

# Desarrollo Web en Entorno Cliente

## Tema 10





# [eventos]

JavaScript fue diseñado para añadir dinamismo a las páginas web y gran parte del mismo se consigue a través de los eventos.

En la programación tradicional, las aplicaciones se ejecutan secuencialmente, de principio a fin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es la programación basada en eventos. Los scripts y programas esperan, sin realizar ninguna tarea, hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JS permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos.

No obstante, los eventos son la manera que existe en JS para controlar las acciones de los usuarios y poder definir un comportamiento de la página cuando se produzcan.

Cuando un usuario hace algo en la página web se produce un evento.

# [eventos]

Algunos eventos que no están relacionados directamente con acciones de usuario, como por ejemplo el evento *load* de un documento, que se produce automáticamente cuando un documento ha sido cargado.

Para entender las diferentes formas que tiene JS de manejar los eventos hay que remontarse al pasado con el fin de ver su evolución.

En el nivel 1 de DOM no se incluía especificaciones relativas a los eventos JS. El nivel 2 de DOM se incluía ciertos aspectos relacionados con los eventos y el nivel 3 de DOM incluye la especificación completa de los eventos de JS, pero, desafortunadamente, la especificación de nivel 3 de DOM se publicó en el año 2004, más de diez años después de que los primeros navegadores incluyeran los eventos.

Por este motivo, muchas de las propiedades y métodos actuales relacionados con los eventos son incompatibles con los de DOM. Navegadores como Internet Explorer trataban los eventos siguiendo su propio modelo incompatible con el estándar.

# [eventos]

El modelo simple de eventos se introdujo en el estándar HTML4 y se considera parte del nivel más básico de DOM. Aunque sus características son limitadas, es el único modelo que es compatible con todos los navegadores y, por tanto, el único que permite crear aplicaciones que funcionan de la misma manera en todos los navegadores.

Con HTML4, y posteriormente con HTML5, los eventos son capturados a través de gestores o manejadores de eventos definidos como atributos en las etiquetas HTML (modelo de eventos en línea o *inline*).

Así, cuando un usuario hace clic en un botón, se disparaba el evento *onclick*, previamente programado en la etiqueta HTML. Ese evento hace una llamada a una función en la que se realizan ciertas operaciones como respuesta a dicha interacción del usuario sobre ese botón.

```
<button onClick="saludo()"></button>
```

# [eventos]

Posteriormente, surgió el modelo de registro de eventos consistente en que los eventos se asignen como una propiedad del objeto y fuera de la estructura HTML, separando el código HTML del código JS.

```
document.getElementById("boton").onclick = saludo();
```

En este modelo de gestión de eventos se usan las propiedades del elemento para asignarle el manejador del evento, esto es, la función que será ejecutada cuando sobre dicho elemento se produzca el evento.

No es un modelo estándar de registro de eventos, pero si es utilizado ampliamente.

Además, se usa la palabra reservada *this*, para hacer referencia al objeto dónde se programó el evento, esto es, *this* dice *quién* fue el que invocó al manejador del evento. La palabra reservada *this* en JS guarda una referencia al objeto que invocó o llamó al manejador del evento o método en el caso de un objeto.

# [eventos]

Por último, está el modelo de registro avanzado de eventos de W3C.

Este nuevo modelo estándar, propuesto por el W3C, asigna una función al evento mediante el método *addEventListener()* y para eliminarlo se usa *removeEventListener()*.

Es personalizable, lo que permite especificar cuándo se quiere que se dispare el evento: en la carga, en la ejecución de un script,...

Ejemplo:

```
const elemento=document.querySelector("#enlace");  
elemento.addEventListener('click',saludo, false);
```

# [eventos]

Como se ha podido observar, existen tres modelos para implementar eventos en JS.

En resumen, los tres modelos de captura y respuesta de eventos son:

Ejemplo	Descripción	Modelo
<code>&lt;button onClick="..."&gt;&lt;/button&gt;</code>	Mediante un atributo HTML precedido de <i>on</i> que llama a la función	Modelo de eventos mediante HTML (estandarizado por Netscape). Llamado Modelo de registro de eventos en <i>inline</i> o en línea
<code>.onclick = ...</code>	Mediante propiedades del elemento precedidas de <i>on</i> que contienen una función	Modelo de eventos mediante propiedades o Modelo de eventos DHTML
<code>.addEventListener("click", ...)</code>	Mediante la escucha de eventos a través de <i>listeners</i>	Modelo de eventos propuesto y estandarizado de W3C



## modelo de registro de eventos en línea

Probablemente este modelo es la forma más básica y sencilla de manejar eventos, y se realiza a través de un atributo en una etiqueta HTML, como por ejemplo `<button>`.

Se trata de una aproximación simple y sencilla, que para casos muy básicos puede ser más que suficiente, aunque no es recomendable usar este modelo ya que tiene el problema de que se mezcla la estructura de la página web con la programación de la misma.

Cada elemento HTML tiene definida su propia lista de posibles eventos que se le pueden asignar.

El nombre de los eventos se construye mediante el prefijo *on*, seguido del nombre en inglés de la acción asociada al evento. Así, el evento de hacer clic en un elemento con el ratón se denomina *onClick* y el evento asociado a la acción de mover el ratón se denomina *onMousemove*.

```
<a href="pagina.html" onClick="alert('Has pulsado en el enlace')">Pulsa aquí</a>
```

## modelo de registro de eventos en línea

Al evento se le asocia una serie de instrucciones que se ejecutarán cuando se desencadene el evento, este código recibe el nombre de “*manejador del evento*” y suele encapsularse a una función.

```
<a href="pagina.html" onClick="alert('Has pulsado en el enlace')">Pulsa aquí</a>
```

Es buena práctica mantener en un fichero de JS el manejador del evento fuera del código HTML:

```
<a href="pagina.html" onClick="alertar()">Pulsa aquí</a>
```

```
let veces=0;

function alertar(){
    veces+=1;
    alert(`Has pulsado ${veces} veces en el enlace`);
    return false;
}
```

## modelo de registro de eventos en línea

Si un evento genera la ejecución de un script y, además, también se genera la acción por defecto para ese objeto entonces:

- El script se ejecutará primero.
- La acción por defecto se ejecutará después.

```
<a href="pagina.html" onClick="alertar()">Pulsa aquí</a>
```

```
function alertar(){  
    alert("Has pulsado en el enlace");  
}
```

En el ejemplo, al hacer clic en el enlace se mostrará la alerta y a continuación se conectará con la página *pagina.html*. En ese momento desaparecerán de memoria los objetos que estaban en un principio, cuando se originó el evento.

