EVENTOS

Contenido

¿Qué es un evento?

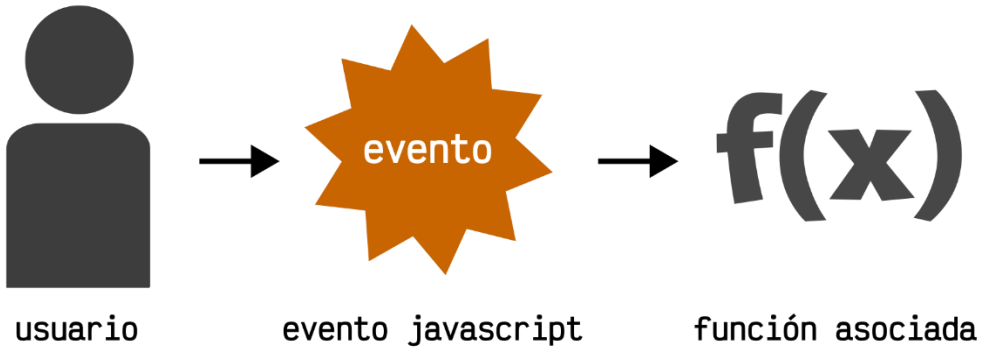
En Javascript existe un concepto llamado **evento**, que no es más que una notificación de que alguna **característica interesante** acaba de ocurrir, generalmente relacionada con el **usuario** que navega por la página.

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»

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.



Uno de los eventos más comunes, es el evento **click**, que es el que se produce **cuando el usuario hace clic** con el ratón en un elemento de la página. Vamos a utilizar este evento a modo de ejemplo en las siguientes secciones de la página, pero recuerda que hay muchos tipos de eventos diferentes.

Formas de manejar eventos

Existen varias formas diferentes de manejar eventos en Javascript. Vamos a ver cada una de ellas, con sus particularidades, pero antes hagamos un pequeño resumen:

Forma	Ejemplo	Artículo en profundidad
Mediante **atributos HTML**	`<button onClick="..."></button>`	Eventos JS desde atributos HTML
Mediante **propiedades Javascript**	`.onclick = function() { ... }`	Eventos JS desde propiedades Javascript
Mediante `addEventListener()`	`.addEventListener("click", ...)`	Eventos JS desde listeners

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. En los siguientes apartados veremos detalladamente sus características, pero por norma general, lo aconsejable es utilizar la última, los **listeners**, ya que son las más potentes y flexibles.

¿Qué es un evento Javascript? Eventos JavaScript desde HTML

Un **evento Javascript** es una característica especial que ha sucedido en nuestra página y a la cual le asociamos una funcionalidad, de modo que se ejecute cada vez que suceda dicho evento. Por ejemplo, el evento **click** se dispara cuando el usuario hace clic en un elemento de nuestra página.

Imaginemos el siguiente código HTML:

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

En nuestro navegador nos aparecerá un botón con el texto «**Saludar**». Sin embargo, si lo pulsamos, no realizará ninguna acción ni tendrá funcionamiento. Para solucionar esto, podemos asociarle un evento:

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

En este ejemplo, cuando el usuario haga clic con el ratón en el botón **Saludar**, se disparará el evento **click** en ese elemento HTML (**<button>**). Dicho botón, al tener un atributo **onClick** (*cuando hagas click*), ejecutará el código que tenemos asociado en el valor del atributo HTML (*en este caso un **alert()***), que no es más que un mensaje emergente con el texto indicado.

Ten en cuenta que el nombre del evento es **click**, sin embargo, en los atributos HTML se coloca siempre precedido de **on**. Las minúsculas/mayúsculas dan igual, aunque lo más habitual es utilizar camelCase.

