Unidad de Trabajo 5: Gestión de Eventos y Formularios en JavaScript

1. Introducción

En esta unidad de trabajo vamos a desarrollar herramientas cruciales para el trabajo en el entorno cliente mediante javaScript: el manejo de eventos, formularios, expresiones regulares y cookies. Estos elementos, no sólo constituyen la columna vertebral de las aplicaciones web modernas en *front-end*, sino que también desempeñan un papel esencial en la creación de experiencias interactivas y eficientes para los usuarios.

El punto de partida será el manejo de eventos, ya que el dominio efectivo de de los mismos en JavaScript es la puerta de entrada a la interactividad dinámica. Desde entender el modelo de registro de eventos hasta explorar el flujo de eventos y el objeto Event, esta unidad proporciona las herramientas clave para crear experiencias web que respondan y se adapten a las acciones del usuario.

Asimismo, los formularios son una de las interfaces principales a través de la cual los usuarios interactúan con nuestras aplicaciones. Aprenderemos no solo a capturar datos eficientemente sino también a validarlos en el lado del cliente, mejorando la usabilidad, el flujo de información y la seguridad de nuestras aplicaciones. Como parte del estudio de la validación de formularios, las expresiones regulares se convierten en aliadas poderosas para llevar a cabo dicha tarea, al permitir verificar que los campos de entrada introducidos por el usuario son correctos antes de enviarlos al servidor.

Finalmente, veremos una introducción al manejo de cookies para el almacenamiento de datos de navegación del cliente, herramienta esencial para mejorar la experiencia del usuario y la facilidad de navegación.

2. Eventos

Hasta este momento, has desarrollado tus aplicaciones web de un modo tradicional, es decir, de tal modo que las mismas se ejecuten secuencialmente desde la primera hasta la última instrucción produciendo los resultados previamente definidos. Igualmente, y mediante determinadas estructuras de control de flujo (p.ej., if, for y while), es posible modificar “ligeramente” el comportamiento de tus aplicaciones en función de determinadas condiciones. No obstante, este tipo de aplicaciones son poco útiles, ya que no interactúan con los usuarios y no pueden responder a los diferentes eventos que se producen durante la ejecución de una aplicación.

Por suerte, las aplicaciones web creadas con el lenguaje JavaScript pueden utilizar el **modelo de programación basada en eventos**. Es más, mediante JavaScript vas a poder crear aplicaciones web que implementen ambos tipos de comportamientos, lo que va a permitir mejorar la experiencia del usuario en la web. En este tipo de programación basado en eventos, los scripts están diseñados para que respondan al comportamiento del usuario

pero **también** podrás diseñar eventos que se produzcan sin ningún tipo de interacción (como, p.ej., eventos de carga de la web, etc.)

JavaScript permite, así mismo, asignar una función a cada uno de los eventos. De esta forma, cuando se produce cualquier evento, JavaScript ejecuta su función asociada. Este tipo de funciones se denominan "event handlers" en inglés y suelen traducirse por "manejadores de eventos".

2.1. Modelo de registro de eventos

Crear páginas y aplicaciones web siempre ha sido mucho más complejo de lo que debería serlo debido a las **incompatibilidades** entre navegadores. A pesar de que existen decenas de estándares para las tecnologías empleadas, los navegadores no los soportan completamente o incluso los ignoran.

Las principales incompatibilidades se producen en el lenguaje XHTML, en el soporte de hojas de estilos CSS y sobre todo, en la **implementación de JavaScript**. De todas ellas, la incompatibilidad más importante se da precisamente en el modelo de eventos del navegador. Así, con el tiempo se fueron desarrollando diferentes modelos de registro y manejo de eventos. No obstante, cabe destacar que actualmente la compatibilidad entre los principales navegadores es alta puesto que los mismos tienen a desarrollar dichas tecnologías con los estándares actualmente establecidos. Así, los principales modelos de registro son: el **modelo en línea o mediante atributos HTML**; el **modelo tradicional o mediante funciones de javaScript** y, finalmente, un **modelo avanzado mediante listeners** (siendo este último el más empleado en la actualidad).

2.1.1. Modelo de registro de eventos en línea

Este método consiste en añadir el evento como un atributo más a la etiqueta de HTML. Se trata de la forma más fácil de trabajar con eventos y fue la primera en desarrollarse. No obstante, aunque resulte un método más sencillo es poco recomendable a día de hoy puesto que tendremos en el mismo archivo tanto el código HTML como JavaScript, siendo la tendencia actual separar los mismos con el fin de que los trabajos sean más fáciles de mantener y claros.