# modelo de eventos mediante HTML

A veces, es necesario bloquear o evitar que se ejecute la acción por defecto.

Cuando se programa un gestor de eventos, ese gestor puede devolver un valor *true* o *false*, usando la instrucción *return true* o *return false*.

Si se devuelve *false* se evita que se ejecute la acción por defecto.

```
<a href="pagina.html" onClick="alertar();return false">Pulsa aquí</a>
```

En el ejemplo, se llama a la función *alertar()* pero se evita que se conecte con la página *pagina.html*, que es la acción por defecto de la etiqueta *<a>*, mediante la devolución de *false* (*return false*).

## modelo de registro de eventos en línea

En el siguiente ejemplo se utiliza como valor de retorno lo que el usuario responda a la ventana de dialogo *confirm*, *true* o *false*.

Este valor será lo que devuelva el script, si es *true* se ejecutará la acción por defecto, saltará a la sección con el *id Profesor*, y si es *false*, no se realizará dicha acción.

```
<a href="#Profesor" onclick="return confirmar()">Profesor</a>
```

```
function confirmar(){  
    return confirm("¿Deseas saltar a la sección Profesor?");  
}
```

## modelo de registro de eventos en línea

En ocasiones, se necesita obtener información más específica del evento, como, por ejemplo, el número de veces que hizo clic el usuario o sobre que elemento concreto se ha producido el evento, o cuando ha sucedido.

Para ello, se dispone de un objeto especial llamado *event* que permite obtener información adicional sobre el evento.

El objeto *event* es enviado como parámetro a la función manejadora del evento si se incluye en su definición.

```
<a href="#Profesor" onClick="((e)=>{alert(`Has hecho ${e.type}`)})(event);">Profesor</a>
```

## modelo de registro de eventos en línea

Cada evento está representado por un objeto que se basa en la interfaz *Event*, y puede tener propiedades y/o funciones personalizadas adicionales para obtener más información acerca de lo sucedido.

Los eventos pueden representar cualquier cosa desde las interacciones básicas del usuario para notificaciones automatizadas de las cosas que suceden en el modelo de representación.

Existen muchos tipos de eventos, algunos son eventos estándar definidos en las especificaciones oficiales, mientras que otros son eventos usados internamente por los navegadores específicos, como, por ejemplo, eventos específicos de Mozilla y que solo se pueden usar para interactuar en este navegador.

Para más información sobre los eventos y tipos de eventos se puede consultar la siguiente página:

<https://developer.mozilla.org/es/docs/Web/Events>

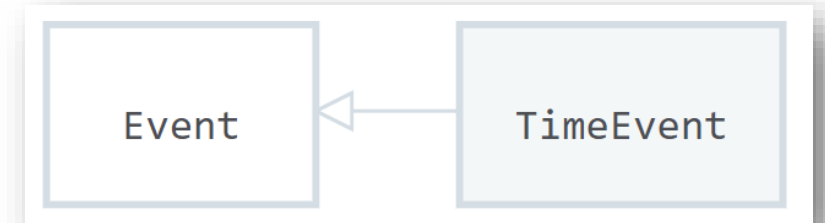
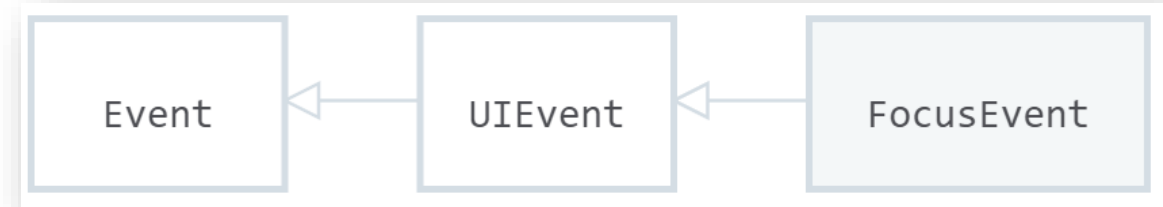
# modelo de registro de eventos en línea

Existen muchos y variados tipos de interfaces de eventos, a la derecha se muestran unos pocos.

Por ejemplo, la interfaz *UIEvent* representa eventos simples del usuario, *InputEvent* representa eventos que indican cambios en un elemento con contenido editable, como un elemento *textarea*.

Todos los objetos de evento en el DOM se basan en el objeto de *Event*.

Por lo tanto, todos los demás objetos de evento (como *UIEvent*, *MouseEvent*, *KeyboardEvent* ) tienen acceso a las propiedades y métodos del objeto de *Event*.





# modelo de registro de eventos en línea

A continuación, se muestran las propiedades y métodos del objeto de *Event*.

Propiedad/Método	Descripción
<i>bubbles</i>	Devuelve si un evento específico es un evento burbujeante o no
<i>cancelBubble</i>	Establece o devuelve si el evento debe propagarse hacia arriba en la jerarquía o no (burbujeante)
<i>cancelable</i>	Devuelve si se puede evitar o no la acción predeterminada de un evento
<i>composed</i>	Devuelve si el evento está compuesto o no
<i>createEvent()</i>	Crea un nuevo evento
<i>composedPath()</i>	Devuelve la ruta del evento
<i>currentTarget</i>	Devuelve el elemento que desencadenó el manejador de evento
<i>defaultPrevented</i>	Devuelve si se llamó o no al método <i>preventDefault()</i> para el evento
<i>eventPhase</i>	Devuelve la fase del flujo de eventos se está evaluando actualmente: 1 – Fase capturing 2 – En el elemento objetivo 3 – Fase bubbling
<i>isTrusted</i>	Devuelve si un evento es de confianza o no
<i>preventDefault()</i>	Cancela la acción por defecto asociado al evento si tiene asociada alguna
<i>stopImmediatePropagation()</i>	Evita que se llamen a otros manejadores del mismo evento
<i>stopPropagation()</i>	Evita una mayor propagación de un evento durante el flujo de eventos

