

- 6.30. Profundiza en la clasificación de nodos del DOM por tipo y escribe un programa en el que los identifiques por medio de sus constantes asociadas.
- 6.31. Investiga cuáles son los métodos disponibles del objeto **HTMLCollection** (tipo de dato devuelto al hacer `getElementsByName`), elige uno, y escribe un programa donde pruebes su funcionamiento.
- 6.32. ¿Qué otras propiedades, que no se hayan visto, consideras de utilidad para navegar por el DOM?
- 6.33. Escribe un pequeño ensayo sobre los peligros y las vulnerabilidades que podría tener un cliente cuyo navegador tiene las *cookies* habilitadas. ¿Cómo propondrías solucionarlas sin desactivar las *cookies*?
- 6.34. Busca información sobre el concepto *cross-site scripting* y explica con tus palabras qué es y cómo se puede evitar.

Enlaces web de interés

-  **Mozilla Developers** - https://developer.mozilla.org/es/docs/Web/API/Document_Object_Model
(website para desarrolladores)
-  **Uniwebsidad** - <https://uniwebsidad.com/>
(recursos para el aprendizaje de la programación)
-  **W3Schools** - <https://www.w3schools.com/>
(sitio web dedicado a la programación)
-  **Stackoverflow** - <https://stackoverflow.com/>
(la mayor comunidad de programadores de internet)
-  **ECMAScript 2022** - <https://262.ecma-international.org/13.0/>
(última actualización de la especificación del lenguaje)
-  **W3Resource** - <https://www.w3resource.com/>
(recursos libres para programadores front-end)
-  **Ayuda Ley** - <https://ayudaleyprotecciondatos.es/>
(todo lo necesario para conocer la Ley de Protección de Datos y Garantía de Derechos Digitales, en relación con las *cookies*)
-  **La web del Programador** - <https://www.lawebdelprogramador.com/>
(comunidad de programadores)



Interacción con el usuario: eventos

Objetivos

- Entender la importancia de los eventos y su gestión en las aplicaciones web.
- Interiorizar los caminos bidireccionales de comunicación entre el usuario y la aplicación.
- Conocer la forma en la que se capturan, crean, lanzan y cancelan eventos.
- Comprender el funcionamiento de la propagación de eventos.
- Profundizar en la estructura del objeto evento.
- Estudiar cómo modificar el comportamiento predeterminado de los eventos asociados a algunos elementos.
- Detallar las particularidades de los formularios como parte imprescindible de la comunicación con el usuario.
- Repasar los tipos de eventos disponibles.
- Asumir las consecuencias de la naturaleza dinámica del DOM al gestionar eventos.

Contenidos

- 7.1. Introducción a los eventos
- 7.2. Captura de eventos
- 7.3. Objeto del evento
- 7.4. Propagación de eventos
- 7.5. Cancelación de eventos
- 7.6. Lanzar eventos
- 7.7. Tipos de eventos

Introducción

Cuando se ha interactuado con el usuario se tenía la limitación del uso de ciertos cuadros de diálogo ofrecidos por objetos como **window** o **document**, en los que ya estaba predefinido el comportamiento de sus botones. Pero en las aplicaciones web, esos métodos rara vez se utilizan. JavaScript proporciona una vía muy amplia mucho más potente, personalizable y atractiva de interactuar con el usuario: la gestión de eventos.

En esta unidad se aprende a capturar las acciones del usuario, darles respuesta, definir acciones propias y modificar el comportamiento predeterminado de los distintos elementos que el DOM pone a disposición del programador. Además, se descubren las posibilidades que ofrece el auténtico canal de interacción con los usuarios en las aplicaciones de hoy, el formulario.

Tras finalizar esta unidad se habrá cubierto ya todo lo imprescindible para crear aplicaciones web del lado cliente.

7.1. Introducción a los eventos

Los eventos son el idioma en el que JavaScript entiende las acciones del usuario. Es decir, son el mecanismo por el que se consigue establecer una comunicación bidireccional y en tiempo real entre la aplicación y los usuarios.

Tal y como su nombre indica, un evento no es más que un suceso que se ha producido en la página, normalmente provocado por la acción del usuario.

Lo más interesante de este mecanismo es que es asíncrono, por tanto, no hay que esperar por él. Simplemente se prepara el código para que cuando se produzca el evento el programa lo capture y ejecute la lógica que interese.

Los eventos se asocian a elementos concretos del DOM. En el argot de los programadores se dice que «escuchamos» a ese elemento, o lo que es lo mismo, tenemos un programa preparado para capturar el evento que lanza ese elemento.

7.2. Captura de eventos

Para capturar eventos se puede definir un **listener** (con **addEventListener**) sobre el elemento que interese y, por medio de un **callback** y del nombre del evento a capturar, ejecutar las acciones deseadas.

```
<p>
  Así se capturan eventos asociados
  a un <span id="miSpan">elemento</span> HTML.
</p>
```

A modo de ejemplo el siguiente párrafo contiene un **span** y se desea mostrar en consola cuántas veces el usuario hace clic sobre ese **span**:

```
let miSpan = document.getElementById("miSpan");
let veces = 0;
miSpan.addEventListener("click", ()=>{
  veces++;
  console.log(`El usuario ha hecho ${veces} clic en el span.`);
});
```

La salida de este programa se muestra en la Figura 7.1.