Organizando la funcionalidad

El valor del atributo **onClick** llevará la funcionalidad en cuestión que queremos ejecutar cuando se produzca el evento. En nuestro ejemplo anterior, hemos colocado un **alert()**, pero lo habitual es que necesitemos ejecutar un fragmento de código más extenso, por lo que lo ideal sería meter todo ese código en una función, y en lugar del **alert()**, ejecutar dicha función:

```
<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**.

Por esta razón, suele ser buena práctica no incluir llamadas a funciones Javascript en nuestro código **.html**, sino que es mejor hacerlo desde el fichero externo **.js**, localizando los elementos del DOM con un **.querySelector()** o similar.

En resumen:

- Gestionar **eventos Javascript** desde HTML es muy sencillo.
- Hay que tener en cuenta que «mezclamos» código Javascript dentro de HTML.
- Para que el código sea más legible y fácil de mantener, se recomienda gestionar eventos desde Javascript (DOM) o, mejor aún, gestionar eventos mediante addEventListener().

Eventos a través del DOM

Existe una forma de **gestionar eventos Javascript** sin necesidad de hacerlo desde nuestros ficheros **.html**. No obstante, se trata de una «trampa», puesto que seguimos haciéndolo desde HTML, sólo que ese HTML se crea desde Javascript, y nos permite llevarlo a los ficheros **.js**.

Utilizando propiedad Javascript

La idea es la misma que vimos en el artículo anterior, sólo que en esta ocasión haremos uso de una propiedad Javascript, a la que le asignaremos la función con el código asociado.

Vamos a realizar el mismo ejemplo anterior para verlo claramente:

```
<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()**. Esto podríamos hacerlo mediante una clase o un id, pero en este ejemplo lo hemos hecho directamente mediante el botón, para simplificar.

En la linea siguiente, observa que asignamos una función con el código deseado (*el código que queremos ejecutar cuando ocurre el evento*) en la propiedad **.onclick** del elemento **<button>**. Esta es una propiedad especial que pasaremos a explicar a continuación.

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.

Utilizando setAttribute()

Realmente lo que estamos haciendo es equivalente a añadir un atributo **onclick** en nuestro **<button>**, solo que lo hacemos a través de la API de Javascript. Otra forma similar, donde si se verá más claro, sería la siguiente:

```
<button>Saludar</button>

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

Observa que en este caso, si vemos la similitud con la forma del artículo anterior, ya que estamos utilizando el método **.setAttribute()**, donde añadimos el atributo **onclick** con el valor indicado a continuación.

En resumen:

- A grandes rasgos, se trata de una forma alternativa a gestionar los eventos Javascript desde HTML, pero creando el HTML mediante la API del DOM de Javascript.
- En el caso de que necesitemos añadir más de una función al evento, la cosa se puede complicar. Podríamos tener una función que ejecute varias funciones, pero sin duda alguna, utilizar el método **.addEventListener()** será mucho más cómodo, sencillo y legible.

El método addEventListener

En los artículos anteriores hemos visto qué son los **eventos Javascript** y cómo gestionarlos a través de código HTML, o a través de código Javascript, utilizando la API del DOM. Sin embargo, la forma más recomendable es hacer uso del método **.addEventListener()**, el cuál es mucho más potente y versátil para la mayoría de los casos.

- Con **.addEventListener()** se pueden añadir fácilmente **varias** funcionalidades.
- Con **.removeEventListener()** se puede **eliminar** una funcionalidad previamente añadida.
- Con **.addEventListener()** se pueden indicar ciertos **comportamientos** especiales.

Método .addEventListener()

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*). De forma opcional, se le puede pasar un tercer parámetro con ciertas opciones, que veremos más adelante:

Método	Descripción
.addEventListener(STRING event, FUNCTION func)	Escucha el evento **event**, y si ocurre, ejecuta **func**.
.addEventListener(STRING event,FUNCTION func, OBJECT options)	Idem, pasándole ciertas opciones.

Para verlo en acción, vamos a crear a continuación, el mismo ejemplo de apartados anteriores, de esta forma veremos cómo funciona y podremos comparar con los anteriores:

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

Observa algunas cosas de este ejemplo:

- En el **primer parámetro** indicamos el nombre del evento, en nuestro ejemplo, **click**. Con **.addEventListener()** no se precede con **on** los nombres de eventos y se escriben en minúsculas, sin camelCase.
- En el **segundo parámetro** indicamos la función con el código que queremos que se ejecute cuando ocurra el evento.

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.

Múltiples listeners

Dicho método **.addEventListener()** 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>
```

