el atributo no es la forma más recomendada.
- Las más correctas serían mediante el DOM y el uso de listeners.

Uso de DOM y listeners

Podemos establecer los controladores utilizando una propiedad DOM y addEventListener. El uso de **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 "click" 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 click 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 click del elemento <p>.

Con el método **addEventListener ()** podemos especificar el tipo de propagación utilizando el parámetro **"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; padding: 20px;}
</style>

<body>
  <h2>Bubling and Capturing</h2>

  <div id="myDiv1">
    <h2>Bubbling:</h2>
    <p id="myP1">Haz click !!! </p>
  </div><br>

  <div id="myDiv2">
    <h2>Capturing:</h2>
    <p id="myP2">Haz click !!! </p>
  </div>
```

```
</body>

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


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

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

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 "click" 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.

Vamos a ver un ejemplo de cómo obtener coordenadas de puntero del objeto de evento:

```
<button id="miBoton" class="btn btn-primary">Haz click...</button>

<script>
  let miBoton=document.getElementById("miBoton")
  miBoton.onclick = function(event) {
    alert("Evento " + event.type + " en " + event.currentTarget);
    alert("Coordenadas: " + 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.

Para el ejemplo anterior:


```
▼ PointerEvent i
    isTrusted: true
    altKey: false
    altitudeAngle: 1.5707963267948966
    azimuthAngle: 0
    bubbles: true
    button: 0
    buttons: 0
    cancelBubble: false
    cancelable: true
    clientX: 54
    clientY: 25
    composed: true
    ctrlKey: false
    currentTarget: null
    defaultPrevented: false
    detail: 1
    eventPhase: 0
    fromElement: null
    height: 1
    isPrimary: false
    layerX: 54
    layerY: 25
    metaKey: false
    movementX: 0
    movementY: 0
    offsetX: 54
    offsetY: 25
    pageX: 54
    pageY: 25
    pointerId: 1
    pointerType: "mouse"
```


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.

```
function handler(){
    alert("Evento " + event.type + " en " + event.currentTarget);
    alert("Coordenadas: " + event.clientX + ":" + event.clientY);
  }
  let miBoton2=document.getElementById("miBoton2")
  miBoton2.addEventListener("click", handler)
```

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

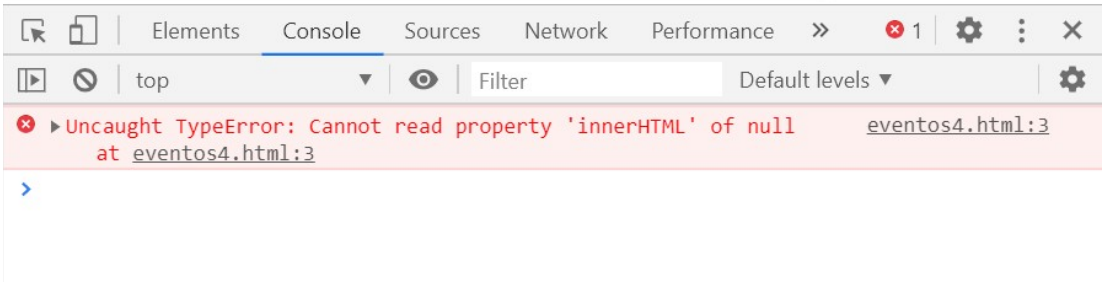
Para poder acceder a los elementos del DOM es necesario que se construya el árbol, para ello se debe de cargar la página web, en ese momento estarán disponibles todos los elementos para poder acceder a ellos mediante código.

Para que esto suceda la forma más sencilla es 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.

```
<html>
  <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.

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

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

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.

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

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>
```

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>
```

6.- Eventos personalizados

A diferencia de los eventos nativos (como click, mouseover, etc.), **los eventos personalizados son creados por el desarrollador** para representar acciones o estados específicos dentro de una aplicación. Esto nos brinda una gran flexibilidad para comunicar entre diferentes partes de nuestro código y crear flujos de trabajo más personalizados.

Usar eventos personalizados nos permite

- **Comunicación entre componentes:** Permiten que diferentes partes de una aplicación se comuniquen de manera desacoplada.
- **Creación de flujos de trabajo personalizados:** Se pueden crear eventos específicos para representar acciones o estados que no están cubiertos por los eventos nativos.
- **Simplificación del código:** Al encapsular la lógica en eventos personalizados, el código se vuelve más legible y mantenible.

Creando eventos personalizados

Para crear un evento personalizado, **utilizamos el constructor CustomEvent**. Este constructor acepta dos argumentos principales:

- **Nombre del evento:** Una cadena que identifica al evento.
- **Opciones:** Un objeto que puede contener propiedades como bubbles (indica si el evento debe burbujear), cancelable (indica si el evento se puede cancelar) y detail (datos adicionales que queremos asociar al evento).

```
<script>

// Crear un evento personalizado llamado 'myCustomEvent' con datos
adicionales
const myEvent = new CustomEvent('myCustomEvent', {
  detail: {
    message: 'Hola desde el evento personalizado',
    data: 42
  }
```

```
});

</script>
```

Lanzando eventos personalizados

Una vez creado el evento, podemos lanzarlo utilizando el método **dispatchEvent()** de un elemento, nosotros utilizaremos window que siempre esta disponible

```
window.dispatchEvent(myEvent)
```

Escuchando eventos personalizados

Para escuchar un evento personalizado, utilizamos el método **addEventListener()** de la misma manera que con los eventos nativos

```
window.addEventListener('myCustomEvent', () => {
 console.log(event.detail.message); // Acceder a los datos adicionales
});
```

Vamos a ver el ejemplo completo:

- Creamos el evento personalizado.
- Añadimos un boton que al hacer click sobre él nos lance nuestro evento personalizado
- Creamos un listener para que cuando se produzca nuestro evento personalizado se ejecute nuestro controlador

```
<body>
    <button id="miBoton">Lanzar evento</button>
</body>

<script>
  const myEvent = new CustomEvent('myCustomEvent', {
      detail: {message: 'Hola desde el evento personalizado',
               data: 42}
  });

let miBoton=document.getElementById("miBoton")
miBoton.addEventListener("click", ()=>window.dispatchEvent(myEvent))

window.addEventListener('myCustomEvent', () => {
  console.log(event.detail.message); // Acceder a los datos adicionales
});

</script>
```

Usos avanzados de eventos personalizados

- **Comunicación entre componentes en frameworks como React, Angular y Vue:** Los eventos personalizados son una herramienta fundamental para la comunicación entre componentes.
- **Creación de sistemas de eventos personalizados:** Podemo crear sistemas de eventos más complejos con múltiples tipos de eventos y canales de comunicación.
- **Simulación de eventos:** Los eventos personalizados pueden utilizarse para simular eventos del usuario y probar la funcionalidad de tu aplicación.