El usuario ha hecho 1 clic en el span.
El usuario ha hecho 2 clic en el span.
El usuario ha hecho 3 clic en el span.

Figura 7.1. Captura del evento click sobre un span.

Los eventos, como se ve, sirven para notificar sucesos interesantes que han ocurrido y que pueden capturarse para darles algún tipo de lógica programada. Cada evento tiene asociado un objeto, basado en la interfaz **Event**, que contiene propiedades y métodos para obtener más información sobre lo sucedido.

7.3. Objeto del evento

Para utilizar este objeto, que está asociado a cualquier evento, simplemente hay que indicarlo como parámetro del **callback** que gestiona el evento. Así, puede reescribirse el **listener** anterior para hacer uso de él:

```
miSpan.addEventListener("click", (objetoEvento)=>{
  console.info(objetoEvento);
});
```

Si se observa la salida en la consola puede verse el contenido del objeto.

```
* PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure: 0, ...}
  isTrusted: true
  altKey: false
  altitudeAngle: 1.5707963267948966
  azimuthAngle: 0
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 310
  clientY: 28
```

Figura 7.2. Contenido parcial del objeto de evento.

En la Tabla 7.1 se repasan algunas de sus propiedades más interesantes.

Tabla 7.1. Propiedades del objeto evento

Propiedad	Utilidad
altKey	Devuelve si la tecla [Alt] fue pulsada durante el evento.
button	Devuelve el botón del ratón que activó el evento: ■ 0: botón principal. ■ 1: botón central. ■ 2: botón secundario. ■ 3 y 4: cuarto y quinto botones (si los hubiera).
charCode	Contiene el valor Unicode de la tecla que se pulsó (evento keypress).
clientX	Coordenada X del ratón con respecto a la ventana.
clientY	Coordenada Y del ratón con respecto a la ventana.
ctrlKey	Devuelve si la tecla [Ctrl] fue pulsada durante el evento.
pageX	Coordenada X del evento, relativa al documento completo.
pageY	Coordenada Y del evento, relativa al documento completo.
screenX	Coordenada X del evento con respecto a la pantalla.
screenY	Coordenada Y del evento con respecto a la pantalla.
shiftKey	Devuelve si la tecla [Mayús] fue pulsada durante el evento.
target	Referencia al elemento que lanzó el evento.
timeStamp	Devuelve el momento en el que se creó el evento.
type	Nombre del evento.

Actividad propuesta 7.1

Contenido del objeto evento

Escribe un pequeño programa donde captures el evento **click** sobre un elemento y tengas acceso al objeto del evento. Muestra la estructura del objeto en la consola usando **console.info()** y repasa todas las propiedades y métodos que ofrece para que tengas una comprensión completa de todo aquello que contiene.

Dispone además de algunos métodos interesantes, pero es preciso avanzar en la gestión de eventos antes de abordarlos.

Para saber más

Con el siguiente enlace o con el código QR se accede a una lista de propiedades y métodos del objeto **evento**:

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



Actividad resuelta 7.1

Tras la pista del ratón

Escribe un programa que permita conocer la posición que ocupa el ratón en la pantalla cada vez que se hace clic.

Solución

```
let cuerpo = document.getElementsByTagName("body")[0];
cuerpo.addEventListener("click", (evento)=>{
    console.log(`{${evento.screenX}, ${evento.screenY}`);
});
```

7.4. Propagación de eventos

El concepto de propagación de eventos es muy importante para entender todo el ecosistema de utilidades que rodean a la gestión de eventos.

La propagación hace referencia a que los eventos pueden gestionarse desde un elemento más profundo hasta la superficie, por eso se denomina **comportamiento de burbuja**. Dicho de otra forma, en una jerarquía de elementos, pueden capturarse eventos desde el nivel más profundo hasta el nivel más superficial, cuando se activa el evento más profundo.

```
<section>
  <p>
    Así se capturan eventos asociados
    a un <span id="miSpan">elemento</span> HTML.
  </p>
</section>
```

Se ha modificado el fragmento HTML con el que se venía trabajando para incluirlo todo dentro de un elemento **section**. Ahora, el elemento que lanza el evento, ****, tiene como padre a **<p>** y como abuelo a **<section>**. Tanto **** como **<p>** y también **<section>**, pueden gestionar sus eventos, aunque lo haya lanzado ****, como consecuencia del efecto de propagación:

```
let miSection = document.getElementsByTagName("section")[0];
let miP = document.getElementsByTagName("p")[0];
let miSpan = document.getElementById("miSpan");
miSection.addEventListener("click", ()=>{
    console.log("<section>: Capturado el evento.");
});
miP.addEventListener("click", ()=>{
    console.log("<p>: Capturado el evento.");
});
miSpan.addEventListener("click", ()=>{
    console.log("<span>: Capturado el evento.");
});
```

Observando la salida mostrada en la Figura 7.3 se puede entender una cuestión muy importante.

```
<span>: Capturado el evento.  
<p>: Capturado el evento.  
<section>: Capturado el evento.  
>
```

Figura 7.3. Gestión del evento de abajo hacia arriba.