Observa que en este ejemplo, hemos añadido una clase **.red** de CSS, que coloca el color de fondo del botón en rojo. Además, hemos creado dos funcionalidades:

- **action**, que muestra un mensaje de saludo
- **toggle**, que añade o quita el color rojo del botón

Observa que al pulsar el botón se efectúan ambas acciones, ya que hay dos listeners en escucha.

Opciones de addEventListener

Al utilizar el método **.addEventListener()**, se puede indicar un tercer parámetro opcional. Se trata de un OBJECT opcional en el cual podemos indicar alguna de las siguientes opciones para modificar alguna característica del listener en cuestión que vamos a crear:

Opción	Descripción
BOOLEAN **capture**	El evento se dispara al inicio (*capture*), en lugar de al final (*bubble*).
BOOLEAN **once**	Sólo ejecuta la función la primera vez. Luego, elimina listener.
BOOLEAN **passive**	La función nunca llama a **.preventDefault()** (mejora rendimiento).

Repasemos cada una de estas opciones:

- En primer lugar, la opción **capture** nos permite modificar la **modalidad** en la que escuchará el evento (*capture/bubbles, ver más adelante*). Esto, básicamente, lo que hace es modificar en qué momento se procesa el evento.
- En segundo lugar, la opción **once** nos permite indicar que el evento se procesará **solo la primera vez** que se dispare un evento. Internamente, lo que hace es ejecutarse una primera vez y luego llamar al **.removeEventListener()**, eliminando el listener una vez ha sido ejecutado.
- En tercer y último lugar, la opción **passive** nos permite crear un **evento pasivo** en el que indicamos que nunca llamaremos al método **.preventDefault() (nos permite cancelar un evento si este es cancelable)** para alterar el funcionamiento del evento. Esto puede ser muy interesante en temas de **rendimiento** (*por ejemplo, al hacer scroll en una página*), ya que los eventos pasivos son mucho menos costosos.

Método .removeEventListener()

El ejemplo anterior, se puede completar haciendo uso del método **.removeEventListener()**, que sirve como su propio nombre indica para eliminar un listener que se ha añadido previamente al elemento. Para ello es muy importante indicar **la misma función** que añadimos con el **.addEventListener()** y no una **función diferente que haga lo mismo que la primera**.

Método	Descripción
.removeEventListener(STRING event, FUNCTION func)	Elimina la funcionalidad **func** asociada al evento **event**.

Veamos el ejemplo anterior, eliminando la funcionalidad **action** mediante **.removeEventListener()**, es decir, sólo debería actuar la funcionalidad **toggle**:

```
<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);       // Add listener
button.addEventListener("click", toggle);       // Toggle red CSS
button.removeEventListener("click", action);    // Delete listener
</script>
```

Ten en cuenta que es posible eliminar el listener del evento porque hemos guardado en una constante la función, y tanto en **.addEventListener()** como en **.removeEventListener()** estamos haciendo referencia a la misma función. Si en lugar de esto, añadiésemos la función literalmente, aunque hagan lo mismo, serían funciones diferentes y no realizaría lo que esperamos.

PROPAGACIÓN DE EVENTOS

Una cuestión fundamental en la captura de eventos es cómo se propagan los eventos sobre los contenedores de los elementos. Es decir, supongamos que tenemos una capa de tipo **div**, en ella un **párrafo** y dentro del párrafo un **botón**. Hacemos clic sobre el botón. ¿El párrafo podría capturar el clic? ¿Y la capa?