En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pinche con el ratón sobre un botón:

```
<input type="button" value="Pinchame y verás" onclick="alert('Gracias por pinchar');" />
```

También se podría realizar lo mismo pero llamando a una función definida de forma anónima:

```
<a href="pagina.html" onclick="function () {alert("Has pulsado en el enlace");}">Pulsa aqui</a>
```

O incluso llevar a cabo más de un evento:

```
<div onclick="console.log('Has pinchado con el ratón');" onmouseover="console.log('Acabas de pasar el ratón por encima');">
    Puedes pinchar sobre este elemento o simplemente pasar el ratón por encima
</div>
```

Tal y como puedes observar, a la hora de escribir los nombres de los eventos la buena práctica consiste en **declararlos en minúsculas.** No obstante, HTML no presenta problemas puesto que es insensible a mayúsculas y minúsculas. En cambio en XHTML, sí que los atributos tendrán que ir obligatoriamente siempre en minúsculas puesto que en este caso sí es *case-sensitive*.

Cabe destacar que, en el caso de etiquetas HTML que presentan una acción por defecto, si se añade funcionalidad a las mismas, el orden de ejecución será **en primer lugar el script y en segundo lugar la acción por defecto**. Por tanto, en el ejemplo siguiente primero recibiremos la alerta y a continuación seremos redirigidos a la página web:

```
<a href="https://www.bing.com/" onclick="alert('Has pulsado en el enlace');">Pulsa aquí</a>
```

¿Cómo podemos evitar la acción por defecto?

En ciertas ocasiones podemos requerir bloquear o evitar que se ejecute esa acción por defecto. Más adelante veremos la forma actual y más útil pero conviene que sepas que mediante este modelo de gestión de eventos en línea, si desarrollamos una acción o evento que devuelva un valor booleano de **false**, el comportamiento por defecto de esa etiqueta no se llevará a cabo. Por ejemplo, si modificamos el ejemplo anterior:

```
<a href="https://www.bing.com/" onclick="alert('Has pulsado en el enlace'); return false">Pulsa aquí</a>
```

En este caso, al pulsar en el enlace se ejecutará la alerta tal y como está configurada pero no se llevará a cabo la acción de redirigir al usuario a la página web indicada mediante el atributo *href*. También podría ser interesante configurar ese comportamiento para que sea el usuario el que quiera decidir entre si quiere o no que se ejecute esa acción. Por ejemplo:

```
<a href="https://www.bing.com/" onclick="return confirm('Va a ser redirigido a una nueva página web. Pulse aceptar para continuar');">Pulsa aquí</a>
```

Es importante fijarse que en este modo de configuración se debe emplear la palabra reservada **return** puesto que de no ser así, y aunque el confirm devolverá un valor de false si el usuario cancela, el comportamiento por defecto de la etiqueta será ejecutado.

Una forma un poco más avanzada de este tipo de manejo de eventos es la de definir las **funciones de modo externo** en las etiquetas de <script> o en un archivo .js y posteriormente hacer el llamado a la función en el atributo de la etiqueta HTML en cuestión. En ese caso, nuestro ejemplo anterior quedaría como:

```
<a href="https://www.bing.com/" onclick="return redirigir();">Pulsa aquí</a>
<script>
  function redirigir(){
      return confirm('Va a ser redirigido a una nueva página web. Pulse aceptar para continuar')
  }
</script>
```

Es importante destacar que en este ejemplo concreto se está procurando evitar el comportamiento por defecto por lo que antes de llamar a la función **redirigir()** se debe emplear la palabra reservada **return.** En casos aún más sencillos se puede omitir dicha palabra escribiendo únicamente el nombre de la función a ejecutar.

Cabe destacar, asimismo, que en este modelo de ejecución de eventos es posible emplear la palabra reservada **this** con el fin de simplificar un poco el código para que quede más limpio y ordenado. Por ejemplo, imaginemos que queremos configurar un para que cuando el

usuario se desplace por encima se cambie el color del borde a negro. El código, en principio, sería el siguiente:

```
<div id="contenidos" style="width:150px; height:60px; border:thin solid silver"
onmouseover="document.getElementById('contenidos').style.borderColor='black';"
onmouseout="document.getElementById('contenidos').style.borderColor='silver';">

  Sección de contenidos...

</div>
```

Ahora bien, si usamos la palabra **this** podemos simplificar dicho código ya que la misma hará referencia a la etiqueta que provoca el evento. Quedaría como:

```
<div id="contenidos" style="width:150px; height:60px; border:thin solid silver"
onmouseover="this.style.borderColor='black';"
onmouseout="this.style.borderColor='silver';">

  Sección de contenidos...

</div>
```

Que tal y como puedes ver es más compacto y cómodo de leer e interpretar.