A pesar de que se han definido los eventos de fuera hacia dentro (**section**, **p** y **span**), los eventos se han ejecutado en orden inverso, de dentro hacia fuera (**span**, **p** y **section**). Esto ocurre porque ese es precisamente el comportamiento predeterminado de la propagación.

No obstante, puede迫使 el cambio de este comportamiento y hacer que la gestión de eventos se haga de fuera hacia dentro. Para ello, solo hay que recurrir al tercer parámetro de **addEventListener**, un valor booleano que establecido a **false** respeta el comportamiento por defecto, pero establecido a **true** invierte la lógica de ejecución:

```
miSection.addEventListener("click",()=>{  
    console.log("<section>: Capturado el evento.");  
},true);  
miP.addEventListener("click",()=>{  
    console.log("<p>: Capturado el evento.");  
}, true);  
miSpan.addEventListener("click",()=>{  
    console.log("<span>: Capturado el evento.");  
},true);
```

El resultado obtenido en esta ocasión es completamente el inverso (Figura 7.4).

```
<section>: Capturado el evento.  
<p>: Capturado el evento.  
<span>: Capturado el evento.  
>
```

Figura 7.4. Gestión del evento de arriba hacia abajo.

Esta inversión de la lógica de propagación puede resultar útil en algunas ocasiones, pero en otras se hace necesario establecer criterios en los que la propagación se produzca o deje de hacerlo para un mismo evento. Para ello, se aplica el método **stopPropagation()** sobre el objeto del evento. El programa reescrito queda de este modo:

```
let veces = 0;  
miSection.addEventListener("click",evento=>{  
    console.log('<section>: Capturado el evento ${veces} veces.');//  
});  
miP.addEventListener("click",evento=>{  
    console.log('<p>: Capturado el evento ${veces} veces.');//  
});  
miSpan.addEventListener("click",evento=>{  
    console.log('<span>: Capturado el evento ${veces} veces.');//  
});
```

```
veces++;  
console.log(`<span>: Capturado el evento ${veces} veces.`);  
if (veces > 1)  
    evento.stopPropagation();  
});
```

Ahora, la propagación se producirá solo la primera vez que se clica sobre el elemento. La segunda vez se ejecutará el método **stopPropagation()** y la propagación se cancelará en los elementos más superficiales del **span** (Figura 7.5).

```
<span>: Capturado el evento 1 veces.  
<p>: Capturado el evento 1 veces.  
<section>: Capturado el evento 1 veces.  
<span>: Capturado el evento 2 veces.  
<span>: Capturado el evento 3 veces.  
>
```

Figura 7.5. Evidencias de la cancelación de la propagación.

7.5. Cancelación de eventos

Para cancelar un evento se debe diferenciar entre dos escenarios: eventos predeterminados y eventos personalizados.

Existen algunos elementos que tienen asignado por defecto un evento de forma predeterminada. Por ejemplo, un elemento **a** (un enlace) tiene asociado por defecto el evento **click**, para navegar hasta donde indique cada vez que se haga clic sobre él. Para evitar que se comporte de esta manera, se puede capturar el evento y anular su comportamiento predeterminado, que es abrir la URL destino:

```
let miEnlace = document.getElementsByTagName("a")[0];  
miEnlace.addEventListener("click",evento=>{  
    evento.preventDefault();  
});
```

Como puede verse, se invoca a **preventDefault()** sobre el objeto asociado al evento, de manera que no realiza la acción que tiene definida de forma predeterminada, aunque es posible indicarle que realice otras acciones. Al ejecutar el código anterior, por mucho que se haga clic sobre el enlace, no se cargará la URL de destino, porque ese es su comportamiento predeterminado y se acaba de anular.

Sin embargo, lo que se ha hecho es cancelar el comportamiento por defecto, no el evento en sí. Para anular completamente un evento se recurre a **removeEventListener()**. Además, hay que tener en cuenta que la anulación no puede hacerse cuando la definición del evento se hace con una función anónima, solo es posible cuando se utilizan funciones con identificador. A continuación se presenta un ejemplo:

```
function miCallback(){  
    console.log("Evento anulado tras su primera ejecución");  
    miSpan.removeEventListener("click", miCallback);
```

```

}
let miSpan = document.getElementById("miSpan");
miSpan.addEventListener("click",miCallback);

```

En esta ocasión, tras el primer clic aparecerá el mensaje en la consola, se ejecutará `removeEventListener()` y retirará el evento al `span` seleccionado. De esta forma, a partir del segundo clic (incluido) no se ejecutará ningún gestor de eventos.

7.6. Lanzar eventos

Anteriormente se explicó que al desarrollar aplicaciones web se pueden capturar los eventos que, normalmente, lanzan los usuarios con sus acciones. Pero también, de forma programática, se pueden lanzar eventos propios. ¿Qué utilidad podría tener esta forma de proceder? Mucha, desde reutilizar gestores de eventos a simular acciones del usuario.