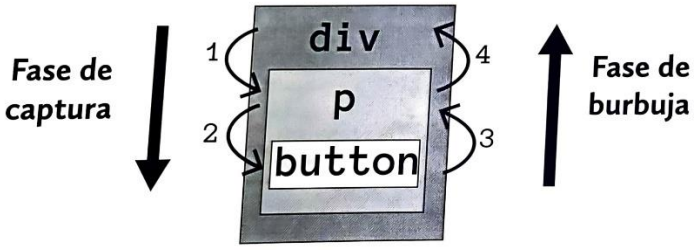
```
<body>


    <div class="container ">
        <div class="row bg-success align-items-center" style="height: 500px;">
            <p class="text-center">
                <button class="btn btn-primary"  id="boton_2" >Pulsar</button>
            </p>
        </div>
    </div>
    <script>
        let capa=document.querySelector("div");
        let p=document.querySelector("p");
        let boton=document.querySelector("button");
        capa.addEventListener("click",()=>console.log("click en capa"));
        p.addEventListener("click",()=>console.log("click en p"));
        boton.addEventListener("click",()=>console.log("click en capa"));


    </script>
</body>
```

La situación comentada es la que presenta el código anterior. Hemos capturado los eventos de tipo clic en los tres elementos (div,p y button). Si ejecutamos el código, hacemos clic en el botón, por consola veremos:

El evento se propaga desde el botón (elemento más interior) hasta el elemento de tipo div (elemento más exterior, más cercano a la raíz del DOM). El proceso de propagación de eventos se describe en la imagen:



Por defecto, la función asociada al evento se lanza en la fase de burbuja, por eso en el código anterior podemos observar primero el mensaje del botón, luego el del párrafo y luego el de la capa.

Podemos modificar este comportamiento si indicamos que el lanzamiento sea en la fase de captura. Se consigue este efecto gracias a que, en realidad, el método **addEventListener** tiene un tercer parámetro relacionado con el tipo de captura. Es un valor booleano con estas posibilidades

- **true.** El Código se lanza en la fase de captura.
- **False.** Valor por defecto, el código se lanza en la fase de burbuja.

Por lo tanto, si cambiamos las tres últimas líneas del código anterior:

```
        capa.addEventListener("click",()=>console.log("click en
capa"),true);
        p.addEventListener("click",()=>console.log("click en p"),true);
        boton.addEventListener("click",()=>console.log("click en
capa"),true);
```

Obtendremos el siguiente resultado:

```
click en capa                    propagacion2.html:22
click en p                       propagacion2.html:23
click en capa                    propagacion2.html:24
```

ANULAR EVENTOS

Imaginemos que deseamos que cuando hagamos un clic a un elemento concreto se nos muestre un mensaje, pero queremos que ese mansaje aparezca una sola vez. Po defecto, el evento se captura indefinidamente, pero existe un método contrario a **addEventListener** que tiene los mismos parámetros. Se trata de **removeEventListener.** Visto anteriormente:

Solo podemos retirar funciones que tengan nombre, no podemos anular funciones anónimas asociadas a eventos.

OBJETO DE EVENTO

UTILIDAD Y USO DEL OBJETO DE EVENTO

Cuando se produce un evento, el navegador **crea automáticamente un objeto** cuyas propiedades pueden ser muy útiles a los desarrolladores. La información fundamental que

graban son las coordenadas del cursor del ratón, si hay teclas pulsadas, el elemento que ha producido el evento, etc.

Ya hemos visto que, cuando se produce un evento, se invoca automáticamente el código de una función. Pues bien, esa función puede tener un parámetro que será una referencia al objeto del evento. Gracias a ese parámetro podremos leer la información del evento.

Una de las propiedades que posee este objeto se llama **target** y es una referencia al elemento que causo el evento.