En el caso de emplear **funciones externas**, es necesario pasar la palabra **this** como un parámetro de la función manejadora del evento:

```
<div style="padding: .2em; width: 150px; height: 60px; border: thin solid silver"
onmouseover="resalta(this)" onmouseout="resalta(this)">
 Sección de contenidos...
</div>
<script>
function resalta(elemento) {
 switch(elemento.style.borderColor) {
   case 'silver':
   case 'silver silver silver silver':
   case '#c0c0c0':
     elemento.style.borderColor = 'black';
     break;
   case 'black':
   case 'black black black black':
   case '#000000':
     elemento.style.borderColor = 'silver';
     break;
 }
}
</script>
```

En el ejemplo anterior, a la función externa se le pasa el parámetro **this**, que dentro de la función se denomina elemento. Al pasar **this** como parámetro, es posible acceder de forma directa desde la función externa a las propiedades del elemento que ha provocado el evento.

Por otra parte, el ejemplo anterior se complica por la forma en la que los distintos navegadores almacenan el valor de la propiedad borderColor. Mientras que Firefox almacena (en caso de que los cuatro bordes coincidan en color) el valor simple black, Internet Explorer lo almacena como black black black black y Opera almacena su representación hexadecimal #000000.

2.1.2. Modelo de registro de eventos tradicional

Este modelo también se conoce como **semántico.** En este nuevo modelo el evento pasa a ser una propiedad del elemento. Esta técnica surgió por la tendencia de separar el código HTML del código JavaScript. Así, por ejemplo, podemos escribir sencillamente en nuestro código HTML:

```
<input id="pinchable" type="button" value="Pinchame y verás" />
<script src="00.js"></script>
```

Y en el archivo “00.js”:

```
function muestraMensaje() {
   alert('Gracias por pinchar');
}
document.getElementById("pinchable").onclick = muestraMensaje;
```

El código HTML resultante es muy "limpio", ya que no se mezcla con el código JavaScript. La técnica de los manejadores semánticos se puede resumir en:

- Asignar un identificador único al elemento HTML mediante el atributo id.
- Crear una función de JavaScript encargada de manejar el evento.
- Asignar la función a un evento concreto del elemento HTML mediante DOM.
- Como parte del mantenimiento de tu web, si se desea eliminar un elemento únicamente se requiere asignarle un valor de **null**. Por ejemplo:

```
document.getElementById("pinchable").onclick = null;
```

Asimismo, mediante este modelo semántico puedes emplear funciones anónimas para asociar más de una función al mismo evento. Por ejemplo:

```
function muestraMensaje() {
   alert('Hola mundo!');
}
function muestraOtroMensaje() {
   alert('Gracias por pinchar');
}
document.getElementById("pinchable").onclick = function () {muestraMensaje(); muestraOtroMensaje()};
```

Es importante fijarse aquí que cuando se declare una función anónima con el fin de llamar a otras funciones previamente definidas, **las mismas tienen que estar declaradas con el paréntesis puesto que lo que queremos es "ejecutar"** esa función, no asignarla como propiedad.

El único inconveniente de este método es que los manejadores se asignan mediante las funciones DOM, **que solamente se pueden utilizar después de que la página se ha cargado** completamente. De esta forma, para que la asignación de los manejadores no resulte errónea, es necesario asegurarse de que la página ya se ha cargado. Para ello se puede emplear el evento **onload** mediante funciones anónimas en las instancias que toman las referencias de los objetos:

```
window.onload = function (){
   document.getElementById("pinchable").onclick = muestraMensaje;
}
```

Lo que permite obtener códigos HTML aún más limpios puesto que podemos cargar ahora el archivo .js desde cualquier punto del documento HTML como, por ejemplo, el encabezado del mismo.

Fíjate que en el registro del evento como una propiedad no usamos paréntesis (). El método onclick espera que se le asigne una función completa. Si defines el evento empleando los

paréntesis en la función (**error habitual en javaScript**), la función será ejecutada de modo directo sin que se produzca el evento, normalmente, al cargar la página.

De igual modo al modelo de ejecución en línea, puedes emplear la palabra reservada **this** para hacer referencia al objeto en el que hemos programado el evento. Por ejemplo:

```
<a id="mienlace" href="https://www.bing.com/">Pulsa aquí</a>
 <script>
    document.getElementById("mienlace").onclick = alertar;
    function alertar(){
       alert("Te conectaremos con la página: "+this.href);
    }
 </script>
```

2.1.3. Tipos de eventos más habituales en el modelo en línea y semántico

Cada elemento HTML tiene definida su propia lista de posibles eventos que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos HTML y un mismo elemento HTML puede tener asociados diferentes eventos.