Para crear un evento y lanzarlo se utiliza `Event` y `dispatchEvent`, respectivamente. En el siguiente ejemplo se tienen dos elementos hermanos, un enlace y un `span`:

```

<a id="miEnlace" href="https://www.paraninfo.es">Enlace.</a>
<span id="miSpan">Ejecutar el evento del enlace.</span>

```

Y se desea definir un evento para cada elemento, de manera que el evento del `span` consiga lanzar el evento del enlace. Se haría lo siguiente:

```

let miSpan = document.getElementById("miSpan");
let miEnlace = document.getElementById("miEnlace");
miEnlace.addEventListener("click",(evento)=>{
    evento.preventDefault();
    console.log("Gestor del evento del enlace.");
});
miSpan.addEventListener("click",()=>{
    let miEvento = new Event("click");
    console.log("Lanzando evento.");
    miEnlace.dispatchEvent(miEvento);
});

```

La salida de la ejecución anterior es muy esclarecedora. Cuando se hace clic sobre el `span`, aparece su mensaje asociado, se dispara el evento creado sobre el enlace y se ejecuta el gestor de eventos del enlace (Figura 7.6).

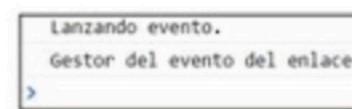


Figura 7.6. Un elemento lanzando un evento sobre otro elemento.

Llegados a este punto, se tiene el conocimiento necesario para poder trabajar con eventos en JavaScript. Lo que queda de unidad está dedicado a dar a conocer los tipos de eventos que pueden capturarse y cómo funcionan aquellos de más interés.

7.7. Tipos de eventos

Existe un amplio catálogo de eventos que pueden capturarse por medio de JavaScript. En esta sección se muestra una recopilación de los más comunes, aunque sería buena idea profundizar en su estudio a través de la documentación oficial del lenguaje.

7.7.1. Eventos de formulario

Los formularios representan las auténticas entradas de datos de las aplicaciones web. Más allá de capturar un clic en un elemento del DOM, existe todo un universo de controles para interactuar con el usuario.

Desde el punto de vista del DOM, todos los elementos de un formulario normalmente se encuentran anidados en el elemento `<form>`. Aun así, el objeto `document` incluye una propiedad especial, llamada `forms`, que contiene todos los formularios de un documento. De esta forma, el primer formulario que encuentra estará localizable en `forms[0]`, el segundo en `forms[1]` y así sucesivamente, por lo que no es necesario utilizar el habitual `document.getElementsByTagName("form")` para trabajar con ellos.

Recuerda

Los formularios contienen numerosos controles, cada uno con atributos, propiedades y comportamientos distintos. La sintaxis puede variar mucho de un control a otro, por lo que se recomienda repasar todo el código HTML relativo a ellos, así como las diferencias entre el envío por **GET** y por **POST**.



Cuando se invoca a `document.forms` se obtiene una colección de objetos `form`, el objeto de cada formulario. En la Tabla 7.2 se recopilan algunas de sus propiedades y métodos.

Tabla 7.2. Propiedades y métodos del objeto forms

Propiedad	Utilidad
<code>action</code>	Contiene la URL que recibirá los datos del formulario.
<code>elements</code>	Contiene todos los controles del formulario.
<code>length</code>	Número de controles del formulario.
<code>method</code>	{GET POST} en función del método elegido para enviarlo.
<code>enctype</code>	Tipo de codificación de los datos del formulario.
<code>acceptCharset</code>	Conjunto de caracteres del formulario.
<code>submit()</code>	Envía los datos del formulario a la URL de <code>action</code> usando <code>method</code> .
<code>reset()</code>	Devuelve el formulario a su estado inicial.
<code>onX</code>	Todos los eventos asociados al formulario, siendo <code>X</code> el nombre del evento: <code>onAbort</code> , <code>onBlur</code> , <code>onCancel</code> , <code>onClick</code> ...

Actividad propuesta 7.2

Enlaces como botones

Crea un formulario con un solo control, una dirección de correo electrónico. Además, incluye dos enlaces, uno para enviar el formulario y otro para reiniciarlo. Los enlaces no deben funcionar como enlaces, sino como botones que lancen los eventos **submit** y **reset**. Antes de permitir el envío del formulario es necesario comprobar que el correo electrónico introducido contiene una @.

A parte de los formularios como conjunto, cabe la posibilidad de trabajar individualmente sobre sus controles.

Los controles de un formulario son elementos HTML, y como tales tienen una serie de características comunes. Por ejemplo, se puede trabajar sobre sus atributos como se haría con cualquier otro elemento usando `getAttribute("valor")`. Sin embargo, es preciso tener en cuenta que los controles son dinámicos, por lo que usar un valor recogido antes de un cambio puede generar importantes errores en la lógica del programa. Por ello, siempre se recomienda utilizar las propiedades específicas de los controles de formulario, que sí asumen los cambios realizados. En la Tabla 7.3 se recogen algunas de las más comunes.

Tabla 7.3. Propiedades de los controles de un formulario

Propiedad	Utilidad
accept	Tipos de archivos que se permiten en un control de tipo file .
autocomplete	Si el valor del control puede ser autocompletado por el navegador.
name	Nombre del control.
type	Tipo de control.
value	Valor actual del control.
checked	true si el control está activado, false si no lo está.
defaultChecked	Valor predeterminado de la propiedad checked .
disabled	true si el control está deshabilitado, false si está habilitado.
hidden	El control es invisible para el usuario, pero no para el programador.
readonly	true si el control es de solo lectura (no modificable), false si no lo es.
required	true si proporcionarle un valor al control es obligatorio, false si no lo es.
maxLength	Anchura máxima del texto.
min	Valor mínimo para el control.
max	Valor máximo para el control.
pattern	Una expresión regular contra la que el valor del control es evaluado.