# modelo de registro de eventos en línea

Propiedad/Método	Descripción
<i>altKey</i>	Devuelve true si se ha pulsado la tecla ALT y false en otro caso
<i>button</i>	El botón del ratón que ha sido pulsado. Posibles valores: 0 – Ningún botón pulsado 1 – Se ha pulsado el botón izquierdo 2 – Se ha pulsado el botón derecho 3 – Se pulsan a la vez el botón izquierdo y el derecho 4 – Se ha pulsado el botón central 5 – Se pulsan a la vez el botón izquierdo y el central 6 – Se pulsan a la vez el botón derecho y el central 7 – Se pulsan a la vez los 3 botones
<i>charCode</i>	El código Unicode del carácter correspondiente a la tecla pulsada
<i>clientX</i>	Coordenada X de la posición del ratón respecto del área visible de la ventana
<i>clientY</i>	Coordenada Y de la posición del ratón respecto del área visible de la ventana
<i>ctrlKey</i>	Devuelve true si se ha pulsado la tecla CTRL y false en otro caso
<i>isChar</i>	Indica si la tecla pulsada corresponde a un carácter
<i>detail</i>	El número de veces que se han pulsado los botones del ratón

# modelo de registro de eventos en línea

Propiedad/Método	Descripción
<i>keyCode</i>	Indica el código numérico de la tecla pulsada
<i>metaKey</i>	Devuelve true si se ha pulsado la tecla META y false en otro caso (tecla <i>Windows</i> , <i>Manzana</i> ...)
<i>pageX</i>	Coordenada X de la posición del ratón respecto de la página
<i>pageY</i>	Coordenada Y de la posición del ratón respecto de la página
<i>screenX</i>	Coordenada X de la posición del ratón respecto de la pantalla completa
<i>screenY</i>	Coordenada Y de la posición del ratón respecto de la pantalla completa
<i>shiftKey</i>	Devuelve true si se ha pulsado la tecla SHIFT y false en otro caso
<i>relatedTarget</i>	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
<i>target</i>	Devuelve el elemento que desencadenó el evento
<i>type</i>	Devuelve el nombre del evento
<i>timeStamp</i>	Devuelve el tiempo (en milisegundos relativa a la época) en la que se desató el evento

# modelo de registro de eventos en línea

A continuación, se enumeran algunos de los eventos asociados a diferentes elementos HTML.

Hay ciertos Eventos del documento HTML completo asociados a la etiqueta `<body>`:

Atributo HTML	Descripción
<i>onLoad</i>	La página (el documento HTML) ha terminado de cargarse
<i>onunload</i>	La página (el documento HTML) va a cerrarse
<i>onscroll</i>	El usuario ha hecho <i>scroll</i> sobre la página (el documento HTML)
<i>onresize</i>	El usuario ha modificado el tamaño de la ventana

Ejemplo:

```
<body onLoad="cargaJuego()">
```

# modelo de registro de eventos en línea

Sobre las etiquetas que cargan un archivo externo, como `<img>`, `<script>` e incluso `<style>` con estilos CSS en línea.

Atributo HTML	Descripción
<i>onLoad</i>	El recurso ha terminado de cargarse en la página
<i>onunLoad</i>	El recurso ha sido eliminado de la página
<i>onabort</i>	El recurso ha sido cancelado y no ha terminado su carga
<i>onerror</i>	El recurso ha dado un error y no ha terminado su carga

Los archivos multimedia `<audio>` y `<video>` tienen sus propios eventos específicos, ya que tienen un proceso de carga especial.

# modelo de registro de eventos en línea

Sobre elementos multimedia como `<audio>` o `<video>`, donde se carga un recurso (*MP4, WebM, MP3, OGG...*) externo:

Atributo HTML	Descripción
<i>onEmpty</i>	El audio o video se ha vaciado (recargar elemento)
<i>onLoadedMetadata</i>	Se han precargado los metadatos del audio o video (duración, subs...)
<i>onLoadedData</i>	Se ha precargado el comienzo del audio o video
<i>onCanPlay</i>	El audio o video se ha precargado lo suficiente para reproducir
<i>onCanPlayThrough</i>	El audio o video se ha precargado completamente
<i>onPlay</i>	El audio o video comienza a reproducirse (tras haber sido pausado)
<i>onPlaying</i>	El audio o video comienza a reproducirse
<i>onPause</i>	El audio o video ha sido pausado
<i>onTimeUpdate</i>	El audio o video ha avanzado en su reproducción
<i>onEnded</i>	El audio o video ha completado su reproducción
<i>onWaiting</i>	El audio o video está esperando a que el buffer se complete
<i>onDurationChange</i>	El audio o video ha cambiado su duración total (metadatos)
<i>onRateChange</i>	El audio o video ha cambiado su velocidad de reproducción
<i>onVolumeChange</i>	El audio o video ha cambiado su volumen de reproducción
<i>onProgress</i>	El audio o video se está descargando
<i>onLoadStart</i>	El audio o video ha comenzado a descargarse
<i>onSuspend</i>	La precarga del audio o video ha sido suspendida (ok o pause)
<i>onAbort</i>	La precarga del audio o video ha sido abortada o reiniciada
<i>onError</i>	Ha ocurrido un error
<i>onStalled</i>	El navegador intenta precargar el audio o video, pero se ha estancado
<i>onSeeking</i>	El navegador comenzó a buscar un momento concreto del audio/video
<i>onSeeked</i>	El navegador terminó de buscar un momento concreto del audio/video
<i>onResize</i>	El video ha sido redimensionado

# modelo de registro de eventos en línea

Sobre etiquetas `<input>`, `<textarea>`, `<select>`, `<a>` o cualquier otra etiqueta, incluida `<body>`, que pueda ser seleccionable por el usuario pulsando la tecla tabulador, existen una serie de eventos para controlar cuando gana o pierde el foco un elemento:

Atributo HTML	Descripción
<code>onblur</code>	El elemento ha perdido el foco (foco de salida)
<code>onfocusout</code>	El elemento ha perdido el foco
<code>onfocus</code>	El elemento ha ganado el foco (foco de entrada)
<code>onfocusin</code>	El elemento ha ganado el foco
<code>onkeypress</code>	Una tecla es presionada y luego soltada mientras el elemento tiene el foco
<code>onkeydown</code>	Una tecla es presionada, independientemente de que sea soltada o no, mientras el elemento tiene el foco
<code>onkeyup</code>	Una tecla es soltada mientras el elemento tiene el foco