El nombre de los eventos se construye mediante el prefijo **on**, seguido del nombre en inglés de la acción asociada al evento. La siguiente tabla resume algunos de los eventos más importantes y habitualmente empleados definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
onblur	Un elemento pierde el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Un elemento ha sido modificado	<input>, <select>, <textarea>
onclick	Pulsar y soltar el ratón	Todos los elementos
ondblclick	Pulsar dos veces seguidas con el ratón	Todos los elementos
onfocus	Un elemento obtiene el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla y no soltarla	Elementos de formulario y

		<body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón "sale" del elemento	Todos los elementos
onmouseover	El ratón "entra" en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos
onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

Los eventos más utilizados en las aplicaciones web tradicionales son **onload** para esperar a que se cargue la página por completo, los eventos **onclick, onmouseover, onmouseout** para controlar el ratón y **onsubmit** para controlar el envío de los formularios.

Cabe destacar que las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar, por ejemplo, sobre un botón de tipo <input type="submit"> se desencadenan los eventos onmousedown, onclick, onmouseup y onsubmit de forma consecutiva.

2.1.4. Modelo de registro avanzado de eventos según W3C

Hasta ahora, hemos visto que son los eventos Javascript y cómo gestionarlos a través de código HTML o a través de código Javascript de forma externa y semántica. Sin embargo, la forma más recomendable (y actual) para manejar los eventos es hacer uso del método **addEventListener()**, el cuál resulta mucho más potente y versátil en general. Asimismo, este modelo de manejo de eventos incorpora también los métodos **removeEventListener()** y **preventDefault()**.

La principal diferencia entre estos métodos y los vistos anteriormente es que se requieren tres parámetros para su manejo (siendo el tercero "opcional"): el nombre del **evento** que queremos captar, una **referencia a la función** encargada de procesar el evento y un **booleano** true o false que indica el tipo de flujo de eventos al que se aplica (*capture* o *bubbling*). Cabe destacar que este último parámetro es opcional y en caso de no declararse se toma por defecto el valor de **false**.

Así, los ejemplos anteriores según este nuevo modelo de gestión de eventos se manejaría como:

```
window.onload = function (){
        document.getElementById("mienlace").addEventListener('click',alertar,false);
    }
    function alertar(){
        alert("Te conectaremos con la página: " + this.href);
    }
```

E incluso, podemos asociar sencillamente varias funciones a un único evento como:

```
document.getElementById("mienlace").addEventListener('click',alertar,false);
document.getElementById("mienlace").addEventListener('click',avisar,false);
document.getElementById("mienlace").addEventListener('click',chequear,false);
```

Por lo tanto, cuando hagamos click en "mienlace" se disparará la llamada a las tres funciones. Sin embargo, cabe destacar que el W3C no indica el orden de disparo, por lo que no sabemos cuál de las tres funciones se ejecutará primero sino que el mismo vendrá gestionado por el navegador. Estudiaremos ese flujo de eventos posteriormente.

Tal y como hemos visto anteriormente, puedes configurar el comportamiento del evento mediante la declaración de funciones anónimas. No obstante, se debe ser cuidadoso con esta práctica puesto que si posteriormente queremos eliminar un evento con **removeEventListener**, aunque volvamos a instanciar de forma anónima la misma función en realidad el navegador las interpreta como diferentes y, por tanto, no se elimina el evento.

```
<script>
  window.onload = function (){
     document.getElementById("pinchable").addEventListener('click',  function
(){this.style.backgroundColor = '#cc0000'}, false)
  }
  </script>
  <input id="pinchable" type="button" value="Pinchame y verás" />
```

En el ejemplo anterior puedes ver, además, que la palabra reservada **this** cumple la misma función que en el modelo tradicional.

2.2. Flujo de los eventos

Tal y como hemos visto, puedes asociar a un mismo elemento varios eventos. Asimismo, cabe destacar que en el caso de tener varios elementos de modo que unos están contenidos dentro de otros (p.ej., un elemento con un botón en su interior), cuando el usuario interactúe con uno de los elementos se va a producir una propagación o flujo de eventos de modo que varios de los elementos puedan responder al mismo evento. Por ejemplo, en el caso de un botón contenido en un se puede manejar el flujo de eventos asignando funciones de respuesta al evento "click" tanto en el botón como en el de modo que ese evento se propague desde el botón hacia el div o viceversa. El orden en el que se ejecutan los eventos asignados a cada elemento de la página es lo que constituye el flujo de eventos y cabe destacar que, además, existen muchas diferencias en el flujo de eventos de cada navegador.

Por ejemplo, si copias y ejecutas el siguiente código podrás comprobar cómo se produce el llamado **burbujeo o bubbling**:

```
<div id="miDiv">
  <button id="miBoton">Haz clic aquí</button>
 </div>
 <script>
  function clickDiv() {
     alert("Se hizo clic en el div");
  }
  function clickBoton() {
     console.log("Se hizo clic en el botón");
  }
  document.getElementById("miDiv").addEventListener("click", clickDiv);
  document.getElementById("miBoton").addEventListener("click", clickBoton);
 </script>
```

Asimismo, otro modelo de flujo de eventos es el que se conoce como **captura o capture**. Vamos a ver a continuación la diferencia entre ambos.