Propiedad	Utilidad
placeholder	Una pista para el usuario sobre lo que debe introducir en el control.
size	Tamaño inicial del control.
step	Tamaño del cambio en el valor de un control.
selectionStart	En una selección de texto la posición del primer carácter seleccionado.
selectionEnd	En una selección de texto, la posición del último carácter seleccionado.

Enviar el formulario

El evento más importante relacionado con los formularios, sin duda, es **submit**, que permite enviar los datos que contiene el formulario a su destino, es decir, la URL indicada en el atributo **action**.

Hay que tener cuidado en este punto con las validaciones de datos, pues hay que hacerlas antes de que se produzca el envío del formulario. Existen muchos *frameworks* de CSS que incorporan pequeñas utilidades JavaScript para realizar esto automáticamente, como Bootstrap, Materialize o Semantic UI. Pero si no se están usando, y los datos no se han validado, hay que cancelar el comportamiento predeterminado de **submit** para que los datos no se envíen. Por ejemplo, véase el siguiente formulario:

```
<form name="formulario" action="procesa.php" method="POST">
    Introduzca una edad mayor de 18 y menor de 65:
    <input type="text" name="edad" placeholder="Edad" value="" />
    <input type="submit" name="submit" value="Enviar" />
</form>
<div id="errores"></div>
```

La idea es validar que la edad sea mayor que 18 y menor que 65 antes de que los datos se envíen a su destino. Para ello, puede capturarse el evento **submit**, realizar la comprobación y si **edad** no cumple con las restricciones, cancelar el envío y mostrar un mensaje de error:

```
let formularioEdad = document.forms[0];
let errores = document.getElementById("errores");
formularioEdad.addEventListener("submit", (evento)=>{
    let edad = parseInt(document.forms[0].elements["edad"].value);
    if (edad<=18 || edad>=65) {
        evento.preventDefault();
        errores.innerHTML="La edad debe ser > 18 y < 65";
    }
});
```

Para conocer el valor que ha introducido el usuario en el campo **edad** se recurre al contenido del objeto `document.forms[0]`, que contiene una propiedad llamada `elements` cuyo contenido es una entrada por cada control del formulario, y cada uno de estos, a su vez, otra lista de propiedades como `value`.

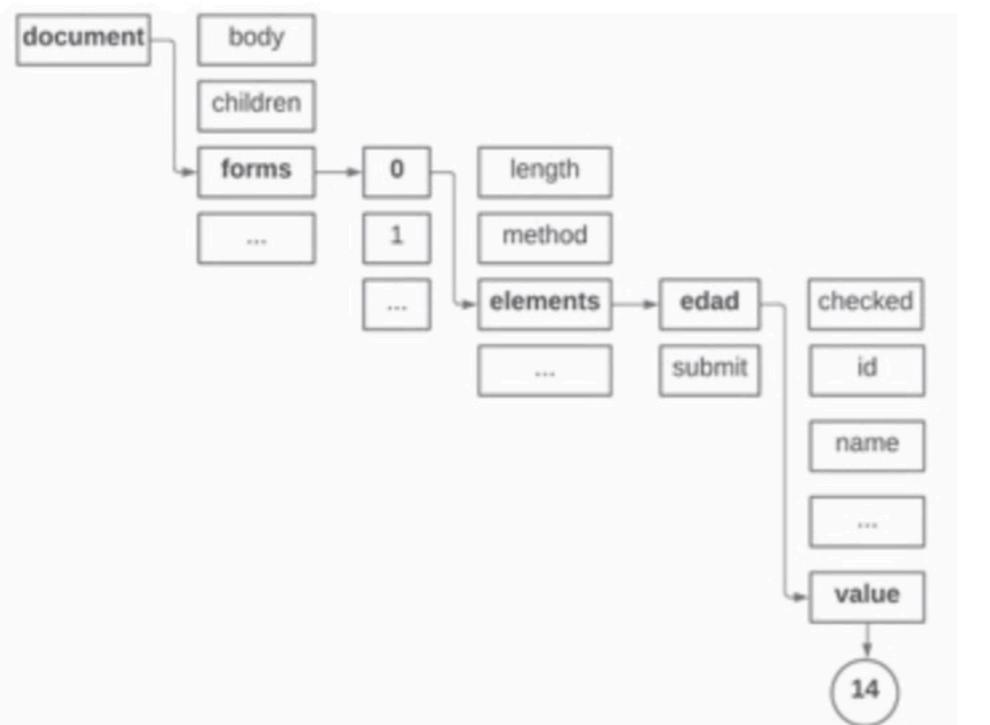


Figura 7.7. Diagrama de la cadena jerárquica de propiedades del objeto forms.

Nota técnica

Es posible que el lector se pregunte dónde es mejor hacer la validación de los datos proporcionados por el usuario, si en el cliente o en el servidor. Técnicamente puede hacerse en cualquiera de los dos lados; sin embargo, hacerlo en el lado cliente tiene muchas ventajas, como, por ejemplo, evitar problemas de seguridad al servidor, aligerar su carga de procesamiento, agilizar la respuesta al usuario, etc... Por tanto, siempre que se pueda, y salvo inconveniente muy específico del proyecto en cuestión, mejor validar en el cliente.