# modelo de registro de eventos en línea

Sobre elementos `<input>` o elementos HTML con el atributo `contentEditable`, y por lo tanto, elementos HTML que son editables por el usuario:

Atributo HTML	Descripción
<code>onbeforeinput</code>	Un elemento <code>&lt;input&gt;</code> o con atributo <code>contentEditable</code> a punto de cambiar
<code>oninput</code>	Un elemento <code>&lt;input&gt;</code> o con atributo <code>contentEditable</code> ha cambiado
<code>onchange</code>	Un elemento <code>&lt;input&gt;</code> o con atributo <code>contentEditable</code> pierde el foco y su contenido ha sido cambiado
<code>onselect</code>	Se ha seleccionado alguna parte del texto en un elemento <code>&lt;input&gt;</code> o con atributo <code>contentEditable</code>



# modelo de registro de eventos en línea

Sobre elementos `<form>` :

Atributo HTML	Descripción
<i>onsubmit</i>	Ocurre cuando el formulario es enviado
<i>onreset</i>	Ocurre cuando el formulario es reestablecido a sus valores por defecto

# modelo de registro de eventos en línea

Los eventos de ratón se utilizan para detectar todas aquellas acciones que el usuario realiza mediante el ratón con algún elemento de la página, como podría ser mover el ratón por encima de ellos, hacer clic, mover la rueda del ratón, etc...

Atributo HTML	Descripción
<i>onclick</i>	El usuario ha pulsado (y soltado) el elemento
<i>ondblclick</i>	El usuario ha hecho doble clic en el elemento
<i>onmousedown</i>	El usuario ha pulsado (aún sin haber soltado) el elemento
<i>onmouseup</i>	El usuario ha soltado el botón pulsado en un elemento
<i>onmousemove</i>	El usuario ha movido el ratón
<i>onmouseenter</i>	El usuario ha movido el ratón dentro de un elemento
<i>onmouseleave</i>	El usuario ha movido el ratón fuera de un elemento
<i>onmouseout</i>	El usuario ha movido el ratón fuera de un elemento
<i>onmouseover</i>	El usuario ha movido el ratón dentro de un elemento
<i>onwheel</i>	El usuario ha movido la rueda del ratón

## modelo de eventos mediante propiedades

Con la llegada de la metodología DHTML (*Dynamic HTML: HTML+CSS+JS*), el modelo de gestión de eventos se extendió para ser más flexible, pasando a ser una propiedad del elemento.

En este modelo, la gestión de eventos se realiza desde JS, separando el código funcional del lenguaje HTML presente en la página web.

El mecanismo es el mismo que en el modelo anterior, pero haciendo uso de una propiedad especial precedida por el prefijo *on* a la que se asigna la función con el código deseado.

El termino DHTML está en desuso ya que cualquier cosa que se haga con JS, como AJAX, animaciones, sliders, carruseles, validación de formularios, etc., se considera DHTML y prácticamente todas las webs tienen elementos dinámicos.

```
function confirmar(){  
    return confirm("¿Deseas cambiar a la página indicada?");  
}
```

```
const elemento= document.querySelector('a');  
elemento.onclick=confirmar;
```

## modelo de eventos mediante propiedades

Se pueden observar ciertas diferencias con respecto al modelo de gestión de eventos anterior, por ejemplo, *onClick* se escribe en minúsculas, además, no se usan los paréntesis junto con el nombre de la función cuando se asigna a la propiedad del elemento, *onClick*, pues no se quiere invocar la función solo asignarla.

De igual manera, se evita la acción por defecto en función de si la función que maneja el evento devuelve *true* o *false*; con *true* se ejecuta la acción por defecto con *false* no se ejecuta.

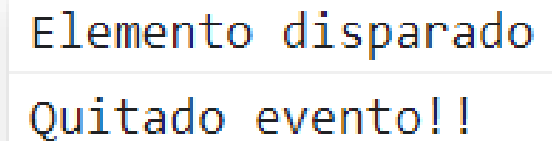
```
function confirmar(){  
    return confirm("¿Deseas cambiar a la página indicada?");  
}  
  
const elemento= document.querySelector('a');  
elemento.onClick=confirmar;
```

# modelo de eventos mediante propiedades

La ventaja de este modelo con respecto al anterior es que se pueden asociar y desasociar eventos dinámicamente, desde código JS, sin necesidad de hacerlo en la página web, a través del lenguaje HTML.

Para desasociar eventos de un elemento se le asigna la palabra reservada *null* o *undefined*.

```
const procesaEvento=(e)=>{  
    console.log("Elemento disparado");  
    e.target.onclick=null;  
    console.log("Quitado evento!!");  
}  
const a=document.querySelector("a");  
a.onclick=procesaEvento;
```



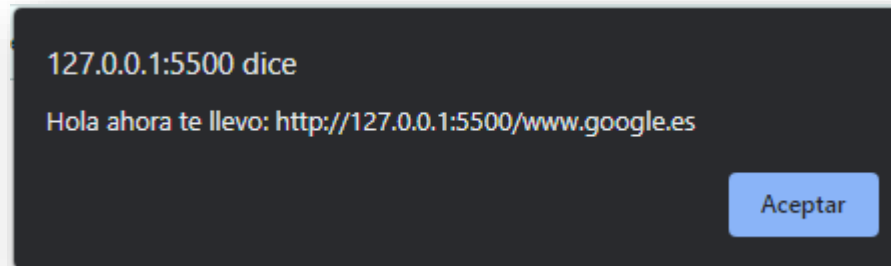
Elemento disparado  
Quitado evento!!

# modelo de eventos mediante propiedades

La palabra reservada *this* en este modelo de eventos hace referencia al objeto donde se ha programado el evento.