Bubbling o modelo de burbujeo de eventos

Imagina que tienes una estructura HTML sencilla como la indica a continuación:

```
<html onclick="Evento()">
        <head><title>Ejemplo de flujo de eventos</title></head>
        <body onclick="Evento()">
                <div onclick="Evento()">Pincha aquí</div>
        </body>
</html>
```

Cuando se pulsa sobre el texto "Pincha aquí" que se encuentra dentro del , se ejecutan los siguientes eventos en el orden que muestra el siguiente esquema:

1. Evento en el DIV
2. Evento en el Body
3. Evento en el HTML
4. Evento en el Document
5. Evento en el Window

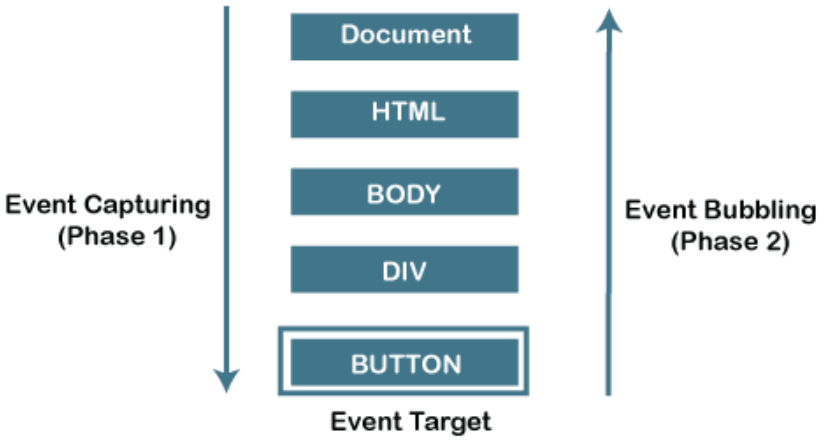
Aunque el objeto window no es parte del DOM, el flujo de eventos implementado por Mozilla recorre los ascendentes del elemento hasta el mismo objeto window, añadiendo por tanto un evento más al modelo de Internet Explorer.

Capture o modelo de captura de eventos

En ese otro modelo, el flujo de eventos se define desde el elemento menos específico hasta el elemento más específico. En otras palabras, el mecanismo definido es justamente el contrario al "event bubbling". Para comprobar cómo se produce el modelo de captura de eventos **modifica el ejemplo** al inicio de este apartado y compara qué evento aparece antes.

Modelo de gestión de eventos según el W3C

Con el fin de estandarizar el modelo de gestión de eventos, W3C decidió tomar una posición intermedia. Así supone que, cuando se produce un evento en su modelo de eventos, primero se producirá la **fase de captura** hasta llegar al elemento de destino, y luego se producirá la **fase de burbujeo** hacia arriba. Este modelo es el **estándar**, que todos los navegadores deberían seguir para ser compatibles entre sí.



Tú podrás decidir cuando quieres que se registre el evento: en la fase de captura o en la fase de burbujeo. El tercer parámetro de addEventListener te permitirá indicar si lo haces en la **fase de captura (true)**, o en la **fase de burbujeo(false)**.

Finalmente, para detener la propagación del evento en la **fase de burbujeo**, disponemos del método **stopPropagation()**. En cambio, en la fase de captura es imposible detener la propagación. A continuación te muestro un ejemplo de cómo modificar el código visto anteriormente para detener el burbujeo:

```
<div id="miDiv">
  <button id="miBoton">Haz clic aquí</button>
</div>
<script>
  function clickDiv() {
    alert("Se hizo clic en el div");
  }
  function clickBoton(event) {
    console.log("Se hizo clic en el botón");
    event.stopPropagation(); // Detener la propagación del evento
  }
  document.getElementById("miDiv").addEventListener("click", clickDiv);
  document.getElementById("miBoton").addEventListener("click", clickBoton);
</script>
```

En este código, la función clickBoton toma un parámetro **event** (que representa el objeto de evento). Luego, se llama al método stopPropagation() en ese objeto para detener la propagación del evento hacia arriba en la jerarquía del DOM, evitando que llegue al div.

2.3. El objeto Event

Cuando se produce un evento, generalmente la función que procesa el evento necesita información relativa al mismo tal y como, por ejemplo, la tecla que se ha pulsado, la posición del ratón, el elemento que ha producido el evento, etc.