Reseteo del formulario

Resetear el formulario significa dejar el formulario en su estado de partida, con sus valores iniciales. Para lanzarlo o capturarlo debe invocarse al evento `reset`.

Cambiar un valor

Se trata de otro de los eventos de más utilidad. `change` se lanza cuando se realiza un cambio de valor en un control del formulario y, además, abandona el foco. Con esta colección hay que tener cuidado porque el evento no se lanza al cambiar el valor de un control, sino justo cuando el foco abandona el control que se ha cambiado.

Ganar y perder el foco

En el contexto de los formularios ganar el foco significa activar un control, bien a través del ratón, bien a través del teclado. Activar se refiere a poder interactuar directamente con el control. Una caja de texto, por ejemplo, gana el foco cuando a través del tabulador se llega a ella y está disponible para escribir dentro; o cuando al situarse sobre una casilla de verificación y se pulsa la barra espaciadora se selecciona o deselecciona el control. Igualmente, perder el foco significa todo lo contrario, abandonar la posibilidad de interactuar directamente con el control.

Cuando un control gana el foco se lanza el evento `focus`, y cuando lo pierde se lanza el evento `blur`.

Actividad resuelta 7.2

Campos obligatorios

Crea un formulario con tres controles, dos de ellos obligatorios, de manera que cuando cada control obligatorio gane el foco se muestre un mensaje indicando este hecho. Y cuando pierda el foco, elimine todos los mensajes mostrados al usuario.

Solución

HTML

```

<form name="formulario" action="procesa.php" method="POST">
  <input type="text" name="edad" placeholder="Edad" />
  <input type="text" name="email" placeholder="Email" required />
  <input type="text" name="tlf" placeholder="Teléfono" required />
  <input type="submit" name="submit" value="Enviar >" />
</form>
<div id="errores"></div>
<script src="eventos.js"></script>
  
```

JavaScript

```

let controles = document.forms[0].elements;
let numEltos = document.forms[0].length;
for (let i=0; i<numEltos; i++) {
  controles[i].addEventListener("focus",()=>{
    if (controles[i].hasAttribute("required")) {
      document.getElementById("errores").innerHTML = "Campo obligatorio";
    }
  });
  controles[i].addEventListener("blur",()=>{
    if (controles[i].hasAttribute("required")) {
      document.getElementById("errores").innerHTML = "";
    }
  });
}
  
```

7.7.2. Eventos de ratón

Los eventos de ratón son producidos no solo por el ratón, sino por cualquier dispositivo apuntador capaz de mover el cursor por la pantalla, como el dedo en una pantalla táctil. En la Tabla 7.4 se recogen todos los eventos disponibles.

Tabla 7.4. Eventos de ratón

Evento	Funcionamiento
click	Hacer clic sobre el botón principal del dispositivo.
dblclick	Hacer doble clic sobre el botón principal del dispositivo.
mousedown	Cuando se pulsa y justo antes de soltarlo, se lanza el evento.
mouseup	El evento se lanza cuando se suelta el botón.
mousenter	El evento se lanza cuando el puntero se sitúa sobre el elemento que capture el evento.
mouseleave	El evento se lanza cuando el puntero deja de situarse sobre el elemento que capture el evento.
mousemove	Mientras se está dentro del elemento, el evento se lanza cada vez que se mueve el puntero.
mouseover	El evento se lanza cuando el puntero se sitúa sobre el elemento que lo capture o sobre cualquiera de sus hijos.
mouseout	El evento se lanza cuando el puntero deja de situarse sobre el elemento que lo capture o sobre cualquiera de sus hijos.
contextmenu	El evento se lanza cuando se solicita un menú contextual.

Actividad resuelta 7.3

Elementos coloreados

Crea un programa que a partir de una nutrida variedad de elementos HTML los coloore con colores aleatorios cada vez que se coloque el ratón sobre ellos, y los vuelva a colorear de blanco cuando el ratón los abandone.

Solución

HTML

```
<p>
    Contenido de un párrafo con un <span>span</span>.
</p>
<ul>
    <li>Primer elemento</li>
    <li>Segundo elemento.</li>
```

```
</ul>
<a href="#">Enlace</a>
<section>
    Nueva sección
</section>
<h2>Subtítulo h2 de la sección</h2>
<script type="text/javascript" src="eventos.js"></script>
```

JavaScript

```
let elementos = document.getElementsByTagName("*");
for (let i=5; i<elementos.length; i++) {
    elementos[i].addEventListener("mouseenter",()=>{
        let r = Math.floor(Math.random()*255);
        let g = Math.floor(Math.random()*255);
        let b = Math.floor(Math.random()*255);
        elementos[i].style.backgroundColor='rgb(${r},${g},${b})';
    });
    elementos[i].addEventListener("mouseleave",()=>{
        elementos[i].style.backgroundColor="#FFFFFF";
    });
}
```

Hay que recordar también que la propiedad **button**, vista en el Apartado 7.3, recoge el botón del ratón que fue pulsado.

