

# GESTIÓN DE EVENTOS

## OBJETIVOS

- Reconocer las ventajas y aplicaciones del control de eventos en JavaScript
- Identificar los principales métodos de captura de eventos
- Asimilar el funcionamiento de los eventos y su propagación por los elementos del DOM
- Crear aplicaciones capaces de lanzar código asociado a eventos
- Reconocer las aplicaciones del objeto de evento y sus principales métodos y propiedades
- Diferenciar los principales tipos de eventos identificando sus acciones principales
- Aplicar el uso eventos a la interacción con el usuario utilizando formularios
- Resolver problemas de aplicaciones web completas utilizando control de eventos

## CONTENIDOS

- 8.1 INTRODUCCIÓN A LOS EVENTOS**
- 8.2 CAPTURA DE EVENTOS**
  - 8.2.1 MÉTODOS CLÁSICOS Y OBSOLETOS
  - 8.2.2 MÉTODO ACONSEJABLE
  - 8.2.3 PROPAGACIÓN DE EVENTOS
  - 8.2.4 ANULAR EVENTOS
  - 8.2.5 CAPTURA DE EVENTOS EN ELEMENTOS DINÁMICOS
- 8.3 OBJETO DE EVENTO**
  - 8.3.1 UTILIDAD Y USO DEL OBJETO DE EVENTO
  - 8.3.2 OBTENER COORDENADAS DEL EVENTO
  - 8.3.3 OBTENER LA TECLA PULSADA
  - 8.3.4 ANULAR COMPORTAMIENTOS PREDETERMINADOS
  - 8.3.5 CANCELAR PROPAGACIÓN
  - 8.3.6 LANZAR EVENTOS
- 8.4 LISTA DE TIPOS DE EVENTOS**
  - 8.4.1 EVENTOS DE RATÓN
  - 8.4.2 EVENTOS DE TECLADO
  - 8.4.3 EVENTOS DE MOVIMIENTO EN LA VENTANA
  - 8.4.4 EVENTOS SOBRE CARGA Y DESCARGA DE ELEMENTOS
  - 8.4.5 OTROS EVENTOS
- 8.5 FORMULARIOS**
  - 8.5.1 EL FORMULARIO COMO OBJETO DEL DOM
  - 8.5.2 EVENTOS DE FORMULARIO
  - 8.5.3 PROPIEDADES DE LOS CONTROLES
  - 8.5.4 MÉTODOS DE LOS CONTROLES
- 8.6 PRÁCTICAS SOLUCIONADAS**
- 8.7 PRÁCTICAS RECOMENDADAS**
- 8.8 RESUMEN DE LA UNIDAD**
- 8.9 TEST DE REPASO**

&lt;

y

## 8.1 INTRODUCCIÓN A LOS EVENTOS

Gracias a esta unidad, tendremos ya todas las armas que permiten crear aplicaciones en el lado del cliente. Con esta unidad hemos llegado al colofón de la programación de aplicaciones web del lado del cliente. Los elementos fundamentales que aporta el lenguaje JavaScript para construir aplicaciones web en el lado del cliente estarían ya completos.

Con esta unidad podremos hacer casi cualquier tipo de aplicación web front-end (salvo las que requieran interacción con servidores). Los conceptos son sencillos, si esta unidad es larga se debe más a la cantidad de eventos y propiedades que aporta JavaScript y a la cantidad de prácticas que se proponen, y que se recomienda realizar encarecidamente, ya que el manejo de los eventos requiere mucha práctica para poder asimilar sus capacidades.

Los eventos son el mecanismo fundamental para comunicar la aplicación con el usuario. En esta interacción, la aplicación variará su funcionamiento en razón de las acciones realizadas por el usuario o usuaria.

Un evento no es más que un suceso, algo que ha ocurrido como resultado de un acto del usuario o por otras razones. Para que se considere realmente un evento, la aplicación tiene que ser capaz de detectarlo y, lo más importante, tiene que ser capaz de ejecutar un código asociado a dicho evento.

Los eventos se asocian a un elemento concreto del DOM, de modo que, por ejemplo, podemos hacer una cosa cuando el usuario haga clic sobre un párrafo y otra cuando el clic sea sobre otro párrafo.

La clave de los eventos en JavaScript es su capacidad asíncrona. Podemos estar esperando que el usuario haga clic en un botón, pero no se puede parar el resto del código por esa espera. La espera se produce en segundo plano, es un proceso especial llamado **listener** (escuchante) el que está esperando a recibir la notificación del clic del usuario. La gestión de estos listeners la hace el navegador.

Por lo tanto, hay un proceso que automatiza la captura, de modo que cuando se produce un evento (por ejemplo, un clic sobre un elemento de la página), se ejecutará el código de una función callback. Lo único que hay que hacer es preparar la escucha del evento y *esta se* produce en segundo plano.

## 8.2 CAPTURA DE EVENTOS

### 8.2.1 MÉTODOS CLÁSICOS Y OBSOLETOS

#### 8.2.1.1 ATRIBUTOS HTML ASOCIADOS A EVENTOS

Hace años, en las primeras versiones de JavaScript, la captura de un evento se realizaba desde el propio código HTML. Todavía se puede hacer porque se ha mantenido esta forma de captura por cuestiones de compatibilidad, pero no es nada aconsejable, ya que complica enormemente el mantenimiento del código al estar mezclado HTML y JavaScript de forma confusa.

Este primer método utiliza atributos en las etiquetas de los elementos de la página que comienzan con la palabra ***on*** seguida del nombre del evento. Por ejemplo: ***onclick***. Como valor del atributo se indica el nombre de la función que contiene el código a ejecutar. Ejemplo:

```
<p id="parrafol" onclick="alert('Me has hecho click')">
Soy 'clickable'</p>
<p id="parrafo2">Yo no</p>
```

EL primer párrafo usa el atributo onclick que permite lanzar código JavaScript cuando el usuario haga clic sobre el párrafo. El segundo párrafo no reacciona a la pulsación, porque no se ha indicado evento alguno.

### 8.2.1.2 PROPIEDADES DE LOS ELEMENTOS ASOCIADOS A EVENTOS

Es una mejora del método anterior, que permite que el código JavaScript se independice de HTML, lo que mejora su mantenimiento. Se trata de usar propiedades que tienen el mismo nombre que los atributos HTML: la palabra ***on*** seguida del nombre del evento (***onclick***, ***onmouseover***, etc). Ha sido el método más utilizado durante muchos años, pero actualmente no se recomienda su uso.

Un ejemplo de funcionamiento sería el siguiente:

```
<p id="parrafol">Soy 'clickable'</p>
<p id="parrafo2">Yo no</p>
...
<script>
let parrafol=document.getElementById("parrafol");
parrafol.onclick=()=>{alert('Me has hecho click')};
</script>
```

Se ha asignado como función callback una función flecha que muestra el mismo mensaje de tipo ***alert*** que en el ejemplo anterior.

### 8.2.2 MÉTODO ACONSEJABLE

El método que se aconseja actualmente es utilizar el método ***addEventListener*** que está disponible en todos los elementos. A este método hay que indicarle el nombre del evento y la función callback a ejecutar. Es el método que se debe utilizar según la norma actual. Todos los navegadores actuales lo reconocen (las versiones de Internet Explorer anteriores a la 9, no lo hacen) y tiene una ventaja técnica sobre los dos métodos anteriores: se puede asignar más de una función al mismo evento.

El código equivalente de este método, equivalente a los usados con los métodos anteriores es:

```
<p id="parrafol">Soy 'clickable'</p>
<p id="parrafo2">Yo no</p>
...
<script>
```

```
let parrafol=document.getElementById("parrafol");
parrafol.addEventListener("click",()=>{alert('Me has hecho click')});
</script>
```

Para asignar varias acciones, basta con invocar otra vez a la función **addEventListener** con el mismo tipo de evento y usando otra función callback. Cuando se produzca el evento, el código de todas las funciones que se han asociado al evento, se ejecutará. En los métodos explicados anteriormente, solo se puede asignar una función a cada evento.

### 8.2.3 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 click? ¿Y la capa?

Veamos lo que ocurre con esta página completa:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" >
    <title>Propagación de eventos</title>
    <style>
        div{
            position:fixed;
            left:0;
            top:0;
            height:50%;
            width:100%;
            background-color:gray;
        }
        p{
            height:50%;
            margin:0;
            background-color:lightgray;
        }
    </style>
</head>
<body>
    <div>
        <p>
            <button> ¡Hazme clic!</button>
        </p>
    </div>
<script>
let capa=document.querySelector("div");
```

```

let p=document.querySelector("p");
let boton=document.querySelector("button");
capa.addEventListener("click",()=>{consolé.log("click en capa")});
p.addEventListener("eclick",()=>{consolé.log("eclick en p")});
boton.addEventListener("eclick",()=>{consolé.log("eclick en boton")});
</script>
</body>
</html>

```

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

```

click en boton
click en p
click en capa

```

El evento se propaga desde el botón (elemento más interior) hasta el elemento de tipo **div** (elemento más exterior, elemento más cercano a la raíz del DOM).

El proceso de captura de eventos se describe en esta imagen:

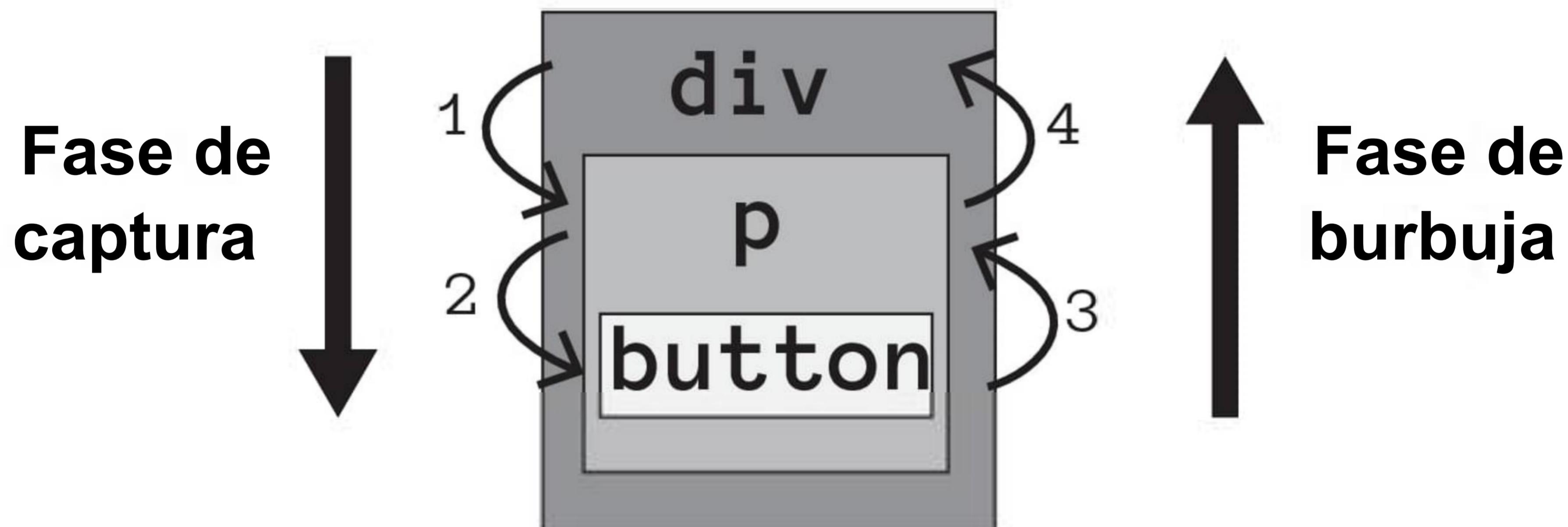


Figura8.1: Propagación de los eventos

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 (no disponible en versiones de Internet Explorer anteriores a la 9) gracias a que, en realidad, el método **addEventListener** tiene un tercer parámetro relacionado con el tipo de captura. Se trata de un valor booleano que tiene estas posibilidades:

- **true**. El código de captura se lanza en la fase de captura.
- **false**. Es el valor por defecto. El código se lanza en la fase de burbuja.

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