El objeto **event** es el mecanismo definido por los navegadores para proporcionar toda esa información. Se trata de un objeto que se crea automáticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento. Por ejemplo, si ejecutas el siguiente código en tu navegador podrás observar cómo se muestra el objeto event en la consola:

```
<button>Click aquí!</button>

<script>
  const button = document.querySelector("button");
  button.addEventListener("click", (event) => {
    console.log(event);
  });
</script>
```

Cabe destacar que el estándar DOM especifica que el objeto **event** es el único parámetro que se debe pasar a las funciones encargadas de procesar los eventos. Dicho evento posee una serie de propiedades que te serán útiles a la hora de manejar los eventos en tu aplicación.

La siguiente tabla recoge las propiedades definidas para el objeto event en los navegadores que siguen los estándares:

Propiedad/Método	Devuelve	Descripción
altKey	Boolean	Devuelve true si se ha pulsado la tecla ALT y false en otro caso
bubbles	Boolean	Indica si el evento pertenece al flujo de eventos de bubbling
button	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: 0 – Ningún botón pulsado 1 – Se ha pulsado el botón izquierdo2– Se ha pulsado el botón derecho3– Se pulsan a la vez el botón izquierdo y el derecho4– Se ha pulsado el botón central5– Se pulsan a la vez el botón izquierdo y el central6– Se pulsan a la vez el botón derecho y el central7` – Se pulsan a la vez los 3 botones
cancelable	Boolean	Indica si el evento se puede cancelar
cancelBubble	Boolean	Indica si se ha detenido el flujo de eventos de tipo bubbling
charCode	Número entero	El código unicode del carácter correspondiente a la tecla pulsada
clientX	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
clientY	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
ctrlKey	Boolean	Devuelve true si se ha pulsado la tecla CTRL y false en otro caso
currentTarget	Element	El elemento que es el objetivo del evento
detail	Número entero	El número de veces que se han pulsado los botones del ratón
eventPhase	Número entero	La fase a la que pertenece el evento: 0 – Fase capturing 1 – En el elemento destino 2 – Fase bubbling
isChar	Boolean	Indica si la tecla pulsada corresponde a un carácter
keyCode	Número entero	Indica el código numérico de la tecla pulsada
metaKey	Número entero	Devuelve true si se ha pulsado la tecla META y false en otro caso
pageX	Número entero	Coordenada X de la posición del ratón respecto de la página

pageY	Número entero	Coordenada Y de la posición del ratón respecto de la página
preventDefault()	Función	Se emplea para cancelar la acción predefinida del evento
relatedTarget	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
screenX	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
screenY	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
shiftKey	Boolean	Devuelve true si se ha pulsado la tecla SHIFT y false en otro caso
stopPropagation()	Función	Se emplea para detener el flujo de eventos de tipo bubbling
target	Element	El elemento que origina el evento
timeStamp	Número	La fecha y hora en la que se ha producido el evento
type	Cadena de texto	El nombre del evento

Este objeto, además, tiene un método muy útil que es el **preventDefault(),** el cual permite evitar la acción por defecto de elementos HTML. Un ejemplo de su uso es el siguiente:

```
<details id="details">
    <summary>Más info</summary>
    <div>Información adicional...</div>
</details>
<script>
    const details = document.getElementById('details');
    details.addEventListener('click', function (event){event.preventDefault()})
</script>
```

2.4. Tipos de eventos

La lista completa de eventos que se pueden generar en un navegador se puede dividir en cuatro grandes grupos. La especificación de DOM define los siguientes grupos:

- **Eventos de ratón:** se originan cuando el usuario emplea el ratón para realizar algunas acciones.
- **Eventos de teclado:** se originan cuando el usuario pulsa sobre cualquier tecla de su teclado.
- **Eventos HTML:** se originan cuando se producen cambios en la ventana del navegador o cuando se producen ciertas interacciones entre el cliente y el servidor.
- **Eventos DOM:** se originan cuando se produce un cambio en la estructura DOM de la página.

Eventos de ratón	
Evento	**Descripción**
click	Se produce cuando se pulsa el botón izquierdo del ratón. También se produce cuando el foco de la aplicación está situado en un botón y se pulsa la tecla ENTER
dblclick	Se produce cuando se pulsa dos veces el botón izquierdo del ratón
mousedown	Se produce cuando se pulsa cualquier botón del ratón
mouseout	Se produce cuando el puntero del ratón se encuentra en el interior de un elemento y el usuario mueve el puntero a un lugar fuera de ese elemento
mouseover	Se produce cuando el puntero del ratón se encuentra fuera de un elemento y el usuario mueve el puntero hacia un lugar en el interior del elemento
mouseup	Se produce cuando se suelta cualquier botón del ratón que haya sido pulsado
mousemove	Se produce (de forma continua) cuando el puntero del ratón se encuentra sobre un elemento

Asimismo, empleando el objeto event tenemos acceso a las siguientes propiedades para los eventos de ratón (entre otras):

- Las coordenadas del ratón (todas las coordenadas diferentes relativas a los distintos elementos)
- La propiedad type
- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)

- La propiedad button (sólo en los eventos mousedown, mousemove, mouseout, mouseover y mouseup)

Cabe destacar que cuando se pulsa un botón del ratón, la secuencia de eventos que se produce es la siguiente: mousedown, mouseup y click. Por tanto, la secuencia de eventos necesaria para llegar al doble click llega a ser tan compleja como la siguiente: mousedown, mouseup, click, mousedown, mouseup, click, dblclick.