Veamos un ejemplo:

```
<div class="container ">
    <div class="row bg-success align-items-center text-white h1">
            <p>Uno</p>
            <p>dos</p>
            <p>tres</p>

    </div>
</div>
<script>
    function escribeContenido(evento){
        alert(evento.target.textContent);
    }

    let parrafos=document.querySelectorAll("p");
    for(let parrafo of parrafos){
        parrafo.addEventListener("click",escribeContenido);
    }
```

OBTENER COORDENADAS DEL EVENTO

Los objetos del evento poseen dos propiedades para obtener las coordenadas del ratón en el momento del evento: **clientX** y **clientY.** La primera obtiene la posición horizontal en pixeles y la segunda la vertical. Ambas lo hacen utilizando como referencia la esquina superior izquierda de la ventana del navegador, que será el punto (0,0) de coordenadas.

Hay dos propiedades similares **screenX** y **screenY**. La diferencia es que el origen de coordenadas no es la esquina del navegador, sino de la pantalla. Coinciden muchas veces, pero si la ventana no está maximizada, no coincidirán.

Realiza la tarea: 8.1 Ratón persiguiendo al ratón

Otras coordenadas que se almacenan en el objeto evento son **pageX** y **pageY.** Funcionan como **clientX** y **clientY**, pero no solo toman las coordenadas de la ventana, tienen en cuenta el

desplazamiento realizado en el elemento. Es decir, si hemos avanzado dos pantallas de 500 px de alto usando la barra de desplazamiento y el cursor está mitad de pantalla. **ClientY** nos daría 250, mientas que **pageY** 1250.

OBTENER LA TECLA PULSADA

En eventos de teclado, el objeto de evento posee información sobre la tecla pulsada. Son importantes estos datos en aplicaciones de juegos o aplicaciones de control de la tecla pulsada.

La propiedad más importante es **key** que obtiene el texto que indica la tecla pulsada Así sus valores fundamentales devueltos son:

- Si es una **tecla de carácter**, nos retorna el carácter. Por ejemplo “a”, “g”, ”h”, “A”, “G”, etc.
- Con las **teclas numéricas** actúa igual y con el **shift** pulsado nos devuelve el segundo símbolo de la letra y con **AltGr** pulsadas el tercer símbolo.
- En caso de pulsar, sin más, **teclas de control** nos devolverá su nombre: **“Control”,”Alt”,”Shift”,”AltGr”,”F5”,”KeyUp”,”KeyDown”,”End”, etc.**

A veces es necesario saber si se pulsó a la vez que la tecla, alguna de las teclas de control especial: **Ctrl, Alt o Shift.** Tenemos tres propiedades relacionadas con estas teclas que devolverán true si la tecla en cuestión se pulsó, son: **AltKey, CtrlKey y MetaKey** (tecla de los ordenadores MAC)

Realiza la tarea: 8.2 Tecla que pone imagen de fondo

Otra propiedad interesante es **location**, que permite saber la localización de la tecla en el teclado. El caso habitual es tener que distinguir una tecla **Shift** de la otra. Los valores que puede devolver location son:

VALOR	CONSTANTE RELACIONADA	SIGNIFICADO
0	DOM_KEY_LOCATION_STANDARD	Teclado normal
1	DOM_KEY_LOCATION_LEFT	Zona izquierda
2	DOM_KEY_LOCATION_RIGHT	Zona derecha
3	DOM_KEY_LOCATION_NUMPAD	Teclado numérico

OBTENER LOS BOTONES DEL RATÓN

En los eventos del ratón hay propiedades comunes con las teclas. Por ejemplo, también tenemos las propiedades **AltKey, CtrlKey, ShiftKey y MetaKey,** para saber si la pulsación de alguna de estas teclas acompaña al evento de ratón.

Hay otras propiedades de coordenadas muy interesantes (además de las ya vistas). Así **movementX y movementY** nos retornan la diferencia en píxeles de las coordenadas X e Y respecto al último movimiento del ratón.

Además, la propiedad **button** devuelve el botón de ratón pulsado en el momento del evento. Los valore posibles para la propidad **button** son:

VALOR	SIGNIFICADO
0	Botón principal
1	Botón central (en muchos casos la rueda)
2	Botón secundario
3	Cuarto botón. Retroceder página
4	Quinto botón. Avanzar página

ANULAR COMPORTAMIENTOS PREDETERMINADOS

Además de propiedades, los objetos de evento poseen métodos. Uno de los más importantes es **preventDefault.** No tiene parámetros y lo que hace es conseguir que si ese evento producía en el elemento un comportamiento concreto por defecto, que ese comportamiento no se produzca.