```
let capa=document.querySelector("div");
let p=document.querySelector("p");
let boton=document.querySelector("button");
capa.addEventListener("click",
    ()=>{consolé.log("elick en capa")},true);
p.addEventListener("click",()=>{consolé.log("elick en p")},true);
boton.addEventListener("click",
    ()=>{consolé.log("elick en boton")},true);
```

Si ejecutamos el código, por consola tendremos el siguiente resultado:

```
click en capa
click en p
click en boton
```

## 8.2.4 ANULAR EVENTOS

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

```
<p>Hazme clic!</P>
<script>
    function mensaje(){
        alert("!Me has hecho clic!")
        p.removeEventListener("click",mensaje);
    }

    let p=document.querySelector("p");
    p.addEventListener("click",mensaje);
</script>
```

Dentro de la propia función **mensaje**, tras escribir el mensaje en sí, se invoca al método **removeEventListener** el cual provoca que no se vuelva a invocar a esta función tras otro clic.

Hay que tener en cuenta que solo podemos retirar funciones de captura que tengan nombre, no podemos anular funciones anónimas asociadas a eventos.

## 8.2.5 CAPTURA DE EVENTOS EN ELEMENTOS DINÁMICOS

A la hora de implementar capturas de eventos, hay que tener en cuenta que los listeners se asocian a elementos del DOM que existan en ese momento. Eso puede dar lugar a problemas si no se tiene cuidado.

Supongamos que queremos crear un documento en el que consigamos que al hacer clic en todos los párrafos de tipo p, se muestre en un cuadro de alerta, el texto del párrafo. El código podría ser este:

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>
<p>Cuatro</p>
<script>
let parrafos=document.querySelectorAll("p");
for(let párrafo of parrafos){
    párrafo.addEventListener("click",()=>{alert(párrafo.textContent)});
}

```

La página posee cuatro párrafos, con los textos **Uno, Dos, Tres** y **Cuatro**. El código JavaScript asocia todos los párrafos a la variable que se llama **párrafos**. Pero, para implementar el control de eventos, necesitamos asignar un evento a cada párrafo. Por eso hay un bucle que recorre cada elemento devuelto por el método **querySelectorAll** y así vamos asignando a cada párrafo la captura del evento. Si ejecutamos este código, veremos que, efectivamente, al hacer clic en cualquiera de los cuatro párrafos, se nos muestra el mensaje.

Sin embargo, si añadimos un quinto párrafo de tipo p mediante JavaScript:

```
let nuevoParrafo=document.createElement("p");
nuevoParrafo.textContent="Cinco";
document.body.appendChild(nuevoParrafo);
```

Este párrafo aparecerá al final de los otros cuatro. Al hacer clic en él, no ocurrirá nada. La razón es lógica. Se ha creado después del bucle y por ello no se ha ejecutado el método **addEventListener** sobre él. Tendríamos que añadir el código que permita capturar el evento en el nuevo elemento.

## 8.3 OBJETO DE EVENTO

### 8.3.1 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 de 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 causó el evento.

Veamos un ejemplo usando esta propiedad para entender cómo funciona el objeto de evento:

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>

<script>
    function escribeContenido(evento){
        consolé.log(evento.target.textContent);
    }
    let parrafos=document.querySelectorAll("p");
    for(let párrafo of parrafos){
        párrafo.addEventListener("click",escribeContenido);
    }
</script>
```

En este código se asigna a los elementos de los tres párrafos, el código de la función *escribeContenido* será ejecutado cuando se produzca un clic en cada párrafo. La función recibe un parámetro en el que se almacenará el objeto con los datos importantes del evento. La función, en este caso, usa la propiedad **target** para llegar al elemento que causó el clic y así acceder al texto de ese elemento que, finalmente, se mostrará en la consola.

### 8.3.2 OBTENER COORDENADAS DEL EVENTO

Los objetos de 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 píxeles 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. Coincidirán muchas veces, pero si la ventana no está maximizada, no coincidirán.

---

#### ACTIVIDAD 8.1: CARTEL PERSIGUIENDO AL RATÓN

- Es bueno que vayamos practicando eventos para solucionar tareas y así coger bagaje y habilidad en el uso de eventos y la aplicación de sus propiedades.
  - Aprovechando estas propiedades, haz la Práctica 8.1 en la página 322, que consigue que un cartel con el texto **Hola**, persiga el cursor del usuario.
- 

Otras coordenadas que se almacenan en el objeto de evento son **pageX** y **pageY**. Funcionan como **clientX** y **clientY**, pero no solo toman las de la ventana, tienen en cuenta el desplazamiento realizado en el elemento. Es decir, si hemos avanzado dos pantallas de 500 píxeles de alto usando las barras de desplazamiento y el cursor está a mitad de pantalla, **clientY** nos diría 250, mientras que **pageY** nos diría 1250. Depende de lo que deseemos nos vendrán mejor unas u otras coordenadas.

## ACTIVIDAD 8.2: COMPARAR COORDENADAS

- Crea una página con mucho contenido. Puedes hacerlo rápido escribiendo en Visual Code, dentro del apartado body de la aplicación, la abreviatura `p*60>lorem500`. Tras pulsar el tabulador llenaremos la página de contenido.
- Escribe este código JavaScript justo antes de cerrar la etiqueta body:

```
<script>
document.addEventListener("mousemove",function(ev){
    consolé.log('screenX:${ev.screenX}, screenY:${ev.screenY}\n' +
        'clientX:${ev.clientX}, clientY:${ev.clientY}\n' +
        'pageX:${ev.pageX}, pageY:${ev.pageY}');
})
</script>
```

- Haz que la ventana donde se ejecuta la página no esté maximizada. Desplázate hacia abajo en el documento varias pantallas y ahora, mostrando la consola, compara las coordenadas. Verás como `pageY` es la más grande porque cuenta todo el desplazamiento, como `screenY` es la segunda porque toma la pantalla y que `clientY` es la más pequeña.

### 8.3.3 OBTENER LA TECLA PULSADA

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

La propiedad más importante es `key` que obtiene el texto que indica la tecla pulsada. Es el más cómodo de programar porque es fácil de entender. Así sus valores fundamentales devueltos son:

- Si es una tecla de carácter, nos retorna el carácter. Por ejemplo: `"a"`, `"g"`, `"k"`, `"ñ"`, etc. También nos indica si hay mayúsculas porque la letra la retornaría mayúscula: `"A"`, `"G"`, `"K"`, `"Ñ"`, etc.
- Con las teclas numéricas actúa igual, nos retorna el número. Pero con shift pulsada a la vez, nos devuelve el segundo símbolo de la letra (&lt;&gt;, \$, %, etc.). Con Alt Gr nos retorna el tercero (|, &lt;&gt;, #, etc.).
- Actúa igual con cualquier tecla que escriba caracteres, simplemente indica qué carácter normalmente sale escrito.
- En el caso de pulsar, sin más, teclas de control, nos devolverá su nombre: `"Control"`, `"Alt"`, `"Shift"`, `"AltGraph"`, `"F5"`, `"F6"`, `"KeyUp"`, `"KeyDown"`, `"Home"`, `"End"`, etc.

A veces es necesario saber si se pulsó a la vez que la tecla, 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`, `ShiftKey` y `MetaKey` (tecla de los ordenadores Mac).

### ACTIVIDAD 8.3: PRÁCTICA DE CAPTURA DE TECLAS

- Haz la Práctica 8.2 en la página 323 que permite practicar y entender cómo hacer captura de teclas.

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. La propiedad **location** lo consigue, esta propiedad puede devolver los siguientes valores:

Devuelve un número entero que indica la posición del teclado en la que se encuentra la tecla. Valores posibles que devuelve:

VALOR	CONSTANTE RELACIONADA	SIGNIFICADO
0	<code>DOM_KEY_LOCATION_STANDARD</code>	Teclado normal
1	<code>DOM_KEY_LOCATION_LEFT</code>	Zona izquierda (para distinguir las dos teclas control por ejemplo)
2	<code>DOM_KEY_LOCATION_RIGHT</code>	Zona derecha (para distinguir las dos teclas control por ejemplo)
3	<code>DOM_KEY_LOCATION_NUMPAD</code>	Teclado numérico

#### 8.3.3.1 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 valores posibles para la propiedad **button** son:

VALOR	SIGNIFICADO
0	Botón principal del ratón.
1	Botón central del ratón (en muchos casos, el botón de la rueda).
2	Botón secundario del ratón (en el caso de personas diestras, es el botón derecho).
3	Cuarto botón. En muchos ratones es el de retroceder página.
4	Quinto botón. En muchos ratones es el de avanzar página.

### 8.3.4 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 la idea 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 lo confirma, no nos moveremos de la página actual. El código podría ser este:

```
<a href="Ir a la Wikipedia">
```

```
<script>
  let enlace=document.querySelector("a");
  enlace.addEventListener("click",function(ev){
    if(confirm("¿Realmente desea abandonar esta página?"))
      location="https://es.wikipedia.org/";
  });
</script>
```

Si ejecutamos este código en una página web, efectivamente se nos requiere confirmar la acción, pero aunque no aceptemos el cuadro, la acción se lleva a cabo, porque los enlaces, por defecto envían al usuario al destino. Por ello el código debería ser este:

```
<a href="Ir a la Wikipedia">
```

```
<script>
  let enlace=document.querySelector("a");
  enlace.addEventListener("click",function(ev){
    if(!confirm("¿Realmente desea abandonar esta página?"))
      ev.preventDefault();
  });
</script>
```

### 8.3.5 CANCELAR PROPAGACIÓN

Anteriormente (8.2.3 "*Propagación de eventos*", en la página 302 ) hemos explicado cómo funciona la propagación de eventos. Hemos visto que los eventos se lanzan primero en el elemento más interior, luego en el siguiente y así, hasta que al final se lanzan los relacionados con los elementos más exteriores.

Podemos cancelar la propagación mediante un método llamado **stopPropagation**. Veamos este ejemplo:

```
<div>
  Pinta de rojo
  <button>Pinta de verde</button>
</div>
```

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

El botón está dentro de un elemento de tipo div. Hacer clic en el elemento div (con el texto **Pinta de rojo**) provoca colorear todo el cuerpo de la página en color 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 pudiéramos ver a cámara lenta el proceso, veríamos 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 podemos usar el método **stopPropagation**:

```
<div>
    Pinta de rojo
    <button>Pinta de verde</button>
</div>
<script>
    let boton=document.querySelector("button");
    let div=document.querySelector("div");
    boton.addEventListener("click",(ev)=>{
        document.body.style.backgroundColor="green";
        ev.stopPropagation();
    });
    div.addEventListener("click",(ev)=>{
        document.body.style.backgroundColor="red";
    });
</script>
```

Hay que observar que hemos indicado el fin de la propagación en el elemento div. De esa forma, este elemento se retira del lanzamiento del evento en la fase de burbuja y así solo se ejecuta el código en el clic del botón.

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

### 8.3.6 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 **MouseEvent** por ejemplo) el cual funciona indicando el tipo de evento en la creación. Por ejemplo:

```
let evento=new Event("elick");
```

Después lanzaríamos el evento a cualquier elemento que esté capturando ese evento. Un ejemplo de uso (poco práctico, pero fácil de entender) sería el que proporciona este código:

```
<P>
    <button id="boton1">Coloreo de rojo</button>
    <button id="boton2">Yo hago lo mismo</button>
</p>
<script>
let boton1=document.getElementById("boton1");
let boton2=document.getElementById("boton2");
boton1.addEventListener("click",()=>{
    document.body.style.backgroundColor="red";
});
boton2.addEventListener("click",()=>{
    let el=new Event("elick");
    boton1.dispatchEvent(el);
});
</script>
```

Aparecerán dos botones, ambos realizan la misma acción, porque el evento **onclick** del segundo botón realiza un clic sobre el primero.

## 8.4 LISTA DE TIPOS DE EVENTOS

En todos los ejemplos anteriores hemos utilizado el evento **click**. Sin embargo, hay muchos otros eventos que podemos capturar. De detallan los principales a continuación.

### 8.4.1 EVENTOS DE RATÓN

Los eventos de ratón los produce este dispositivo. Pero también los dispositivos táctiles pueden producir algunos de estos eventos. Por ejemplo, el evento **click** ocurre cuando el usuario pulsa y despulsa el botón principal del ratón sobre el objeto que posee el **listener**, pero también ocurre si el usuario, en un dispositivo táctil, golpea y suelta el dedo sobre dicho objeto.

Por otro lado, aunque los eventos se refieren al ratón, realmente valen para cualquier dispositivo apuntador capaz de mover el cursor por la pantalla.

La tabla completa de los eventos de ratón es la siguiente:

EVENTO	EXPLICACIÓN
<b>click</b>	El usuario hace clic sobre el botón principal del dispositivo apuntador. Esta acción exactamente consiste en bajar y subir rápidamente el botón principal del dispositivo apuntador (o dar un golpe con el dedo en un dispositivo táctil)
<b>dblclick</b>	El usuario golpea dos veces rápidas sobre el botón principal del dispositivo apuntador (o con el dedo en un dispositivo táctil)
<b>mousedown</b>	El usuario pulsa un botón del dispositivo apuntador. Se produce este evento justo cuando el usuario pulsa el botón, antes de que le suelte.
<b>mouseup</b>	Se produce cuando el usuario suelta el botón del dispositivo apuntador que había pulsado previamente.
<b>mousenter</b>	Se produce cuando el usuario mueve el cursor del dispositivo apuntador dentro del elemento que captura el evento. Solo se produce si el cursor estaba fuera del elemento y el usuario le acaba de mover dentro.
<b>mouseleave</b>	Se produce cuando el usuario mueve el cursor del dispositivo apuntador fuera del elemento que captura el evento. Solo se produce si el cursor estaba dentro del elemento y el usuario le acaba de mover fuera.
<b>mousemove</b>	Se produce cada vez que el usuario mueve el cursor dentro del elemento (estando previamente dentro).
<b>mouseover</b>	Se produce cuando el cursor apuntador entra dentro del elemento o de cualquiera de los hijos del elemento.
<b>mouseout</b>	Se produce cuando el cursor apuntador abandona el elemento o cualquiera de los hijos del elemento.
<b>contextmenu</b>	Se produce cuando el usuario acaba de pedir el menú de contexto. Normalmente ese menú aparece al pulsar el botón secundario del dispositivo apuntador. En dispositivos táctiles suele ocurrir cuando el usuario deja el dedo apretado sobre el elemento 2 o 3 segundos al menos.

#### ACTIVIDAD 8.4: CAPTURA DE EVENTOS DE RATÓN

- Es muy interesante hacer la Práctica 8.3 en la página 325, para comprender la mayoría de eventos de ratón y cómo distinguir los botones en la práctica.

#### 8.4.2 EVENTOS DE TECLADO

EVENTO	EXPLICACIÓN
<b>keypress</b>	Se produce cuando un usuario pulsa y suelta una tecla.
<b>keydown</b>	Se produce justo cuando el usuario ha pulsado la tecla (antes de que la suelte).
<b>keyup</b>	Se produce cuando el usuario suelta la tecla.

### ACTIVIDAD 8.5: CAPTURA DE EVENTOS DE TECLADO

- La Práctica 8.4 en la página 327, facilita mucho el aprendizaje para capturar eventos de teclado. Es muy recomendable hacerla.

### 8.4.3 EVENTOS DE MOVIMIENTO EN LA VENTANA

EVENTO	EXPLICACIÓN
<b>scroll</b>	Ocurre cuando se ha desplazado la ventana a través de las barras de desplazamiento o usando el dispositivo táctil.
<b>resize</b>	Evento de window que se produce cuando se cambia el tamaño de la ventana.

### ACTIVIDAD 8.6: CAPTURA DE SCROLL

- Realizar la Práctica 8.5 en la página 328 para practicar con el evento scroll y las propiedades relacionadas.

### 8.4.4 EVENTOS SOBRE CARGA Y DESCARGA DE ELEMENTOS

EVENTO	EXPLICACIÓN
<b>load</b>	Se concluyó la carga del elemento. Es uno de los elementos más importantes a capturar para asegurar que el código siguiente funciona con la seguridad de que está cargado lo que necesitamos.  Cuando se aplica al elemento <b>window</b> , se produce cuando todos los elementos del documento se han cargado.
<b>DOMContentLoaded</b>	Similar al evento <b>load</b> . Se produce cuando el documento HTML ha sido cargado. A diferencia de <b>load</b> , se dispara sin esperar a que se terminen de cargar las hojas de estilos, imágenes y elementos en segundo plano.  En general, para JavaScript, es más conveniente este evento.
<b>abort</b>	Se produce cuando se anula la carga de un elemento.
<b>error</b>	Sucede si hubo un error en la carga.
<b>progress</b>	Se produce si la carga está en proceso.
<b>readystatechange</b>	Ocurre cuando se ha modificado el estado del atributo readyState, lo cual ocurre cuando se ha modificado el estado de carga y descarga.

### ACTIVIDAD 8.7: PANEL DE CARGA

- La Práctica 8.6 en la página 330, es muy interesante para manejar el evento **load**. Se recomienda encarecidamente hacerla.

Concretamente, el evento **load** del objeto **window** es muy importante. Hay que tener en cuenta que desde JavaScript manejamos elementos del DOM continuamente. Además, JavaScript es un lenguaje asíncrono, como se ha comentado en este libro varias veces, y eso puede implicar que el código JavaScript intente manipular un componente que aun no se ha cargado.

Por ello es muy habitual que todo el código JavaScript para manipular elementos del DOM, se coloque dentro de la función callback asociada al evento **load** del objeto **window**.

## 8.4.5 OTROS EVENTOS

Deliberadamente hemos obviado los eventos relacionados con los formularios. Debido a la importancia de los formularios en la creación de aplicaciones, hemos dedicado un apartado completo al final de esta unidad.

Pero aun hay más eventos. Los eventos de esta sección son más específicos, no son de uso habitual, pero nos pueden ser de utilidad en algunas aplicaciones:

### 8.4.5.1 EVENTOS SOBRE EL HISTORIAL

EVENTO	EXPLICACIÓN
<b>popstate</b>	Se produce si se cambia el historial.

### 8.4.5.2 EVENTOS RELACIONADOS CON LA REPRODUCCIÓN DE MEDIOS

EVENTO	EXPLICACIÓN
<b>waiting</b>	Sucede cuando el vídeo se ha detenido por falta de datos.
<b>playing</b>	El medio está listo para su reproducción después de que se detuviera por falta de datos u otras causas.
<b>canplay</b>	Ocurre cuando se detecta que un vídeo (u otro elemento multimedia) ya se puede reproducir (aunque no se haya cargado del todo).
<b>canplaythrough</b>	Se produce si se estima que el vídeo ha cargado suficientes datos para poderse reproducir sin tener que esperar la llegada de más datos.
<b>pause</b>	El medio de reproducción se ha pausado.
<b>play</b>	Ocurre cuando el medio de reproducción se ha empezado a reproducir tras una pausa.
<b>ended</b>	Se produce si la reproducción del vídeo o audio se ha detenido, sea por haber llegado al final o porque no hay más datos disponibles.
<b>loadeddata</b>	Se produce si se ha cargado el frame actual (normalmente el primero).
<b>suspend</b>	Se lanza si se ha suspendido la carga del medio.
<b>emptied</b>	Sucede si se ha vaciado el medio por un nuevo intento de carga por parte del usuario o por otras razones.
<b>stalled</b>	Sucede ante un fallo en la carga, pero sin que se detenga la misma.

EVENTO	EXPLICACIÓN
<b>seeking</b>	Ocurre cuando se ha iniciado una labor de búsqueda en el medio.
<b>seeked</b>	Ocurre cuando se ha finalizado la labor de búsqueda.
<b>loadedmetadata</b>	Se han cargado los metadatos del medio
<b>durationchange</b>	Sucede si se modifica el atributo duration del medio.
<b>timeupdate</b>	Ocurre si se ha modificado el atributo currentTime del medio.
<b>ratechange</b>	Se produce cuando el ratio del vídeo se ha modificado.
<b>volumechange</b>	Se lanza si el volumen se ha modificado.

#### 8.4.53 EVENTOS DE ARRASTRE

Están relacionados con la interfaz de arrastre que es parte de HTML 5. Para que un elemento pueda ser arrastrable debe tener el atributo **draggable** colocado con valor **true**:

```
<div id="capaArrastrable" draggable="true"> ¡Soy arrastrable!</div>
```

En este tipo de operaciones hay dos capas protagonistas: una capa arrastrable (la que realmente se arrastra) y un posible destino del arrastre.

La capa que se arrastra puede generar estos eventos:

EVENTO	EXPLICACIÓN
<b>dragstart</b>	Se produce cuando el usuario empieza a arrastrar el elemento.
<b>drag</b>	Ocurre, una vez iniciado el arrastre, cada vez que se sigue arrastrando el elemento (cada vez que lo movemos).
<b>dragstop</b>	Se produce cuando el arrastre finaliza.

Eventos que se produce en el elemento destino del arrastre.

EVENTO	EXPLICACIÓN
<b>dragenter</b>	Se produce cuando el elemento que se está arrastrando, entra en el elemento destino.
<b>dragover</b>	Ocurre cada vez que se continúa, tras haber entrado, arrastrando el elemento origen sobre el destino.
<b>dragleave</b>	Ocurre cuando el elemento que se arrastra, sale del destino.
<b>drop</b>	Ocurre cuando el elemento origen se suelta dentro del destino. Para que este evento se pueda capturar hay que eliminar el comportamiento por defecto del evento <b>dragover</b> .

#### ACTIVIDAD 8.8: CAPTURA DE ARRASTRE

- En la Práctica 8.7 en la página 332, podemos observar cómo utilizar estos eventos para poder arrastrar una capa sobre otra.

#### 8.4.5.4 EVENTOS SOBRE ANIMACIONES Y TRANSICIONES

Son eventos que podemos capturar cuando hemos programado animaciones sobre uno o más elementos desde CSS o por transiciones. Tenemos estos eventos posibles a capturar:

EVENTO	EXPLICACIÓN
<b>animationstart</b>	Se produce cuando se inicia una animación sobre el elemento.
<b>animationinteraction</b>	Se produce justo cuando se repite la animación.
<b>animationend</b>	Se produce cuando finaliza la animación.
<b>transitionrun</b>	Se lanza cuando ya se ha preparado para empezar la transición.
<b>transitionstart</b>	Ocurre cuando se inicia una transición.
<b>transitionend</b>	Ocurre al finalizar la transición.

#### 8.4.5.5 EVENTOS DEL PORTAPELES

Permiten capturar los eventos de cortar, copiar y pegar. Todos ellos se relacionan con el objeto **clipboard** que es el que permite gestionar el portapapeles del sistema.

EVENTO	EXPLICACIÓN
<b>cut</b>	Se produce cuando el usuario intenta cortar contenido del elemento.
<b>copy</b>	Se produce cuando el usuario intenta copiar contenido del elemento.
<b>paste</b>	Se produce cuando el usuario intenta pegar contenido.

#### 8.4.5.Ó EVENTOS ESPECIALES

EVENTO	EXPLICACIÓN
<b>offline</b>	Solo funciona para el objeto window y se produce si el navegador se desconecta de la red.
<b>online</b>	Solo funciona para window y se produce si el navegador vuelve a conectarse a la red después de haber estado desconectado.
<b>fullscreenchange</b>	Ocurre cuando un elemento pasa a modo de pantalla completa.
<b>fullscreenerror</b>	Sucede si hay un error al pasar un elemento a modo de pantalla completa.
<b>message</b>	Evento que se asocia a numerosos elementos de envío de mensajes como los que se producen a través de las APIs <b>WebSockets</b> , <b>WebWorkers</b> y otras.

### 8.5 FORMULARIOS

#### 8.5.1 EL FORMULARIO COMO OBJETO DEL DOM

Indudablemente, los formularios son el componente fundamental de captura de eventos en una aplicación web. Nos han sido muy útiles los métodos **alert**, **prompt** y **confirm**, pero la realidad es que las aplicaciones profesionales no los utilizan nunca. 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 siempre 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 del documento. Pero es posible, y más recomendable de hecho, acceder al formulario mediante otros métodos, por ejemplo, por su identificador.

Acceder a los elementos del formulario se hacía, de forma clásica, mediante el atributo **name** de los controles del formulario. 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. Así, si tenemos un control para grabar el primer apellido del usuario y ese control utiliza el atributo **name** para darle el nombre de **apellido1**, la forma de acceder a ese control es (suponemos que estamos en el primer formulario del documento):

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

Igualmente, podríamos hacerlo así:

```
document.forms[0]["apellido1";
```

Y, por supuesto, esta otra forma es válida:

```
consolé.log(document.querySelector("[name='apellido1']));
```

Si hay varios elementos con el mismo nombre (como ocurre con los botones de radio), esta última forma de acceder debe utilizar **querySelectorAll**, la cual devolverá una colección con todos los botones de radio con ese nombre. Después, tendremos que ir recorriendo cada botón de la colección.

## RECOMENDACIÓN

- Damos por hecho que se conocen los entresijos de los formularios HTML tales como: tipos de controles, envío de datos por GET y por POST, atributos de los controles (especialmente **value**, **name** e **id**), botones **submit** y **reset**, etc. De no ser así, se aconseja un repaso de estos conceptos para aprovechar mejor este apartado.

El objeto de formulario (**form**) posee estas propiedades y métodos:

PROPIEDAD O MÉTODO	USO
<b>elements</b>	Devuelve una colección que contiene todos los controles del formulario.
<b>length</b>	Obtiene el número de controles del formulario.
<b>action</b>	Devuelve el contenido de la propiedad <b>action</b> , que marca la URL destino de los datos del formulario. Permite su modificación.

PROPIEDAD O MÉTODO	USO
<b>method</b>	Retorna el contenido de la propiedad method del formulario, que marca la forma de envío de los datos (get o post). Permite su modificación.
<b>enctype</b>	Obtiene o cambia la forma de codificar los datos del formulario.
<b>acceptCharset</b>	Obtiene o modifica el conjunto de caracteres del formulario.
<b>submit()</b>	Envía los datos del formulario a su destino.
<b>reset()</b>	Deja los valores de los controles del formulario a su estado por defecto.

Esos métodos van a permitir automatizar acciones de forma muy efectiva en los formularios.

## 8.5.2 EVENTOS DE FORMULARIO

### 8.5.2.1 EVENTO SUBMIT

Se trata del evento que produce el envío de los datos del formulario a su destino. Es muy habitual validar los datos previamente a su envío para que el usuario detecte los fallos antes de realizar el envío. Este evento se produce justo antes del envío y se asocia al propio formulario (también se puede capturar en el botón de tipo **submit** del formulario). Tras su captura podemos validar los datos, pero teniendo la precaución de que si no cancelamos la acción por defecto de este evento (mediante el **preventDefault** del objeto de evento), la acción se llevará a cabo.

Como ejemplo, supongamos que tenemos un formulario en el que pedimos al usuario su nombre de usuario y que dicho nombre solo pueden ser letras del alfabeto inglés y con un mínimo de 6 caracteres. No enviaremos datos si no se cumple esa premisa y cuando detectemos el fallo, avisaremos del error y no permitiremos el envío. El código podría ser éste:

```
<form action="https://jorgesanchez.net/servicios/ver-datos.php">
    <input type="text" name="usuario" id="usuario"><br>
    <button type="submit">Enviar</button>
</form>
<div id="mensaje"> </div>
<script>
let formulario=document.forms[0];
let tUsuario=document.getElementById("usuario");
let cMensaje=document.getElementById("mensaje");

formulario.addEventListener("submit",function(ev){
    let exp=/^a-zA-Z]{6,}$/;
    if(exp.test(tUsuario.value)==false){
        ev.preventDefault();
        mensaje.innerHTML=<p>Error: nombre de usuario no válido</p>;
    }
});
```

```
</script>
```

La captura del evento permite detener el envío de datos, si no se cumple la expresión regular en el valor del cuadro de texto en el que se escribe el nombre de usuario.

### 8.5.2.2 EVENTO RESET

Este evento se produce cuando se ha ordenado, a través de un botón de tipo **reset** por ejemplo, que el formulario muestre los valores iniciales y borre lo que el usuario hubiera escrito. La captura del evento nos permitirá matizar esta operación o añadir acciones a la misma.

### 8.5.2.3 EVENTOS DE ENFOQUE

Son eventos asociados a los controles de formulario en los que el usuario puede escribir o elegir opciones. Un control obtiene el **foco** cuando al usar el teclado manejamos dicho control. El enfoque se consigue gracias a la interacción con el dispositivo apuntador (un clic de ratón por ejemplo) o al cambio de control de formulario gracias a la tecla del tabulador.

Un control de formulario lanza el evento **focus** cuando obtiene el foco y lanza **blur** cuando pierde el foco.

### 8.5.2.4 EVENTO DE CAMBIO DE VALOR

El evento **change** es, seguramente, el evento más importante de los formularios. Se produce cuando modificamos el valor de cualquier control de formulario. Ejemplos de ello son:

- Cambiar el estado de un botón de radio o de tipo **check**.
- Modificar el valor de un cuadro de texto o de 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.

## 8.5.3 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 podemos hacer lo siguiente:

```
consolé.log(control.getAttribute("valué"));
```

Suponiendo que **control** es el nombre de un control de formulario, efectivamente aparece el valor. Pero, el valor es algo que se cambia de forma dinámica en los formularios. Si el usuario ha modificado, por ejemplo, el valor de un cuadro de texto porque ha escrito algo en él, **getAttribute("value")** no refleja estos cambios. Por ello es mejor usar propiedades que sí reflejen los cambios que hace el usuario. Estas propiedades son:

PROPIEDAD	USO	CONTROLES QUE LA USAN
<b>name</b>	Nombre del control.	Prácticamente todos.
<b>type</b>	Valor de atributo <b>type</b> .	Controles de tipo <b>input</b> .
<b>value</b>	Devuelve el valor actual del control.	Prácticamente todos.
<b>checked</b>	Puede valer true o false. Índica, sobre un control de activar o desactivar, si el control está activado o no.	Botones de <b>radio</b> y botones de tipo <b>check</b> .
<b>defaultChecked</b>	Indica el estado inicial de la propiedad <b>checked</b> en el control.	Botones de <b>radio</b> y botones de tipo <b>check</b> .
<b>disabled</b>	Índica, con <b>true</b> o <b>false</b> , si el control está deshabilitado o no.	Prácticamente todos.
<b>readonly</b>	Índica, con <b>true</b> o <b>false</b> , si el control es de solo lectura o no.	Prácticamente todos.
<b>required</b>	Indica con <b>true</b> o <b>false</b> si es obligatorio o no cambiar el valor del control.	Controles de entrada de texto o listas.
<b>maxLength</b>	Permite ver y modificar la propiedad que define la anchura máxima de texto.	Controles de entrada de texto.
<b>min</b>	Valor mínimo posible para el control.	Input de tipo numérico o de fecha.
<b>max</b>	Valor máximo posible para el control.	Input de tipo numérico o de fecha.
<b>step</b>	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.
<b>selectionStart</b>	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.  Santa Bárbara ¡número 5  El cuadro de texto con selección que se muestra en la imagen anterior tendría estos valores en las propiedades:  ■ <b>seleccionStart: 6</b>  ■ <b>seleccionEnd: 14</b>	Controles de entrada de texto.

PROPIEDAD	USO	CONTROLES QUE LA USAN
<b>selectionEnd</b>	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

## 8.5.4 MÉTODOS DE LOS CONTROLES

Los controles también ofrecen métodos con los que realizar acciones que simulan la acción del usuario. La lista es la siguiente:

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

## 8.6 PRÁCTICAS SOLUCIONADAS

### Práctica 8.1: Cartel que persigue al ratón

- Crea una página web con bastantes párrafos.
- Haz que en cada movimiento de ratón sobre la página, un pequeño cartel con fondo amarillo y el texto Hola, siga al cursor del usuario o usuaria.

#### SOLUCIÓN: PRÁCTICA 8.1

Podemos empezar creando una serie de párrafos de muestra que cubren mucho contenido. Por ejemplo, 50 párrafos con el texto de prueba  *Lorem ipsum...* Esto en Visual Studio Code lo podemos conseguir escribiendo dentro del apartado **body**:

```
p*50>lorem
```

Después pulsamos inmediatamente la tecla tabulador.

Además, crearemos una capa con posicionamiento de tipo fixed (y por lo tanto inmune al desplazamiento) y que tendrá un tamaño adecuado para el texto *Hola*. Esa capa se identificará como *cHola*. Un resumen del código HTML y CSS sería este:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Cartel perseguidor</title>
    <style>
        #cHola{
            background-color: yellow;
            color: black;
            position: fixed;
            left: 0;
            top: 0;
            padding: 4px;
            width: 100px;
            height: 20px;
            text-align: center;
        }
    </style>
</head>
<body>
    <div id="cHola">Hola</div>
    <p>Lorem ipsum dolor sit amet consectetur, adipisicing elit. Sunt debiti esse corrupti autem repellat. Praesentium similique, excepturi repudiandae facilis dignissimos, cupiditate unde cumque voluptatum sit fugit</p>
</body>
```

corporis tempore. Aliquam, amet. Aspernatur ipsum in nostrum praesentium reprehenderit odit maiores iure nesciunt animi amet soluta quam quis tenetur dolores, similique fugit sed molestias temporibus facere!

Dignissimos ea omnis ipsum libero amet dolores.</p>

<p>Esse, in impedit optio facilis repellendus sed quia voluptates itaque, sint, voluptatum eos quo ducimus? Accusantium aspernatur nesciunt in quas blanditiis recusandae, quam magnam voluptatum ipsam pariatur officiis corporis similique laboriosam ex. Ipsa explicabo obcaecati totam tenetur cum, facilis blanditiis ipsam aliquid quidem commodi ullam, corrupti fuga necessitatibus soluta porro quas cupiditate nam accusamus deleniti in eius eaque? Molestias, quis.</p>

...  
<p>Accusantium labore ut esse nam ullam iusto aut illum quidem laborum. Beatae, ea. Quibusdam quod quo voluptatum in alias, dolor consequuntur nam nobis, est id aliquid amet beatae quas dolorem cumque velit sapiente iure quos ipsa? Libero cum illo nam optio saepe. Vel necessitatibus repudiandae distinctio. Incidunt, ut soluta harum exercitationem quos qui vel illum non eveniet porro esse saepe?</p>

<script src="acción.js"></script>

</body>

</html>

No se han colocado todos los párrafos para evitar ocupar mucho espacio, sin que apenas aporte demasiado la información. Se invoca al archivo JavaScript **accion.js**, el cual contiene este código:

```
document.body.addEventListener("mousemove",function(ev){
    let cHola=document.getElementById("cHola");
    cHola.style.left=ev.clientX+"px";
    cHola.style.top=ev.clientY+"px";
});
```

Se captura el evento **mousemove** que se produce cada vez que el usuario mueve el ratón, en este caso es el evento ideal a capturar. Lo único que hacemos es tomar las coordenadas clientX y clientY del evento y dárselas a las propiedades **left** y **top** de la capa, añadiendo la palabra **px** para indicar la unidad a CSS.

## Práctica 8.2: Tecla que pone imagen de fondo

- Crea una página web que tenga un texto que indique que al pulsar Alt+F12, podremos colocar una imagen de fondo. El texto tiene que salir centrado.
- La idea es que inicialmente aparezca una pantalla negra con el texto y hasta que el usuario no pulse esa tecla, la imagen no se muestre.



Pulsa Alt + F12 para cambiar la imagen de fondo

**Figura 8.2:** Imagen de la aplicación antes de pulsar la combinación de teclas Alt+F12

- Tras pulsar Alt+F12 una imagen ocupará el fondo completo. Se aconseja usar una imagen aleatoria del servicio gratuito **unsplash** (<https://source.unsplash.com/random>).



**Figura 8.3:** Ejemplo de apariencia de la aplicación tras pulsar la combinación de teclas Alt+F12

---

## SOLUCIÓN: PRÁCTICA 8.2

Curiosamente, lo más difícil de esta práctica es el formato CSS. Porque el código JavaScript es muy sencillo. Se presenta el código HTML que incluye tanto el CSS como el JavaScript de esta práctica:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Imagen tras tecla</title>
    <style>
        #imagen{
            background-color: black;
            position:fixed;
            left:0;
            top:0;
            width:100%;
            height:100%;
```

```

        display:flex;
        justify-content: center;
        align-items: center;
        background-size:cover;
        background-position: center;

    }
    h1{
        color:white;
        text-align: center;
    }
</style>
</head>
<body>
    <div id="imagen">
        <h1>Pulsa <kbd>Alt + F12</kbd> para cambiar la imagen de fondo</h1>
    </div>
    <script>
        document.body.addEventListener("keyup",function(ev){
            if(ev.altKey && ev.key=="F12"){
                let capa=document.getElementById("imagen");
                capa.style.backgroundImage=
                    "url('https://source.unsplash.com/random')";
            }
        })
    </script>
</body>
</html>

```

### Práctica 8.3: Capa *rollup*

- Crear una aplicación web que muestre una capa centrada que ocupe el 50% del ancho y el alto de la ventana.
- Inicialmente la capa será blanca y solo se la verá el borde.
- Al arrimar el ratón se colorea de verde.
- Al hacer clic encima con el botón principal se colorea de rojo, pero solo mientras el botón principal está abajo, si está arriba, entonces se quita el color rojo.
- Al hacer clic encima con el botón secundario ocurre lo mismo que en el caso anterior, pero se muestra el color azul. No se mostrará en ningún caso el menú de contexto.

### SOLUCIÓN: FIGURA 8.3

En este caso hemos dividido el código CSS, el HTML y el JavaScript. El código HTML simplemente tiene, en el cuerpo, una capa que se identifica precisamente como *capa*.

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Control de ratón</title>
    <link rel="stylesheet" href="estilos.css">
</head>
<body>
    <!DOCTYPE html>
    <html lang="es">
        <head>
            <meta charset="UTF-8">
            <title>cambio de color</title>
        </head>
        <body>
            <div id="capa"></div>
        </body>
        <script src="acción.js"></script>
    </html>
</body>
</html>

```

Se destaca la carga de la hoja de estilos, la capa y la carga del archivo JavaScript. El archivo **estilos.css** podría tener este contenido:

```

#capa{
    position:fixed;
    width:50%;
    height:50%;
    left:25%;
    top:25%;
    border:1px solid black;
}

```

Simplemente coloca la capa con el tamaño adecuado en el centro.

Finalmente el código JavaScript del archivo **accion.js** se encargará de capturar cada evento de ratón que nos interesa, además de utilizar la propiedad **button** del evento para distinguir los dos botones del ratón, y finalmente de anular el comportamiento del evento **contextmenu** para que no aparezca el menú del botón derecho,

```

var capa=document.getElementById("capa");

//El ratón entra en la capa
capa.addEventListener("mouseenter",(ev)=>{
    capa.style.backgroundColor="green";
});

```

```
//E1 ratón sale de la capa
capa.addEventListener("mouseleave",(ev)=>{
    capa.style.backgroundColor="transparent";
});

//Se aprieta el botón principal sobre la capa
capa.addEventListener("mousedown",(ev)=>{
    if(ev.button==0)
        capa.style.backgroundColor="red";
    else if(ev.button==2)
        capa.style.backgroundColor="blue";
});

//Se suelta el botón principal sobre la capa
capa.addEventListener("mouseup",(ev)=>{
    capa.style.backgroundColor="green";
});

//Se intenta sacar el menú de contexto, acción que se provoca
//en el botón secundario, pero que se debe anular aparte
capa.addEventListener("contextmenu",(ev)=>{
    ev.preventDefault();
});
```

## Práctica 8.4: Control de velocidad

- Crea una aplicación web que muestre el texto: *Velocidad 0.*
- Si pulsamos la tecla de la flecha hacia arriba del teclado, la velocidad se incremente en uno.
- Si es la flecha hacia abajo, la velocidad baja en uno.
- La velocidad no puede sobrepasar los 120 ni descender de cero.
- Experimenta con los eventos de teclado para investigar cuál es el más apropiado en este caso.

### SOLUCIÓN: PRÁCTICA 8.4

Se trata de una práctica muy sencilla. Es importante, para aprender la diferencia, hacer otras soluciones sustituyendo el evento **keydown** (que es el aconsejable en este caso) por los eventos **keyup** y **keypress**.

Este podría ser el código:

```
<!DOCTYPE html>


```

```

        font-size:2em;
    }
</style>
</head>
<body>
    <p>Velocidad <span>0</span> </p>
    <script>
        const TOPE_SUPERIOR=120;
        const TOPE_INFERIOR=0;
        let velocidad=document.querySelector("span");
        document.body.addEventListener("keydown", (ev)=>{
            let v=Number(velocidad.textContent);
            if(ev.key=="ArrowUp"){
                if(v<TOPE_SUPERIOR)
                    velocidad.textContent = v + 1;
            }
            else if(ev.key=="ArrowDown"){
                if(v>TOPE_INFERIOR)
                    velocidad.textContent = v - 1;
            }
        });
    </script>
</body>
</html>

```

## Práctica 8.5: Cierre de panel con scroll

- Crea una capa que ocupe el 50% del ancho y el alto de la ventana y que en ella se muestre una gran cantidad de texto a la que podemos acceder mediante las barras de desplazamiento.
- Cuando hayamos conseguido llegar al final del desplazamiento, se mostrará un botón con el texto "***Cerrar***". Haciendo clic en él, se elimina la capa anterior.