Ejemplo, haciendo uso de *hoisting*:

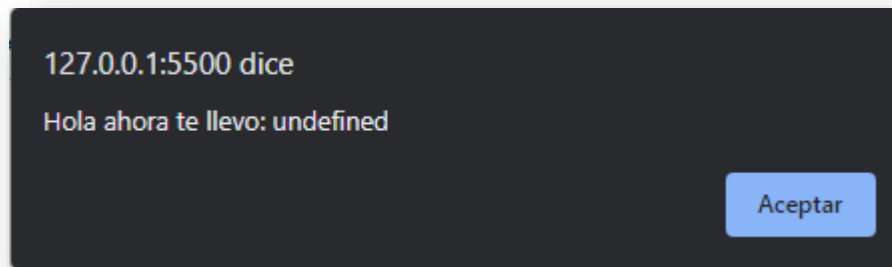
```
document.querySelector("a").onclick=saludo;  
function saludo(){alert("Hola ahora te llevo: "+this.href);return false;};
```



# modelo de eventos mediante propiedades

Hay ciertas diferencias con respecto al uso de una función flecha por ejemplo no se puede hacer uso *hoisting*, y el uso de la palabra reservada *this*, provoca *undefined*, ya que *this* hace referencia al objeto *window* y no al elemento donde se ha programado el evento, en este caso el enlace:

```
const saludo=(()=>{alert("Hola ahora te llevo: "+this.href);return false;});  
document.querySelector("a").onclick=saludo;
```



## modelo de eventos estandarizado DOM

El W3C, en su especificación del DOM de nivel 2, pone especial atención a los problemas del modelo tradicional de registro de eventos, el modelo mediante propiedades, y ofrece una forma sencilla de registrar sobre un objeto determinado los eventos que se quieran.

Así, la especificación DOM define dos métodos denominados *addEventListener()* y *removeEventListener()* para asociar y desasociar *Event Handlers* o manejadores de eventos.

Al igual que el modelo de eventos anterior, permite gestionar eventos desde código JS sin necesidad de hacerlo desde el lenguaje HTML.

Es posible añadir *Event Listeners* a cada uno de los elementos mediante JS gracias a que todos ellos implementan la interfaz *EventTarget*.



# modelo de eventos estandarizado DOM

Este modelo estandarizado de eventos es el recomendado en la actualidad y aporta ciertas ventajas sobre los métodos de eventos anteriores:

- Se puede agregar muchos gestores de eventos del mismo tipo a un elemento, es decir, dos eventos de "clic".
- Permite agregar más de una función de respuesta a un solo evento.
- Da un control mas detallado de la fase en la que el *listener* se activa (*capturing* o *bubbling*).
- Funciona con cualquier elemento del DOM, no solo con elementos de HTML, como, por ejemplo, se pueden añadir al objeto *window*.

# [addEventListener]

El método `addEventListener(type, callback [, opciones/fase=false])` permite añadir un *listener* o una escucha de un evento a un elemento.

El primer argumento no es el nombre completo del evento, como sucede en los modelos de eventos anteriores, sino que se debe eliminar el prefijo *on*. En otras palabras, si se utilizaba el nombre *onclick*, ahora se debe utilizar *click*.

El segundo argumento es el gestor de eventos, la función que será ejecutada cuando se produzca dicho evento.

El tercer argumento tiene carácter opcional, por defecto es `false` indicando que el *listener* opera en la fase de *bubbling*, si fuera `true` se emplea en la fase de *capturing*.

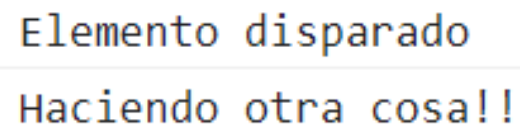
```
const elemento=document.querySelector('button');
elemento.addEventListener("click",(event)=>console.log('clic en botón'));
```

# [addEventListener]

El método *addEventListener()* agrega un gestor o manejador de eventos a un elemento sin sobrescribir los gestores o manejadores de eventos ya existentes.

Ejemplo:

```
const procesaEvento1=(e)=>console.log("Elemento disparado");  
const procesaEvento2=(e)=>console.log("Haciendo otra cosa!!");  
  
const a=document.querySelector("a");  
a.addEventListener("click",procesaEvento1);  
a.addEventListener("click",procesaEvento2);
```



```
Elemento disparado  
Haciendo otra cosa!!
```

# [addEventListener]

El tercer argumento del método `addEventListener()` permite a través de un objeto definir el comportamiento cuando se produzca el evento.

Opción booleana	Descripción: por defecto es <i>false</i>
<i>capture</i>	Si es true el evento se dispara al inicio ( <i>capture</i> ), en lugar de al final ( <i>bubble</i> ).
<i>once</i>	Sólo ejecuta la función la primera vez. Luego, elimina <i>listener</i> .
<i>passive</i>	La función nunca llama a <i>preventDefault()</i> , por lo que siempre se ejecuta la acción por defecto del evento.

```
const procesaEvento=(e)=>console.log("Elemento disparado");  
  
const a=document.querySelector("a");  
a.addEventListener("click",procesaEvento,{once: true});
```

## [removeEventListener]

El método *removeEventListener(type, callback [, opciones])* permite quitar un *listener* de un evento a un elemento previamente registrado con *addEventListener()*.

El *listener* del evento a quitar se identifica usando una combinación del tipo de eventos, la función usada como gestora del evento, e incluso la fase de captura del evento o las opciones si las tuviera.

```
element.addEventListener("mousedown", mi_funcion, true);
```

```
element.removeEventListener("mousedown", mi_funcion, false); // Fallo  
element.removeEventListener("mousedown", mi_funcion, true);  // Éxito
```

## [preventDefault]

El método *preventDefault()* cancela el evento si este es cancelable, sin detener el resto del funcionamiento del evento, es decir, puede ser llamado de nuevo.

Permite, por ejemplo, cancelar la acción por defecto asociada al elemento, como pudiera ser una etiqueta `<a>` cuya acción por defecto es navegar a lo que indique su atributo *href*.

```
const procesaEvento=(e)=>{
    e.preventDefault();
    console.log("Elemento disparado");
}

const a=document.querySelector("a");
a.addEventListener("click",procesaEvento);
```

Elemento disparado

# [propagación de eventos]

Los navegadores incluyen un mecanismo relacionado con la gestión de los eventos llamado flujo de eventos o *Event Flow*.

El flujo de eventos permite que varios elementos diferentes o incluso manejadores de eventos distintos puedan responder a un mismo evento, como ya se ha visto en ejemplos anteriores.