Es más fácil de entender con un ejemplo. Supongamos que hemos puesto un enlace y deseamos que el usuario confirme que desea ir al destino de ese enlace, de modo que si no se confirma nos quedamos en la página actual.

```
        let enlace=document.querySelector("a");
        enlace.addEventListener("click",function(ev){
            if(!confirm("¿Continuar al enlace?"))
                ev.preventDefault();
        });
```

CANCELAR PROPAGACIÓN

Anteriormente hemos visto la propagación de eventos. Podemos cancelar la propagación de eventos con un método llamado **stopPropagation.** Veamos un ejemplo:

```
    <div class="container ">
        <div class="row  align-items-center" style="height: 500px;">
            <p class="text-center h1">
                Pinta de rojo
            </p>
            <button>Pinta de verde</button>
        </div>
    </div>
```

```
<script>
    let boton=document.querySelector("button");
    let div=document.querySelector(".row");
    boton.addEventListener("click",(ev)=>{
        document.body.style.background="green";
    });
    div.addEventListener("click",(ev)=>{
        document.body.style.background="red";
    });
</script>
```

El botón está dentro de un elemento de tipo div. Hacer clic en el elemento **div** (con el texto pinta rojo) provoca colorear todo el cuerpo de la página en rojo. Hacer clic en el botón lo pinta de verde. Sin embargo, al hacer clic en el botón, la página también se pinta de rojo. Si pudiésemos capturarlo a cámara lenta veriamos primero el color verde y luego el rojo, predomina el color rojo porque el evento clic se propaga al elemento div, después de haber sido capturado por el botón.

Para evitar este efecto disponemos del método **stopPropagation;**

```
    boton.addEventListener("click",(ev)=>{
        document.body.style.background="green";
        ev.stopPropagation();
    });
```

El funcionamiento también podíamos haberlo modificado si la ejecución del código la hubiéramos obligado a realizar en la fase de captura.

LANZAR EVENTOS

JavaScript permite lanzar eventos a voluntad. La idea se basa en crear objetos de evento propios y mediante el método **dispatchEvent** de los elementos, enviar el evento creado.

La creación del evento se basa en el objeto **Event** (o en sus descendientes más específicos como el **mouseEvent** por ejemplo) el cual funciona indicando el tipo de evento en la creación. Por ejemplo:

```
<div class="container ">
    <div class="row  align-items-center" style="height: 500px;">

        <button class="button">Pinto de verde</button>
        <button>Hago lo mismo</button>
    </div>
```

```
</div>
<script>
    let boton1=document.querySelectorAll("button")[0];
    let boton2=document.querySelectorAll("button")[1];
    boton1.addEventListener("click",(ev)=>{
        document.body.style.background="green";
    });
    boton2.addEventListener("click",()=>{
        let e1=new Event("click");
        boton1.dispatchEvent(e1);
    });
</script>
```

Ambos realizan la misma acción porque el evento **onclick** del segundo botón realiza un click sobre el primero.

EVENTOS MÁS COMUNES

Eventos del mouse:

- **`click`** – cuando el mouse hace click sobre un elemento (los dispositivos touch lo generan con un toque).
- **`dblclick`**-doble click
- **`contextmenu`** – cuando el mouse hace click derecho sobre un elemento.
- **`mouseover`** / **`mouseout`** – cuando el cursor del mouse ingresa/abandona un elemento.
- **`mousedown`** / **`mouseup`** – cuando el botón del mouse es presionado/soltado sobre un elemento.
- **`mousemove`** – cuando el mouse se mueve.
- **`Mouseenter/mouseleave`**

Realiza la tarea 3: Capa rollup

Eventos del teclado:

- **`keydown`** / **`keyup`** – cuando se presiona/suelta una tecla.
- **`Keypress`** – cuando el usuario pulsa y suelta una tecla.