Actividad propuesta 7.3

Sin menú contextual

Crea un programa que no permita que se abra el menú contextual sobre ningún elemento de la web.

7.7.3. Eventos de teclado

Al trabajar con eventos de teclado, el objeto de evento contiene información que puede resultar de mucho interés. Por ejemplo, una de las propiedades disponibles en el objeto es **key**, que almacena información de la tecla pulsada, con independencia de si se pulsaron directamente, o en combinación con [Mayús], [AltGr], [Ctrl] o cualquier otra.

Para comprobar si se pulsó alguna de ellas, también puede revisarse el contenido de las propiedades **AltKey**, **CtrlKey**, **ShiftKey** o **MetaKey**, esta última dedicada a la tecla especial de los ordenadores Apple.

Además, también podría resultar de interés cuando existen varias teclas para la misma función (como [Mayús] o [Ctrl]) en distintas zonas del teclado, cuál de ellas se pulsó. Para ello, se puede recurrir a la propiedad **location**, cuyos valores pueden ser los que se muestran en la Tabla 7.5.

Tabla 7.5. Posibles valores de la propiedad location del teclado

Valor	Funcionamiento	Constante asociada
0	Teclado estándar	DOM_KEY_LOCATION_STANDARD
1	Parte izquierda	DOM_KEY_LOCATION_LEFT
2	Parte derecha	DOM_KEY_LOCATION_RIGHT
3	Teclado numérico	DOM_KEY_LOCATION_NUMPAD

En la Tabla 7.6 se recogen los tres eventos existentes para gestionar el comportamiento del usuario con respecto al teclado.

Tabla 7.6. Eventos lanzados tras la pulsación de una tecla

Evento	Funcionamiento
keypress	El evento se lanza tras pulsar y soltar una tecla.
keydown	El evento se lanza tras pulsar y antes de soltar la tecla.
keyup	El evento se lanza tras soltar la tecla.

El siguiente ejemplo muestra los eventos de la Tabla 7.6 en funcionamiento.

```
let todoBody = document.getElementsByTagName("body")[0];
todoBody.addEventListener("keypress", (evento)=>{
    console.log(`Lanzado KEYPRESS con la tecla ${evento.key}`);
});
todoBody.addEventListener("keydown", (evento)=>{
    console.log(`Lanzado KEYDOWN con la tecla ${evento.key}`);
});
todoBody.addEventListener("keyup", (evento)=>{
    console.log(`Lanzado KEYUP con la tecla ${evento.key}`);
});
```

Lo único que se ha hecho es asociar los tres eventos al elemento **body** para que estén disponibles en todo el cuerpo del documento. Cada vez que se pulse una tecla aparecerán en consola los mensajes lanzados por cada evento asociado:

Lanzado KEYDOWN con la tecla d
Lanzado KEYPRESS con la tecla d
Lanzado KEYUP con la tecla d

Figura 7.8. Eventos lanzados tras la pulsación de la tecla [d].

7.7.4. Eventos de arrastrar y soltar

Son eventos que se lanzan al utilizar elementos definidos como arrastrables, es decir, que tengan establecido a **true** el atributo **draggable**. Al tratarse de una acción con un origen y un posible destino, deben diferenciarse los dos conjuntos de eventos (Tabla 7.7).

Tabla 7.7. Eventos asociados a la acción *drag&drop*

Evento	Funcionamiento
Elemento que se arrastra	
drag	Cada vez que se arrastra una vez que se ha iniciado el arrastre.
dragstart	Al comenzar a arrastrar el elemento.
dragstop	Al finalizar el arrastre del elemento.
Elemento donde puede soltarse	
dragenter	Cuando el elemento que se arrastra entra en el elemento destino.
dragleave	Cuando el elemento que se arrastra sale del elemento destino.
dragover	Cada vez que continúa el arrastre sobre el elemento de destino.
drop	Cuando el elemento que se arrastra se suelta sobre el elemento destino.

El siguiente ejemplo muestra el uso de estos eventos:

```
<ul id="origen">
    <li draggable="true" id="origen1">Primer elemento</li>
    <li draggable="true" id="origen2">Segundo elemento</li>
    <li draggable="true" id="origen3">Tercer elemento</li>
</ul>
<div id="destino"></div>
<script src="eventos.js"></script>
```

Esta pieza de código HTML tiene una lista desordenada con tres elementos arrastrables. Lo que se pretende es que el usuario pueda arrastrar cada uno de los elementos a la caja (**<div>**) que está debajo. Cada vez que se arrastre un elemento y se suelte sobre la caja, se añadirá al final de la lista de elementos que contenga. Este es el código JavaScript necesario para conseguirlo:

```
let elementosOrigen = document.querySelectorAll("#origen > li");
let destino = document.getElementById("destino");
elementosOrigen.forEach(function(elemento){
    elemento.addEventListener("dragstart", (evento)=>iniciadoArrastre(evento));
});
destino.addEventListener("dragover", (evento)=>permitirSoltar(evento));
destino.addEventListener("drop", (evento)=>soltar(evento));

function iniciadoArrastre(evento) {
    evento.dataTransfer.setData("idElementoOrigen", evento.target.id);
}
function permitirSoltar(evento) {
    evento.preventDefault();
}
function soltar(evento) {
    evento.preventDefault();
    let elementoArrastrandose = evento.dataTransfer.getData("idElementoOrigen");
    destino.appendChild(document.getElementById(elementoArrastrandose));
}
```