---

### SOLUCIÓN: PRÁCTICA 8.5

Se trata de un ejemplo muy parecido al que utilizan muchas aplicaciones para mostrar las condiciones de uso del software. Al llegar al final de la lectura, se permite aceptar esas condiciones y seguir con la instalación. En este caso, al llegar al final del panel, simplemente cerramos el panel.

Empezamos creando el HTML que lo único que hace es cargar el CSS del archivo **estilos.css** y el javaScript **accion.js**. Además crea un primer panel con el nombre **capaScroll** y una segunda capa llamada **capaBoton** en la que se mostrará el botón cuando lleguemos al final del scroll. Hemos creado el contenido de forma automatizada, como viene siendo habitual en muchas otras prácticas.

```

<!DOCTYPE html>
<html lang="es">

```

```

<head>
    <meta charset="UTF-8">
    <title>Scrol1</title>
    <link rel="stylesheet" href="estilos.css">
</head>
<body>
    <div id="capa">
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit.
Fuga ipsa delectus pariatur, quasi nobis eos perspiciatis harum aspernatur assumenda porro animi velit, eaque debitis laborum sit? Dignissimos reprehenderit corrupti deserunt?</p>
        <p>Reprehenderit amet omnis inventore quod, reiciendis molestias impedit quo rerum? Incidunt consequuntur sit laboriosam esse commodi atque soluta, fugit, tempore possimus illo labore omnis nihil ex quibusdam nesciunt cum animi!</p>
        <p>Expedita quam ab harum. Eius tempore dolore ducimus commodi, porro quis accusantium expedita dolores in molestias assumenda nobis ipsum possimus sapiente! Vitae dolorum debitis beatae dolore nesciunt quasi doloribus porro?</p>
        ...
        <p>Consectetur qui voluptate placeat dolores quo illum quia numquam consequuntur laudantium saepe, quas dolorum laboriosam eum quidem vero accusamus quisquam aperiam, vitae inventore. Dolorum quisquam obcaeci eum nostrum, sit suscipit.</p>
    </div>
    <div id="boton">
        <button>Cerrar</button>
    </div>
    <script src="acción.js"></script>
</body>
</html>

```

El código CSS oculta la capa del botón y deja el tamaño de la primera capa y la marca para que se pueda acceder a su contenido con barras de desplazamiento:

```

#capa{
    position:fixed;
    width:50%;
    height:50%;
    left:25%;
    top:25%;
    border:1px solid black;
    overflow: scroll;
}
#botonf
    position:fixed;

```

```

width:100%;
height:100px;
bottom:0;
left:0;
display:none;
text-align: center;
background-color: black;
}
«boton buttonf
height:50px;
width:50%;
margin-top:25px;
}

```

Finalmente, el código JavaScript es muy sencillo gracias a las propiedades de scroll del DOM. El evento permite obtener la coordenada de desplazamiento vertical (**scrollTop**) y de la propia capa la propiedad **scrollHeight** nos permite saber el tamaño completo, y **clientHeight** el tamaño de la capa en la ventana del usuario actual. La suma de scrollTop y clientHeight, cuando hemos llegado al final, tiene que llegar al tamaño de scrollHeight. Esa es la base del código.

```

let capaScroll=document.getElementById("capa");
let capaBoton=document.getElementById("boton");
let boton=document.querySelector("«boton button»");
capa.addEventListener("scroll",(ev)=>{
    //para saber si hemos llegado al final
    //hay que sumar la anchura del elemento
    //a la propiedad scrollTop y ver si redondeando
    //hacia arriba, hemos llegado a scrollHeight
    if(Math.ceil(capaScroll.scrollTop+capaScroll.clientHeight)>=
        capaScroll.scrollHeight){
        capaBoton.style.display="block";
    }
    else{
        capaBoton.style.display="none";
    }
});
boton.addEventListener("click",(ev)=>{
    document.body.removeChild(capaScroll);
});

```

## Práctica 8.6: Panel con el mensaje "cargando"

- Crea una página que descargue de Internet 10 imágenes de alta calidad
- Se mostrará un panel con el mensaje "***Cargando...***" durante la carga.
- El panel se retira automáticamente cuando se cargan todas las imágenes

## SOLUCIÓN: PRÁCTICA 8.6

En el código HTML hemos utilizado varios servicios de Internet que permiten descargar imágenes de forma gratuita y aleatoria. Con esa carga, la página tarda un poco en mostrarse.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8"      >
    <title>Panel de carga</title>
    <link rel="stylesheet" href="estilos.css">
</head>
<body>
    <div id="cargando">
        <h1>Cargando</h1>
    </div>
    
    
    
    
    
    <script src="acción.js"></script>
</body>
</html>
```

El código CSS permite colocar la capa tapando todo el contenido que se va cargando. Además, se crea una clase llamada *ocultar* que oculta la capa haciendo una animación que la hará ver cómo desciende de arriba a abajo, descubriendo el contenido de la página.

```
#cargando{
    background-color: black;
    position:fixed;
    left:0;
    top:0;
    width:100%;
    height:100%;
    display:flex;
    justify-content: center;
    align-items: center;
    background-size:cover;
    background-position: center;
    z-index:10;
}
.ocultar{
    top: 110% !important;
    transition:ls;
}
```

```
hl{
    color:white;
    text-align: center;
}
```

Lo más sencillo es el código JavaScript:

```
window.addEventListener("load",(ev)=>{
    let cargando=document.getElementById("cargando");
    cargando.classList.add("ocultar");
});
```

## Práctica 8.7: Arrastre de una capa en otra

- Crea una aplicación que muestre dos capas del mismo tamaño concretamente 200 píxeles de ancho por 100 de alto.
- La primera capa mostrará el texto "***Soy arrastrable***" y tendrá fondo amarillo y la segunda muestra "***Soy el destino***" y tiene fondo blanco. Ambas dejan un borde de un píxel, negro.
- La aplicación permite arrastrar la primera sobre la segunda.
- Durante el arrastre, la primera capa se mostrará con una opacidad del 50%. Al arrastrar sobre la segunda, esta (el destino) se muestra con fondo rojo. Al soltar en esa segunda capa, la primera desaparece y en la segunda aparecerá el texto "***;Lo has logrado!***"-

### SOLUCIÓN: PRÁCTICA 8.7

Esta práctica la vamos a resolver usando la interfaz de tipo ***drag and drop*** de HTML5. Esta interfaz se basa en eventos de arrastre, los cuales funcionan en elementos con el atributo **draggable** colocado en **true**.

El código HTML es muy sencillo:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Arrastre de una capa en otra</title>
    <link rel="stylesheet" href="estilos.css">
</head>
<body>
    <div id="capa1" draggable="true" style="background-color: yellow">
        Soy arrastrable
    </div>
    <div id="capa2">Soy el destino</div>
    <script src="acción.js"> </script>
</body>
</html>
```

El código CSS (archivo **estilos.css**) sería:

```
div{
    border:1px solid black;
    padding:1em;
    position:fixed;
    top:20px;
    width:200px;
    height:100px;
}
#capa1 {
    left:50px;
}
#capa2{
    left:400px;
}
```

Finalmente, el código JavaScript (**accion.js**):

```
let capa1=document.getElementById("capa1")
let capa2=document.getElementById("capa2")
capa1.addEventListener("drag",(ev)=>{
    capa1.style.opacity=.5;
});
capa1.addEventListener("dragend",(ev)=>{
    capa1.style.opacity=1;
    capa1.style.backgroundColor="transparent";
});
    capa2.addEventListener("dragenter",(ev)=>{
        capa2.style.backgroundColor="red";
});
    capa2.addEventListener("dragleave",(ev)=>{
        capa2.style.backgroundColor="transparent";
});
    capa2.addEventListener("dragover",(ev)=>{
        ev.preventDefault();
});
    capa2.addEventListener("drop",(ev)=>{
        document.body.removeChild(capa1);
        capa2.textContent=" ¡Lo has logrado!";
        capa2.style.backgroundColor="yellow";
});
```

Es muy importante tener en cuenta qué eventos hay que escuchar en la capa de arrastre y cuáles en la capa destino. También es fundamental cancelar la acción predeterminada para el evento **dropover** en la capa de destino.

## Práctica 8.8: Validación de formularios

- Crea una aplicación con esta apariencia inicial:

**INFORME INCIDENCIA**



Tipo de incidencia

Número de serie

**Figura 8.4:** Imagen inicial, ejemplo, de la Práctica 8.8

- La imagen de la izquierda se descargará de Internet (en páginas que permiten la descarga sin restricciones de derecho de uso). Realmente necesitamos tres: una relacionada con el área de distribución, otra con el área de oficina y una tercera para el área de producción.
- El cuadro combinado ***Tipo de incidencia*** contiene esas mismas tres entradas: distribución, oficina y producción. Eligiendo cada opción, se modifica automáticamente la imagen de la izquierda.
- Cuando pulsamos en el botón con el texto ***Mostrar descripción***, aparece un área de texto en el que podemos describir la incidencia. Tras el botón la apariencia es esta:

**INFORME INCIDENCIA**



Tipo de incidencia

Número de serie

Descripción

**Figura 8.5:** Imagen de la Práctica 8.8 tras pulsar el botón de mostrar descripción

- El número de serie debe de cumplir estas reglas:
  - Empezar con tres números
  - Seguir con 4 letras mayúsculas
  - Acabar con los números 1 o 2; o bien con la letra A
- Se valida el número de serie al pulsar enviar. Si es incorrecto, se avisará con un mensaje de error (fondo rojo y letra blanca). Además se marcará la etiqueta y el cuadro de texto con un borde fino de color rojo. Tanto el mensaje como los bordes, se quitan cuando el usuario se dispone a modificar de nuevo el número de serie.
- Los datos, realmente, no se enviarán a ningún destino.

## SOLUCIÓN: PRÁCTICA 8.8

Esta práctica es un buen ejemplo de validación, donde el botón de enviar se controla para que no envíe nada si hay errores. Además, hay elementos de formulario que modifican elementos del DOM. Todo ello permite una práctica ya bastante real de cómo usar formularios en las aplicaciones web para comunicarse con el usuario.

Supones que hemos creado tres directorios: **img** para las imágenes, **css** para el código CSS y **js** para JavaScript.