Si en una página web se define un elemento `<div>` con un botón en su interior, cuando el usuario pulsa sobre el botón, el navegador permite asignar una función de respuesta al botón, otra función de respuesta al `<div>` que lo contiene y otra función de respuesta a toda la página web completa.

De esta forma, un solo evento, la pulsación de un botón, provoca la respuesta de tres elementos de la página, incluyendo la propia página web.

El orden en el que se ejecutan los eventos asignados a cada elemento de la página es lo que constituye el flujo de eventos. Además, existen muchas diferencias en el flujo de eventos de cada navegador.

# [event bubbling]

*Event Bubbling* o evento burbujeante es un símil en referencia al efecto de las burbujas ascendiendo hacia arriba.

En este mecanismo de flujo de eventos, el orden que se sigue para responder a los eventos asignados es desde el elemento más específico hasta el elemento menos específico.

Ejemplo:

Aunque se hace clic en el `<div>` el evento se propaga hacia arriba, como una burbuja, disparando eventos en los elementos superiores como `<body>` y `<html>`.

```
<!DOCTYPE html>
<html lang="es" onclick="procesa(event)">
<head>
  <meta charset="UTF-8"><title>Documento</title></head>
<body onclick="procesa(event)">
  <div onclick="procesa(event)">Pincha aquí</div>
  <script src="js/eventos.js"></script>
</body>
</html>
```

```
const procesa=(e)=>console.log(e.currentTarget.nodeName);
```

DIV
BODY
HTML

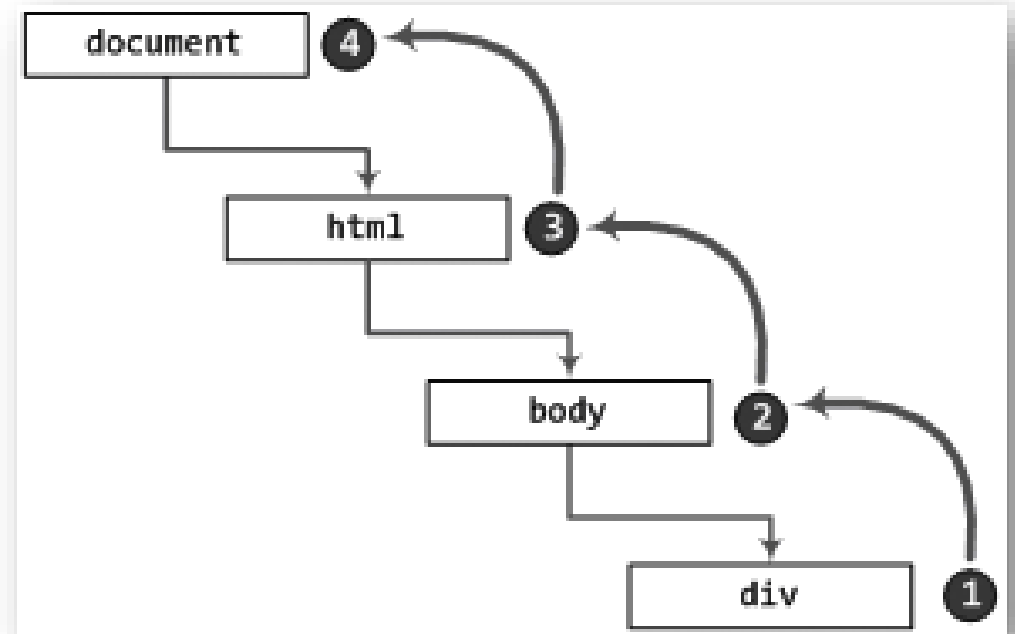


## [event bubbling]

En el ejemplo anterior, cuando el usuario hace clic con el ratón sobre el texto "Pincha aquí", que se encuentra dentro del `<div>`, se ejecutan los siguientes eventos en el orden que muestra en el esquema.

El primer evento que se tiene en cuenta es el generado por el `<div>` que contiene el mensaje. A continuación, el navegador recorre los ascendentes del elemento hasta que alcanza el nivel superior, que es el elemento *document*.

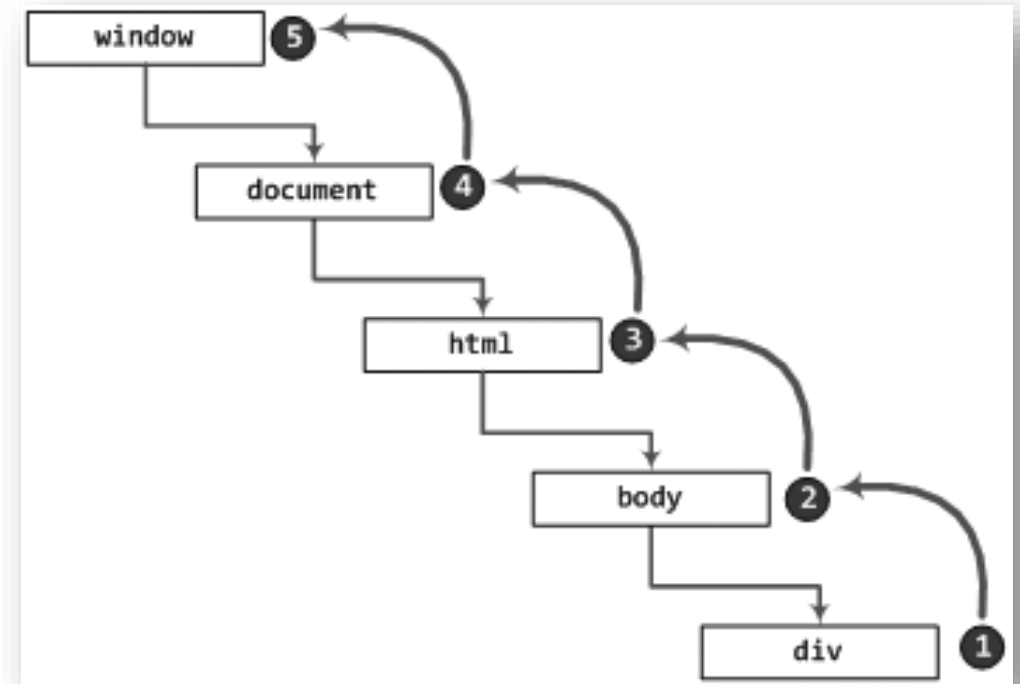
Este modelo de flujo de eventos es el que incluyen la mayoría de los navegadores.