Realiza la tarea 4: Control de velocidad

Eventos de movimiento en la ventana:

scroll – Cuando se ha desplazado la ventana a través de las barras de desplazamiento o usando el dispositivo táctil.

resize – Se produce al cambiar el tamaño de la ventana

Realiza la tarea 5: Cierre de panel con scroll

Eventos del documento:

- **domcontentloaded** --cuando el HTML es cargado y procesado, el DOM está completamente construido. A diferencia de load, se dispara sin esperar a que se terminen de cargar hojas de estilos, imágenes y elementos de segundo plano.
- **load** – Se concluyó la carga del elemento. Es uno de los elementos más importantes a capturar para asegurar qué el código siguiente funciona con la seguridad de que está cargado todo lo que necesitamos. Cuando se aplica al elemento **window,** se produce cuando todos los elementos del documento se han cargado.
- **abort**- Se produce cuando se anula la carga de un elemento.
- **error**- Sucede si hubo un error en la carga.
- **progress**- Se produce si la carga está en proceso.
- **readystatechange** – Ocurre cuando se ha modificado el estado del atributo readystate, lo cual ocurre cuando se ha modificado el estado de carga y descarga.

- **Realiza la tarea 6:** Panel de carga

OTROS EVENTOS

- **Eventos sobre el historial (popstate)**
- **Eventos de arrastre (dragstart,drag, dragstrop)**
- **Eventos relacionados con la reproducción de medios(waiting, playing,pause,play….)**
- **Eventos CSS sobre animaciones y transiciones (animationstart,animationend,transitionrun,transitionstart,transitionend…)**
- **Eventos del portapapeles(cut,copy,paste)**
- **Eventos especiales(offline,online,fullscreenchange…..)**

FORMULARIOS

Indudablemente, los formularios son el componente fundamental de captura de eventos en una aplicación web. Nos han sido muy útiles métodos **alert,prompt** y **confirm**, pero la realidad es que las aplicaciones reales casi no los utilizan. Si hay que mostrar mensajes se hacen en elementos HTML normales (paneles, capas, etc) y la introducción de datos por parte del usuario se hace con formularios.

Los formularios son la base de la comunicación de las aplicaciones web con el usuario. El objeto **document** dispone de una propiedad llamada **forms** que retorna una colección con todos los formularios del documento. Así document.forms[0] hará referencia al primer formulario, pero es posible e incluso más recomendable acceder al formulario mediante otros métodos, por ejemplo su identificador.

De forma clásica se accede a los controles de los formularios mediante su atributo **name.** Este atributo es el que permite dar nombre a la información que se graba con el control y que se enviará cuando se pulse el botón de envío (**submit**) del formulario. Ejemplos

```
document.forms[0].apellido1;
document.forms[0].[“apellido1”];
```

Y por supuesto:

```
document.querySelector(“[‘name=apellido1’]”);
document.querySelector(“#id”);
```

Si hay varios elementos con el mismo nombre (como ocurre con los elementos de radio), esta última forma de acceder debe utilizar querySelectorAll, la cual devolverá una colección que tendremos que ir recorriendo. Repasar conceptos:

- Métodos de envío **GET/POST**
- Atributos de los controles **value**, **name** e **id**
- Botones **submit** y **reset**.

Propiedades y métodos del objeto **form**

PROPIEDAD O MÉTODO	USO
Elements	Colección con todos los controles del formulario
Length	Número de controles del formulario
Action	Devuelve el contenido de action, que marca la URL destino de los datos del formulario. Permite su modificación.
Method	Devuelve la forma de envío de los datos (get o post). Permite su modificación
Enctype	Obtiene la forma de codificar los datos del formulario
acceptCharset	Obtiene o modifica el conjunto de caracteres del formulario
Submit()	Envía los datos del formulario a su destino
Reset()	Deja los valores de los controles del formulario a su estado por defecto

Esos métodos nos permiten automatizar acciones de forma muy efectiva en los formularios.