Eventos de teclado	
Evento	**Descripción**
keydown	Se produce cuando se pulsa cualquier tecla del teclado. También se produce de forma continua si se mantiene pulsada la tecla
keypress	Se produce cuando se pulsa una tecla correspondiente a un carácter alfanumérico (no se tienen en cuenta telas como SHIFT, ALT, etc.). También se produce de forma continua si se mantiene pulsada la tecla
keyup	Se produce cuando se suelta cualquier tecla pulsada

En este caso, el objeto event contiene las siguientes propiedades para los eventos de teclado:

- La propiedad keyCode
- La propiedad charCode (sólo DOM)
- La propiedad srcElement (Internet Explorer) o target (DOM)
- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)

Cuando se pulsa una tecla correspondiente a un carácter alfanumérico, se produce la siguiente secuencia de eventos: keydown, keypress, keyup. Cuando se pulsa otro tipo de tecla, se produce la siguiente secuencia de eventos: keydown, keyup. Si se mantiene pulsada la tecla, en el primer caso se repiten de forma continua los eventos keydown y keypress y en el segundo caso, se repite el evento keydown de forma continua.

Eventos HTML	
Evento	**Descripción**
load	Se produce en el objeto window cuando la página se carga por completo. En el elemento <img> cuando se carga por completo la imagen. En el elemento

unload	Se produce en el objeto window cuando la página desaparece por completo (al cerrar la ventana del navegador por ejemplo). En el elemento <object> cuando desaparece el objeto.
abort	Se produce en un elemento <object> cuando el usuario detiene la descarga del elemento antes de que haya terminado
error	Se produce en el objeto window cuando se produce un error de JavaScript. En el elemento <img> cuando la imagen no se ha podido cargar por completo y en el elemento <object> cuando el elemento no se carga correctamente
select	Se produce cuando se seleccionan varios caracteres de un cuadro de texto (<input> y <textarea>)
change	Se produce cuando un cuadro de texto (<input> y <textarea>) pierde el foco y su contenido ha variado. También se produce cuando varía el valor de un elemento <select>
submit	Se produce cuando se pulsa sobre un botón de tipo submit (<input type="submit">)
reset	Se produce cuando se pulsa sobre un botón de tipo reset (<input type="reset">)
resize	Se produce en el objeto window cuando se redimensiona la ventana del navegador
scroll	Se produce en cualquier elemento que tenga una barra de scroll, cuando el usuario la utiliza. El elemento <body> contiene la barra de scroll de la página completa
focus	Se produce en cualquier elemento (incluido el objeto window) cuando el elemento obtiene el foco
blur	Se produce en cualquier elemento (incluido el objeto window) cuando el elemento pierde el foco

Eventos DOM	
Evento	**Descripción**
DOMSubtreeModified	Se produce cuando se añaden o eliminan nodos en el subárbol de un documento o elemento

DOMNodeInserted	Se produce cuando se añade un nodo como hijo de otro nodo
DOMNodeRemoved	Se produce cuando se elimina un nodo que es hijo de otro nodo
DOMNodeRemovedFromDocument	Se produce cuando se elimina un nodo del documento
DOMNodeInsertedIntoDocument	Se produce cuando se añade un nodo al documento

2.5. Practicando lo aprendido

2.5.1. Práctica guiada: moviendo una esfera

Paso 1: Vamos a practicar la gestión de eventos con un ejemplo sencillo que permitirá al usuario mover una esfera por la pantalla mediante las teclas de dirección del teclado. Para ello, partimos del siguiente código HTML y CSS:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
    body {
      margin: 0;
      overflow: hidden;
    }
    canvas {
      display: block;
      background-color: black;
    }
  </style>
  <title>Moving Sphere</title>
</head>
<body>
<canvas id="myCanvas"></canvas>

<script src="script.js"></script>
</body>
</html>
```

Paso 2: A continuación, creamos el archivo script.js e instanciamos en él una clase para el objeto esfera:

```
class Sphere {
 constructor(x, y, radius, color) {
  this.x = x;
  this.y = y;
  this.radius = radius;
  this.color = color;
 }
}
```

Tal y como ves, esta clase nos permitirá instanciar objetos de tipo esfera en una determinada posición de la pantalla (x,y), con un radio y un color.

Paso 3: Vamos a comenzar a añadir los métodos que permitirán manejar nuestra esfera. Es importante que tengas en cuenta que estos métodos **deben ir dentro de la clase** (esfera en este caso) pero **no** dentro del constructor sino a continuación del mismo. Por ejemplo, añadimos el método **draw()** que dibujará nuestra esfera en un determinado **contexto**:

```
class Sphere {
 constructor(x, y, radius, color) {...}
 //Método para representar dicha esfera
 draw(context) {
  context.beginPath();
  context.arc(this.x, this.y, this.radius, 0, 2 * Math.PI);
  context.fillStyle = this.color;
  context.fill();
  context.closePath();
 }
```

Este método toma un contexto de dibujo (**context**) y luego mediante las instrucciones .beginPath() y closePath() permite representar un objeto. El mismo, será una esfera por lo

que empleamos para dibujar el método .arc(), dándole la posición, el radio de la curva y, como en este caso queremos una esfera cerrada, el recorrido angular a seguir (de 0 a 2π).

Paso 4: Añadimos el método para mover la esfera que posteriormente asignaremos a los eventos y a las pulsaciones del teclado del usuario.

```
move(direction) {
  switch (direction) {
    case 'left': // Verificar que la esfera no se salga del borde izquierdo
      if (this.x - this.radius > 0) {
        this.x -= 5;
      }
      break;
    case 'right': // Verificar que la esfera no se salga del borde derecho
      if (this.x + this.radius < canvas.width) {
        this.x += 5;
      }
      break;
    case 'up': // Verificar que la esfera no se salga del borde superior
      if (this.y - this.radius > 0) {
        this.y -= 5;
      }
      break;
    case 'down': // Verificar que la esfera no se salga del borde inferior
      if (this.y + this.radius < canvas.height) {
        this.y += 5;
      }
      break;
  }
}
```

En este caso hemos añadido también un condicional para que la esfera no pueda abandonar la pantalla (teniendo en cuenta su radio). Si se cumple la condición, actualizamos la posición x o y (según corresponda al switch) en 5 píxeles.

Paso 5: preparamos el canvas para que ocupe la ventana completa del usuario y el contexto de dibujo dentro del canvas (en este caso, el método predefinido getContext('2d') para un renderizado bidimensional). A continuación, creamos un objeto esfera y lo representamos en una posición inicial.

```
// Preparar canvas (ventana completa) y context
const canvas = document.getElementById('myCanvas');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;

const context = canvas.getContext('2d');

// Instanciar una esfera
const sphere = new Sphere(50, 50, 30, 'blue');

// Representar la esfera en su posición inicial
sphere.draw(context);
```

Paso 6: por último, añadimos el evento mediante el **addEventListener()**. En este caso, el evento será que el usuario pulse la tecla. Y a continuación, asignamos una función anónima cuyo parámetro de entrada es el objeto **event**. Después, mediante un switch(), establecemos la condición del movimiento de la esfera en función de la tecla pulsada por el usuario (event.key). Por último, limpiamos el canvas con **clearRect(x, y, width, height)** y repintamos la esfera en su nueva posición:

```
// Manejador de eventos
window.addEventListener('keydown', (event) => {
 switch (event.key) {
  case 'ArrowLeft':
   sphere.move('left');
   break;
  case 'ArrowRight':
   sphere.move('right');
   break;
  case 'ArrowUp':
   sphere.move('up');
   break;
  case 'ArrowDown':
   sphere.move('down');
   break;
 }
 // Limpiar el canvas y pintar la esfera en su nueva posición
 context.clearRect(0, 0, canvas.width, canvas.height);
 sphere.draw(context);
});
```

3. Bibliografía y Webgrafía

[1] "Event Bubbling and Capturing in JavaScript", *www.javatpoint.com*. [En línea]. Disponible en: https://www.javatpoint.com/event-bubbling-and-capturing-in-javascript [Consultado: 12-nov-2023].

[2] J. Padial, "Como utilizar cookies en JavaScript", *CybMeta*, 27-jun-2015. [En línea]. Disponible en: https://cybmeta.com/cookies-en-javascript [Consultado: 12-nov-2023].

[3] "Lenguaje Javascript", *Lenguajejs.com*. [En línea]. Disponible en: https://lenguajejs.com/javascript/ [Consultado: 12-nov-2023].

[4] A. Garro, "JavaScript", *Arkaitzgarro.com*, 01-ago-2014. [En línea]. Disponible en: https://www.arkaitzgarro.com/javascript/index.html [Consultado: 12-nov-2023].

[5] "Introducción a JavaScript", Uniwebsidad.com. [En línea]. Disponible en: https://uniwebsidad.com/libros/javascript [Consultado: 14-nov-2023].