# [event bubbling]

Existen ligeras variaciones, como los navegadores de la familia Mozilla (por ejemplo, Firefox), que en el ejemplo anterior presenta el siguiente flujo de eventos.

Aunque el objeto *window* no es parte del DOM, el flujo de eventos implementado por Mozilla recorre los ascendentes del elemento hasta el mismo objeto *window*, añadiendo por tanto un evento más al modelo.



## [event bubbling]

Casi todos los eventos burbujan, pero existen eventos como, por ejemplo, el evento *focus* que no burbujea.

Para saber cuál fue el elemento en el que se causó o se provocó el evento inicial se llama elemento de objetivo y es accesible a través de la propiedad *event.target*, no cambia a través del proceso de propagación.

Existe también la propiedad *event.currentTarget* que permite hacer referencia al evento actual, que puede ser el original, el target, o el propagado.

```
const procesa=(e)=>console.log(e.target.nodeName,"->",e.currentTarget.nodeName);
```

P -> P

P -> DIV

P -> BODY

P -> HTML

HTML -> HTML

# [event capturing]

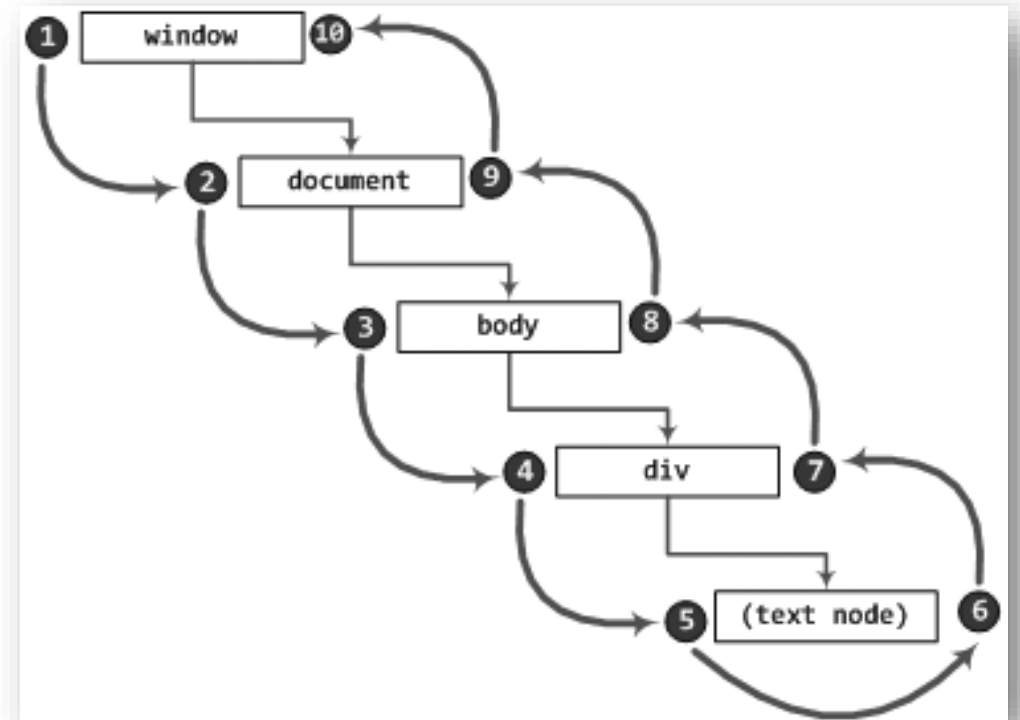
*Event Capturing*, *capturing* o captura de eventos, es otro mecanismo de flujo de eventos, pero en este mecanismo se define desde el elemento menos específico hasta el elemento más específico.

El mecanismo definido es justamente el contrario al *Event Bubbling*.

El flujo de eventos definido en la especificación DOM soporta tanto el *bubbling* como el *capturing*, pero este último se realiza en primer lugar.

Los dos flujos de eventos recorren todos los objetos DOM desde el objeto *document* hasta el elemento más específico y viceversa.

En la mayoría de los navegadores continúa el flujo hasta el objeto *window*.



# [event capturing]

Ejemplo:

```
const procesaEvento=(e)=>console.log(`${e.currentTarget.nodeName}  
->  
Fase: ${e.eventPhase==1? "Fase capturing"  
      : e.eventPhase==2? "En target"  
      : "Fase bubbling"}}`);  
  
const p=document.querySelector("p"),  
div=document.querySelector("div"),  
article=document.querySelector("article"),  
body=document.querySelector("body"),  
html=document.querySelector("html");  
  
p.addEventListener("click",procesaEvento,true);  
div.addEventListener("click",procesaEvento);  
article.addEventListener("click",procesaEvento,true);  
body.addEventListener("click",procesaEvento);  
html.addEventListener("click",procesaEvento,true);
```

```
HTML ->  
Fase: Fase capturing  
  
ARTICLE ->  
Fase: Fase capturing  
  
P ->  
Fase: En target  
  
DIV ->  
Fase: Fase bubbling  
  
BODY ->  
Fase: Fase bubbling
```

## [event capturing]

A veces resulta útil *capturing* para capturar eventos en elementos superiores de elementos objetivo que no hacen *bubbling* o burbujan.

Por ejemplo, si se quisiera capturar el evento *focus* en el objeto *document*, como el evento *focus* no burbuja, podría hacerse en la fase de captura.

```
document.addEventListener('blur', (e)=>console.log(`capturado ${e.type}`), true);  
const input=document.querySelector('input');  
input.addEventListener('blur', (e)=>console.log('desde input'));
```

```
capturado blur  
desde input
```

# stopPropagation

Como se ha visto, el *Event bubbling* y *Event Capturing* provocan que se vaya propagando el evento llamando a todos los gestores del evento que encuentra en la ruta.