Lo primero es ver el HTML. Es extenso debido a la maquetación en forma de tabla de la parte del formulario:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8"      >
  <title>Informe de Incidencias</title>
  <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
<header>
  <h1>INFORME INCIDENCIA</h1>
</header>
<figure>
  
</figure>
<main>
  <form action="#">
    <table>
      <tr>
        <td>
          <label for="tipo">
            Tipo de incidencia
          </label>
```

```
</td>
<td>
    <select name="tipo" id="tipo">
        <option value="distribución" selected>
            Distribución
        </option>
        <option value="oficina">Oficina</option>
        <option value="producción">Producción</option>
    </select>
</td>
</tr>
<tr>
    <td>
        <label id="labelSerie" for="serie">
            Número de serie
        </label>
    </td>
    <td>
        <input type="text" name="serie" id="serie">
    </td>
</tr>
<tr>
    <td colspan="2">
        <button type="button" id="mostrarDescripcion">
            Mostrar Descripción
        </button>
        <div id="fDescripcion">
            <label for="descripción" id="labelDescripción">
                Descripción
            </label> <br>
            <textarea name="descripción" id="descripción" cols="30" rows="10">
            </textarea>
        </div>
    </td>
</tr>
<tr>
    <td colspan="2">
        <br>
        <button id="enviar">Enviar datos</button>
    </td>
</tr>
</table>
<div id="capaError">

</div>
```

```

        </form>
    </main>

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

```

La hoja CSS (estilos.css) es la siguiente:

```

h1{
    text-align: center;
    color:white;
    font-family:         sans-serif;
    background-color: perú;
}

figure{
    position:absolute;
    top:100px;
    left:0;
    width:300px;
}

figure img{
    width:100%;
}

main{
    position:absolute;
    top:100px;
    left:400px;
}

*fDescripcion{
    display:none;
}

#capaError{
    background-color:      red;
    color:white;
    margin-top:2em;
}

.error{
    border:1px solid red;
}

```

Finalmente, el código JavaScript (**accion.js**):

```

//Todo el código lo colocamos de forma que tengamos
//asegurada toda la carga del DOM
window.addEventListener("load",function(ev){
    //asignación de elementos
    var imagen=document.getElementById("imagen");
    var combo=document.getElementById("tipo");
    var serie=document.getElementById("serie");
    var labelSerie=document.getElementById("labelSerie");
    var labelDescripcion=document.getElementById("labelDescripcion");
    var enviar=document.getElementById("enviar");
    var mostrarDescripción=document.getElementById("mostrarDescripción");
    var capaError=document.getElementById("capaError");
    var fDescripcion=document.getElementById("fDescripcion");

    //Cuadro combinado que cambia la imagen
    combo.addEventListener("change",function(ev){
        imagen.setAttribute("src",'img/${combo.value}.jpg');
        labelSerie.removeAttribute("class");
        serie.removeAttribute("class");
    });