EVENTOS DE FORMULARIO (ELEMENTO FORM):

- **submit –** cuando el visitante envía un `<form>`.
- **focus –** cuando el visitante se centra sobre un elemento, por ejemplo un `<input>`.
- **reset –** cuando se resetea el formulario
- **change-** Se produce cuando cambiamos el valor de cualquier control del formulario, ejemplos son:
    - Cambiar el estado de un **botón de radio** o de tipo **check.**
    - Modificar el valor de un **cuadro de texto** o **contraseña**
    - Elegir otro valor de una **lista de opciones**.
    - Arrastrar un **deslizador** para modificar su valor.

  Realmente, el evento se produce cuando se ha modificado el valor y, además se ha perdido el foco sobre ese control. Es decir, durante el cambio no se lanza el evento.

PROPIEDADES DE LOS CONTROLES

Para modificar las propiedades de los controles de formulario podemos trabajar sobre sus atributos como hacemos con cualquier otro elemento. Por ejemplo, para obtener el valor de un control de formulario:

```
control.getAttribute(“value”);
```

Suponiendo que control es el nombre del formulario obtendremos su valor, pero el valor es algo que cambia de forma dinámica en los formularios y **getAttribute** no refleja esos cambios. Por ello es mejor usar propiedades que reflejen los cambios que hace el usuario. Estas propiedades son:

PROPIEDAD	USO	CONTROLES QUE LA USAN
name	Nombre del control.	Prácticamente todos.
type	Valor de atributo **type**.	Controles de tipo **input**.
value	Devuelve el valor actual del control.	Prácticamente todos.
checked	Puede valer true o false. Indica, sobre un control de activar o desactivar, si el control está activado o no.	Botones de radio y botones de tipo **check**.
defaultChecked	Indica el estado inicial de la propiedad **checked** en el control.	Botones de **radio** y botones de tipo **check**.
disabled	Indica, con **true** o **false**, si el control está deshabilitado o no.	Prácticamente todos.
readonly	Indica, con **true** o **false**, si el control es de solo lectura o no.	Prácticamente todos.
required	Indica con **true** o **false** si es obligatorio o no cambiar el valor del control.	Controles de entrada de texto o listas.
maxLength	Permite ver y modificar la propiedad que define la anchura máxima de texto.	Controles de entrada de texto.
min	Valor mínimo posible para el control.	Input de tipo numérico o de fecha.
max	Valor máximo posible para el control.	Input de tipo numérico o de fecha.
step	Mínimo valor de cambio del control. Si el valor es 1, los valores avanzan de uno en uno como poco.	Input de tipo numérico o de fecha.
selectionStart	En controles de entrada de texto indica el índice dentro del texto general en el que comienza la selección actual sobre el texto. Si no hay nada seleccionado, indica la posición del cursor en el texto.  El cuadro de texto con selección que se muestra en la imagen anterior tendría estos valores en las propiedades: ■ **seleccionStart**: 6 ■ **seleccionEnd**: 14	Controles de entrada de texto.

PROPIEDAD	USO	CONTROLES QUE LA USAN
selectionEnd	En controles de entrada de texto indica el índice dentro del texto general en el que finaliza la selección actual sobre el texto. Si no hay nada seleccionado, indica la posición del cursor en el texto.	Controles de entrada de texto

MÉTODOS DE LOS CONTROLES

MÉTODO	USO	CONTROLES
focus()	Fuerza a que el control obtenga el foco.	Prácticamente todos.
blur()	Provoca la pérdida de foco en el control.	Prácticamente todos.
select()	Selecciona todo el texto en el control y también deja el foco en él.	Controles de entrada de texto.
setSelectionRange(inicio,fin)	Selecciona el texto que va desde la posición inicio hasta la posición fin (sin incluir esta última).	Controles de entrada de texto.

LISTA EVENTOS:

https://www.w3schools.com/js/js_events_examples.asp