Pero cualquier gestor del evento puede decidir que el evento se ha procesado por completo y detener la propagación. Para ello se usa el método:

`event.stopPropagation()`

```
const procesaEvento=(e)=>{
  console.log(`${e.currentTarget.nodeName} ->
  Fase: ${e.eventPhase==1? "Fase capturing"
    : e.eventPhase==2? "En target"
    : "Fase bubbling"}`);
  if(e.currentTarget.nodeName=="HTML")
    e.stopPropagation();
};

p.addEventListener("click",procesaEvento,true);
div.addEventListener("click",procesaEvento);
article.addEventListener("click",procesaEvento,true);
body.addEventListener("click",procesaEvento);
html.addEventListener("click",procesaEvento,true);
```

HTML ->

Fase: Fase capturing

# `stopImmediatePropagation`

Si un elemento tiene varios controladores de eventos en un solo evento, aunque uno de ellos detenga la propagación, los otros aún se ejecutarán y harán la propagación.

Para detener la propagación y evitar que se ejecuten los manejadores del elemento actual, existe el método `stopImmediatePropagation()`.

Después de su ejecución, no se ejecuta ningún otro controlador.

```
const procesaEvento1=(e)=>{e.stopImmediatePropagation();  
  console.log("Elemento disparado");};  
const procesaEvento2=(e)=>console.log("Otra cosa!!");  
  
const p=document.querySelector("p"),  
div=document.querySelector("div"),  
article=document.querySelector("article"),  
body=document.querySelector("body"),  
html=document.querySelector("html");  
  
p.addEventListener("click",procesaEvento1);  
p.addEventListener("click",procesaEvento2);
```

Elemento disparado



# [setInterval()]

Es una función que permite ejecutar una función o un trozo de código de forma repetida en el tiempo.

Normalmente se usa para ejecutar funciones recurrentes cada cierto tiempo, como por ejemplo pintar el siguiente *frame* en una animación.

Requiere la función o código a ejecutar como primer parámetro y el tiempo en milisegundos como segundo parámetro que indica el intervalo de tiempo que será llamada la función. Si la función invocada requiere parámetros adicionales pueden especificarse a continuación como parámetros opcionales de *setInterval()*.

La función devuelve un número entero como identificador único que permite cancelar *setInterval()*.

La sintaxis es la siguiente:

*ID=setInterval(código, intervaloDeTiempo)*      o

*ID=setInterval(función, intervaloDeTiempo[, parámetro1, parámetro2, ..., parámetroN])*

## [setInterval()]

El siguiente fragmento de código, a modo de ejemplo, permite crear un reloj que se ejecuta cada segundo, reloj es un elemento HTML como un *span* de la página web donde se muestra el mismo:

```
id=setInterval(() => {  
    time=reloj.textContent;  
    const array=time.split(":");  
    if ((array[2]*1)+1>=60){  
        array[2]="00";  
        if ((array[1]*1)+1>=60){  
            array[1]="00";  
            if ((array[0]*1)+1>=24) array[0]="00";  
            else ((array[0]*1)+1)<10?array[0]="0"+((array[0]*1)+1):array[0]=((array[0]*1)+1);  
        }  
        else ((array[1]*1)+1)<10?array[1]="0"+((array[1]*1)+1):array[1]=((array[1]*1)+1);  
    }  
    else ((array[2]*1)+1)<10?array[2]="0"+((array[2]*1)+1):array[2]=((array[2]*1)+1);  
    reloj.textContent=array.join(":");  
}, 1000);
```

# `clearInterval()`

Esta función permite detener la ejecución de *setInterval()*.

Recibe como argumento el ID que se genera cuando se invoca la función *setInterval()*.

La sintaxis es la siguiente:

*clearInterval(ID)*

Ejemplo:

```
Let id;  
  
function molestar(){  
    alert("Hola, Bienvenido!!");  
    clearInterval(id);  
}  
  
id=setInterval(molestar, 5000);
```

# [ `setTimeout()` ]

Permite ejecutar una función o un trozo de código después de que haya pasado una determinada cantidad de tiempo. A diferencia de *setInterval()* la función solo es llamada una única vez.

Requiere la función o el código a ejecutar como primer parámetro y el tiempo en milisegundos como segundo parámetro que indica el tiempo que ha de esperarse para que sea llamada la función. Si la función invocada requiere parámetros adicionales pueden especificarse a continuación como parámetros opcionales de *setTimeout()*.

La función devuelve un número entero como identificador único que permite cancelar *setTimeout()*.

La sintaxis es la siguiente:

*ID=setTimeout(función[, tiempo, arg1, arg2, ...])*      o

*ID=setTimeout(función[, tiempo])*      o

*ID=setTimeout(código[, tiempo])*

# [ setTimeout() ]

Ejemplos:

```
let colores=[0,0,0], incremento = 1;
const cambia_fondo=(()=>{
  if (colores[0]==250)
    incremento =-1;
  if (colores[0]==0)
    incremento = 1;
  colores.forEach((valor,i)=>colores[i]=valor+incremento);
  fondo.style.backgroundColor = 'rgb(' + colores.toString() + ')';
  setTimeout(cambia_fondo, Math.floor(Math.random() * 5000));
})
setTimeout(cambia_fondo, 5000);
```

```
setTimeout(()=>console.log("Han pasado 3 segundos"),3000);
```

# `clearTimeout()`

Esta función permite detener la ejecución de *setTimeout()*.

Recibe como argumento el ID que se genera cuando se invoca la función *setTimeout()*.

La sintaxis es la siguiente:

*clearTimeout(ID)*

Ejemplo:

```
let id=setTimeout(console.log("Han pasado 3 segundos"),3000);  
clearTimeout(id);
```

# [load vs DOMContentLoaded]

El evento *load* se produce sobre el objeto *window* cuando toda la página y los archivos que incluye, como imágenes, archivos JS, CSS..., se han descargado y ya están presentes en la página web.

Se registra como en el siguiente ejemplo:

```
window.addEventListener('load', funcion);
```

El evento *DOMContentLoaded* se produce sobre el objeto *document* y se produce cuando todo el código HTML de la página web se ha descargado y se está ejecutando, aunque queden dependencias por descargar, como imágenes, archivos JS, CSS...

Se registra como en el siguiente ejemplo:

```
document.addEventListener('DOMContentLoaded', funcion);
```

# Bibliografía y recursos online

- <https://profile.es/blog/patrones-de-diseno-de-software/>
- <https://developer.mozilla.org/es/docs/Web/Events>
- <https://developer.mozilla.org/es/docs/Web/API/setInterval>