    //Si se coloca el foco en el número de serie
    //quitamos los errores
    serie.addEventListener("focus",function(ev){
        capaError.textContent="";
        labelSerie.classList.remove("error");
        serie.classList.remove("error");
    });
    //Validación del número de serie al intentar enviar los datos
    enviar.addEventListener("click",function(ev){
        if(/^[0-9]{3}[A-Z]{4}[12A]$/.test(serie.value)==false){
            ev.preventDefault();
            capaError.textContent="Código no válido";
            labelSerie.classList.add("error");
            serie.classList.add("error")
        }
    });
    //Click en el botón de descripción, muestra una capa
    //con el área de texto y la etiqueta
    mostrarDescripción.addEventListener("click",function(ev){
        fDescripcion.style.display="block";
        this.style.display="none";
    });
});
}
);

```

## 8.7 PRÁCTICAS RECOMENDADAS

### Práctica 8.9: Repasar Modelos de Red

- Crea una aplicación web con esta apariencia aproximada:

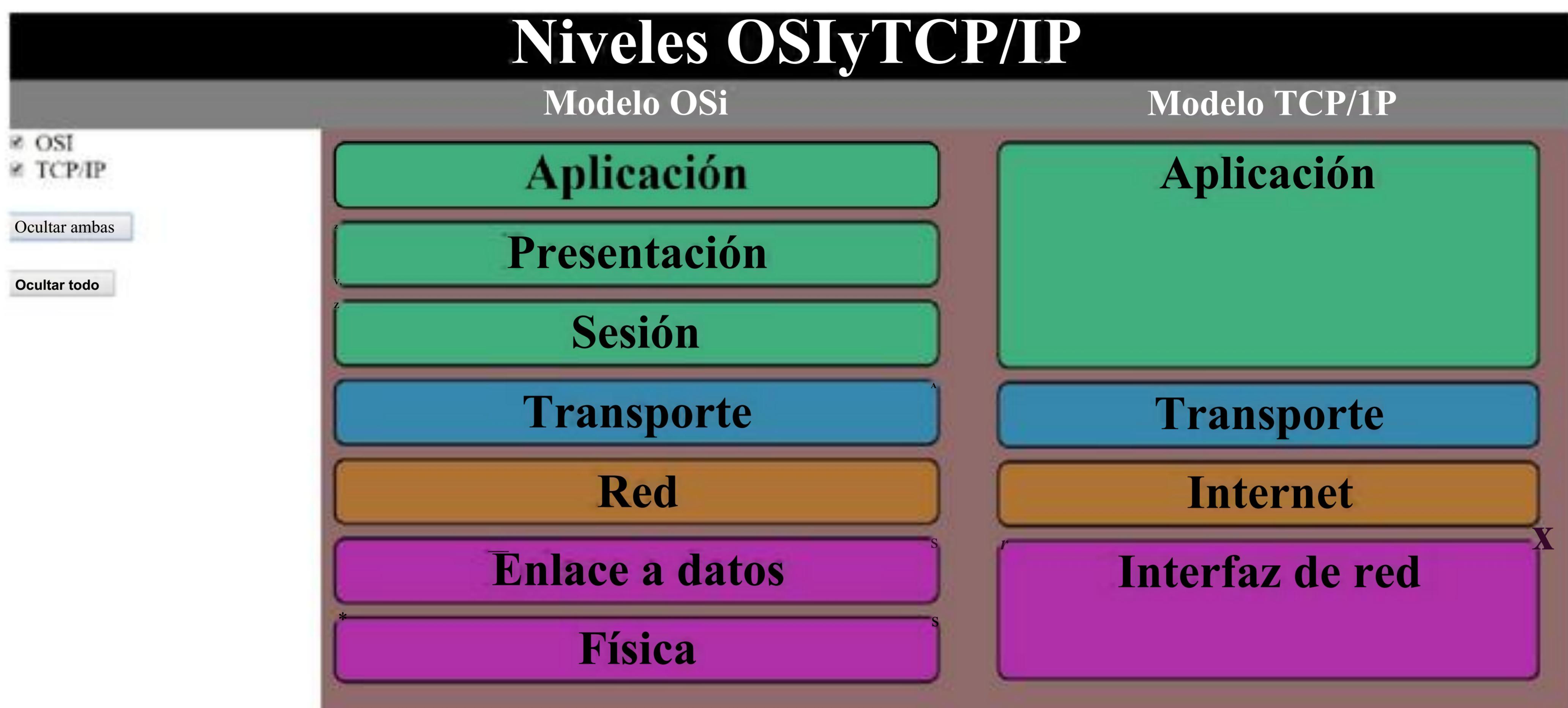


Figura 8.6: Aspecto de la aplicación con los dos modelos a la vista

- Tanto el **modelo OSI** como el **TCP/IP** salen cuando se activan las casillas de verificación correspondientes. Inicialmente tanto las casillas como los modelos están desactivados.
- El botón **Ocultar ambas**, oculta ambos modelos y además desactiva las casillas.
- El botón **Ocultar todo** tapa toda la página con un velo negro. Ese negro sale desde arriba con una animación hasta que oculta toda la página. Tras aparecer y quedar todo en negro, podemos hacer clic en el velo y vuelve a su posición original (también mediante una animación).

### Práctica 8.10: Lista de tareas

- Crea una aplicación web con la apariencia aproximada de la Figura 8.7.
- El botón “+” añade tareas, pero solo si hay contenido en el cuadro
- Cada botón con el texto “**quitar**”, quita la tarea correspondiente. En cuanto se añade una tarea, este botón aparece en ella para poder realizar esta labor.
- Al hacer clic en una tarea (en el nombre de la misma), se quita de donde está, y pasa a ser la primera de la lista.
- El único tipo de letra utilizado es **Montserrat** (letra de **Google Font**) en grosores 400 y 700.
- Colores: **teal**, **brown** y **red**.

- Consigue que las tareas se vayan grabando en una cookie que dure una semana, así al entrar de nuevo, con el mismo navegador, veremos las tareas que habíamos añadido.

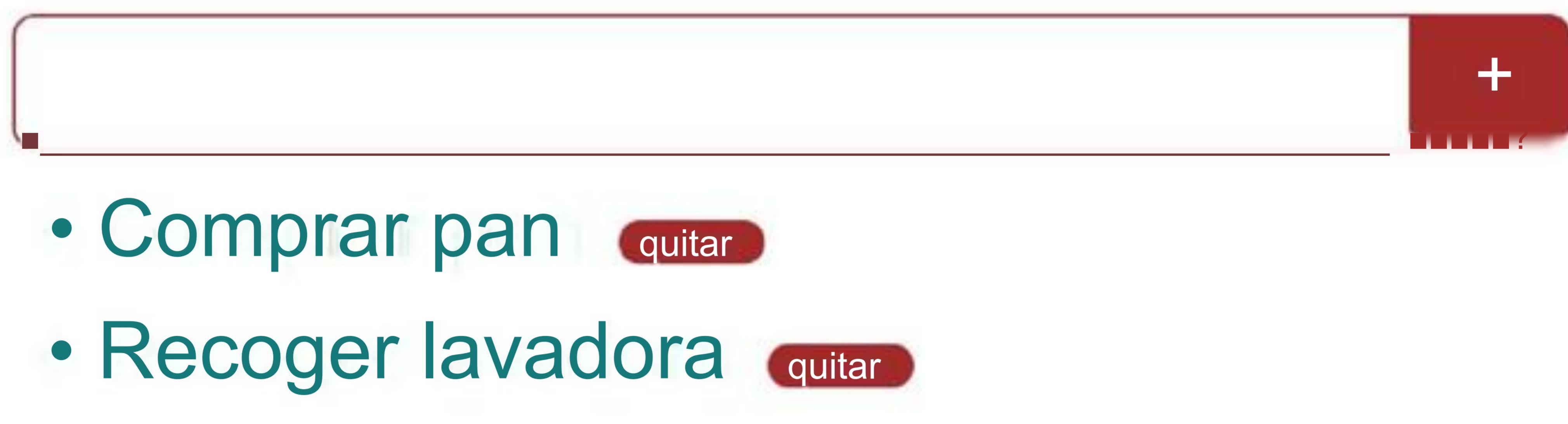


Figura 8.7: Aspecto de la aplicación con dos tareas añadidas

### Práctica 8.11: Calculadora

- Crea una aplicación web con esta apariencia aproximada:

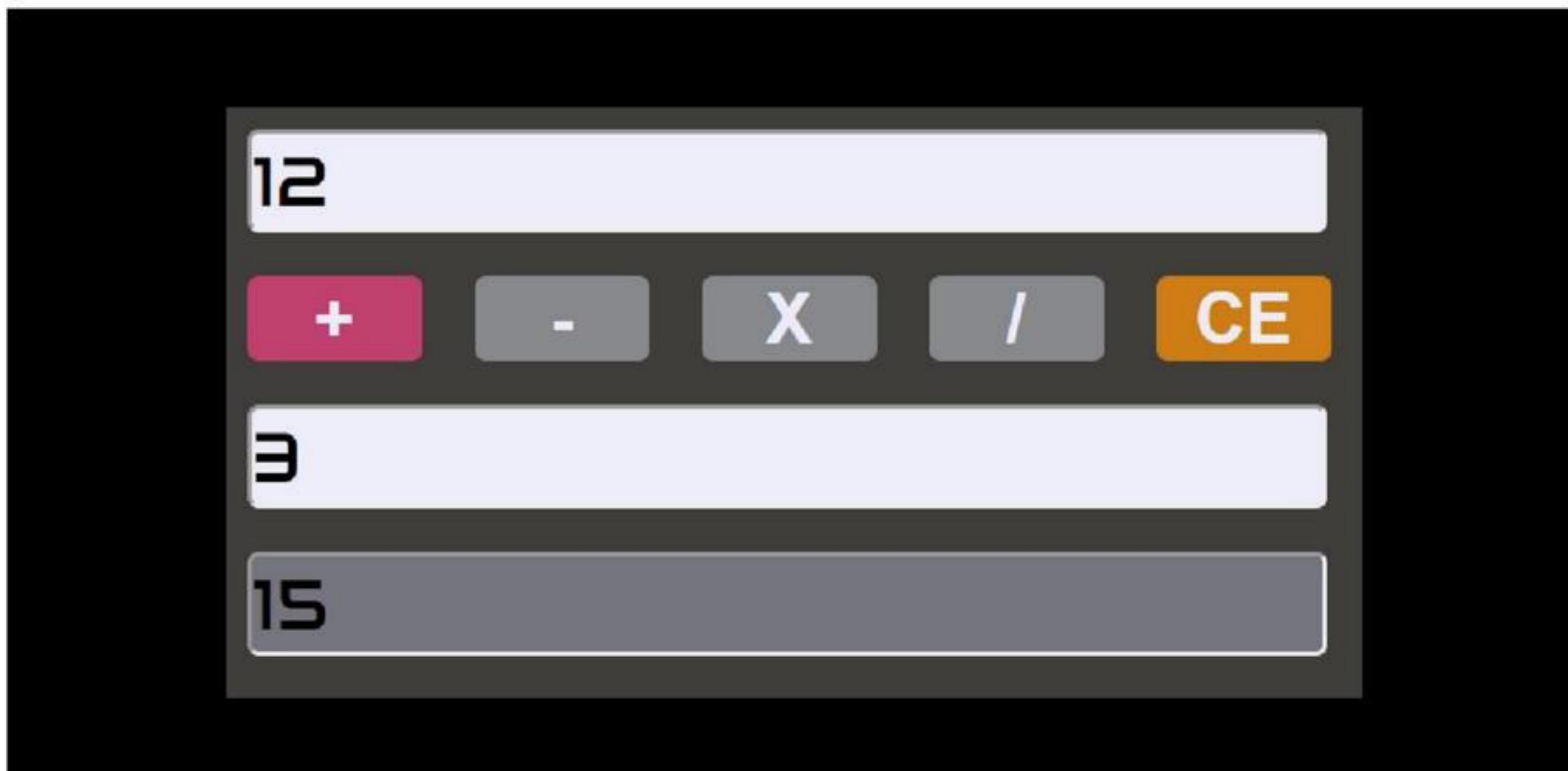


Figura 8.8: Aspecto de la aplicación tras sumar los números 12 y 3

- Se trata de una calculadora que debe de funcionar perfectamente con los botones que aparecen.

- La letra de los cuadros numéricos es de Google Fonts y se llama **Audiowide**.
- El cuadro debe de salir totalmente centrado en la pantalla independientemente de la resolución
- Los colores usados son:
  - negro
  - #3F3e3b (fondo calculadora)
  - #87898D (fondo botón)
  - #C0416E (fondo al arrimar el ratón)
  - #EDEEF9 (color letra botones y fondo cuadros numéricos)
  - # 74757e (fondo control de resultado)
- Hay que conseguir que no se pueda escribir en el cuadro de texto en el que aparece el resultado y que cuando se pulsen los botones correspondientes se calcule el resultado en base a los valores escritos.

---

## Práctica 8.12: Buscaminas

- Consigue crear el juego del buscaminas. Básate en lo realizado en la Práctica 6.9 en la página 243 y en la Práctica 5.11 en la página 198.
  - Ahora el tablero será interactivo, al hacer clic veremos si hay una mina o no. SI no la hay, se nos informará de cuántas minas tiene esa casilla alrededor.
  - Si descubrimos todas las casillas sin mina, habremos ganado la partida
- 

---

## Práctica 8.13: Juego del helicóptero

- Consigue crear un juego en el que un helicóptero intenta posarse en una torre.
- Las imágenes para crear el juego están disponibles en  
<https://github.com/jorgesancheznet/libro-dwec/tree/master/unidad-08/practica-08-13/img>
- Una capa muestra una imagen de un helicóptero, otra una imagen de una plataforma de aterrizaje y otra un marcador con la velocidad. El fondo lo cubre completamente la imagen del cielo
- Las flechas permiten mover el helicóptero en la dirección deseada.
- Todos los movimientos hacen efecto de acelerar y frenar en horizontal y vertical. Es decir, si el helicóptero se mueve a la izquierda y va a 10, la flecha derecha no cambia de dirección, solo aminorará la marcha. Pero, si seguimos pulsando derecha llegaremos a velocidad 0 y a partir de ahí el helicóptero va hacia la derecha.

- Lo mismo vale para subir y bajar, si el helicóptero sube, la flecha de bajar aminora la marcha, pero si insistimos en bajar, la velocidad será negativa.
- Hay que contar con la fuerza de la gravedad que siempre tira hacia abajo. En vertical es como si continuamente alguien está pulsando la flecha hacia abajo,
- Si el helicóptero sobrepasa los límites de la pantalla o choca con la plataforma aparece la imagen **boom**, en la zona donde ha chocado.
- El helicóptero tiene que posarse despacio, si se posa a mucha velocidad ocurre el ¡boom!
- Cuando posa bien, el juego termina y se indica con el texto en grande **¡bravo!**

## 8.8 RESUMEN DE LA UNIDAD

- Los eventos son uno de los elementos más importantes de la programación de aplicaciones web.
- Gracias a los eventos podemos comunicar la aplicación con el usuario y también reaccionar ante situaciones que se producen de forma asíncrona.
- La programación basada en eventos consiste en asignar el código de una función de tipo callback al evento concreto. Cada evento, además, se asocia a un elemento concreto del DOM.
- Hay varios métodos para programar eventos:
  - Usar atributos HTML como **onclick** por ejemplo.
  - Usar métodos de tipo onclick, onload, etc. Que son métodos de los elementos del DOM.
  - Usar el método **addEventListener** que es el más recomendado y que asocia el evento a una función callback de forma más abierta, además de poder asociar más de una función.
- Los eventos se propagan en dos fases: primero de los elementos más grandes a los más pequeños (fase de captura) y luego de los pequeños a los grandes (fase de burbuja). Por defecto, el código asociado al evento se lanza en la fase de burbuja, pero podemos cambiar este comportamiento.
- Al ocurrir un evento se crea un objeto especial que posee propiedades y métodos relacionadas con el evento. Estas propiedades y métodos dependen del tipo de evento, pero son detalles como las coordenadas del ratón, la tecla que hemos pulsado, la posibilidad de anular el comportamiento por defecto del evento, etc.
- Podemos incluso crear nuestros propios eventos y ejecutarlos para su captura igual que los demás eventos.
- Hay numerosos tipos de eventos, lo que nos permite usar este tipo de programación en prácticamente cualquier circunstancia.
- Los formularios alcanzan una nueva dimensión con el apoyo de los eventos. Hay eventos especialmente preparados para los formularios. Además, podemos modificar casi cualquier propiedad de los controles del formulario desde JavaScript, lo que convierte a los formularios en la vía prioritaria para comunicarse con el usuario.

## 8.9 TEST DE REPASO

1.- ¿Qué método es el prioritario para capturar eventos actualmente en JavaScript?

- a) Usar atributos asociados al evento de HTML.
- b) Usar propiedades de los elementos con el del evento.
- c) Usar el método **addEventListener**.
- d) Usar el método **on**.

2.- Observa este código

```
capa.addEventListener("click",
  ()=>{
    consolé.log("Hola")
},true);
```

¿En qué fase se lanza el código asociado al evento?

- a) En la de captura.
- b) En la de burbuja.
- c) En ambas.
- d) En ninguna.

3.- ¿Para qué sirve el método "target" de los objetos de evento ?

- a) Para llegar a las propiedades del objeto de evento.
- b) Para obtener el tipo de evento que se ha producido.
- c) Para obtener el elemento en el que se ha producido el evento.
- d) Para obtener el método de captura del evento.

4.- Queremos distinguir si hemos pulsado el botón izquierdo o el derecho del ratón (posiciones de diestro) ¿Qué propiedad del objeto de evento lo permite?

- a) **location**
- b) **mouse**
- c) **key**
- d) **button**
- e) No se puede averiguar qué botón se ha pulsado

5.- ¿Qué método permite dejar de escuchar un evento?

- a) **removeEventListener**
- b) **addEventListener**
- c) **preventDefault**
- d) **cancelEvent**

6.- Queremos crear un nuevo evento de tipo "dblclick" ¿Qué instrucción lo permite?

- a) new Event("dblclick")
- b) window.createEvent("dblclick")
- c) elemen.createEvent("dblclick")
- d) elemen.  
addEventListenner("dblclick")

7.- ¿Qué evento se produce antes?

- a) click
- b) dblclick
- c) mousedown
- d) mouseup

8.- ¿Qué atributo hay que activar en los elementos que deseamos arrastrar mediante eventos **drag**?

- a) allowDrag
- b) useDrag
- c) **drag**
- d) draggable

9.- ¿Qué evento es el que debemos comprobar para saber si el usuario está situado en un control de formulario?

- a) click
- b) keydown
- c) blur
- d) focus

10.- ¿Qué controles producen eventos de tipo **change**?

- a) **input** de tipo **text**
- b) **select**
- c) **textarea**
- d) Todos los anteriores