Lo primero que se hace es guardar en variables todos los elementos que van a intervenir. Por un lado, se seleccionan todos los elementos de la lista (guardados en **elementosOrigen**) y, por otro, el elemento sobre el que se van a soltar (**destino**). A continuación, se asocia el evento **dragstart** a cada uno de los elementos de la lista y se guarda su **id** en un nuevo identificador «transitorio» denominado **idElementoOrigen**, que se utilizará cuando haya que soltarlo en el destino. Ahora, se asocia al elemento de **destino** el evento **dragover** para detectar cuándo el elemento que se está arrastrando entra en su área de influencia. Por último, se programa el evento **drop**, de manera que recupere el identificador transitorio del elemento que se está arrastrando y lo añada al final del contenido del elemento **destino**.

7.7.5. Eventos de reproducción multimedia

Son aquellos eventos que pueden capturarse cuando se reproducen contenidos multimedia, sean del tipo que sean. En la Tabla 7.8 se recogen los más útiles.

Tabla 7.8. Eventos asociados a la reproducción de contenidos multimedia

Evento	Momento en el que se activa
canplay	Cuando se detecta que el contenido se puede comenzar a reproducir.
canplaythrough	Cuando se estima que el contenido tiene cargados datos suficientes como para poder reproducirse.
durationchange	Cuando se modifica el atributo duration del medio.
emptied	Cuando se vacía de datos el medio.
ended	Cuando se detiene la reproducción, sin intervención del usuario.
loadeddata	Cuando se ha cargado la primera imagen de un vídeo.
loadedmetadata	Cuando se han cargado los metadatos del medio.
pause	Cuando se pausa el medio de reproducción.
play	Cuando se reanuda la reproducción tras una pausa.
playing	Cuando el medio está listo para reproducirse tras una parada.
ratechange	Cuando se modifica el rate del vídeo en reproducción.
seeked	Cuando finaliza la búsqueda en el medio.
seeking	Cuando se inicia la búsqueda en el medio.
stalled	Cuando se produce un fallo en la carga, pero la carga continúa.
suspend	Cuando se suspende la carga del medio.
timeupdate	Cuando se modifica el valor de currentTime del medio.
volumechange	Cuando se modifica el volumen.
waiting	Cuando la reproducción se detiene por falta de datos.

7.7.6. Otros eventos

Aquí se recopilan otros muchos eventos que podrían resultar de interés, pero que por su tamaño no merecen una sección propia.

Tabla 7.9. Otros eventos que pueden capturarse con JavaScript

Evento	Momento en el que se activa
Carga de elementos	
abort	Cuando se cancela la carga de un elemento.
DOMContentLoaded	Cuando se ha cargado el documento HTML.
error	Cuando se produce un error en la carga.
load	Cuando se ha cargado el documento y los elementos externos que incorpora: imágenes, hojas de estilo...
progress	Cuando se está produciendo la carga.
readystatechange	Cuando se modifica el valor del atributo readystate .
Historial de la sesión	
popstate	Cuando se cambia el historial.
pagehide	Cuando se esconde la página actual mientras se visualizan las distintas páginas de la sesión.
pageshow	Cuando el navegador muestra el documento en la ventana.
Ventana	
scroll	Cuando se desplaza la ventana por medio de las barras de desplazamiento.
resize	Cuando se modifica el tamaño de la ventana.
Animación	
animationend	Cuando finaliza la animación.
animationiteration	Cuando se repite la animación.
animationstart	Cuando comienza la animación.
Impresión	
afterprint	Cuando ha comenzado la impresión o se ha cerrado la previsualización de la impresión.
beforeprint	Cuando va a comenzar la impresión o se ha abierto la previsualización de la impresión.
Portapapeles	
copy	Cuando se va a copiar justo antes de ejecutar la acción.
cut	Cuando se va a cortar justo antes de ejecutar la acción.
paste	Cuando se va a pegar justo antes de ejecutar la acción.

Actividad resuelta 7.4

Prohibido pegar

Crea un programa que pida una dirección de correo electrónico y la repetición del correo electrónico, pero que no permita que la segunda dirección se pegue, sino que se escriba de nuevo.

Solución

HTML

```
<form name="formulario" action="procesa.php" method="POST">
    <input type="text" id="email" name="email" placeholder="Email" />
    <input type="text" id="repemail" name="emailconf" placeholder="Repite Email" />
    <input type="submit" name="submit" value="Enviar >" />
</form>
<script src="eventos.js"></script>
```

JavaScript

```
let elemento = document.getElementById("repemail");
elemento.addEventListener("paste", (ev) =>{
    ev.preventDefault();
});
```

Estos son los eventos más comunes con los que se puede interactuar, aunque no son todos. Existen eventos relacionados con la batería del dispositivo, las llamadas telefónicas, las bases de datos, los menús, la red, las notificaciones, etc. Es recomendable repasar la documentación oficial y descubrir todas las posibilidades al alcance del programador.

Para saber más

Con el siguiente enlace o con el código QR se accede a una web que muestra una clasificación de todos los eventos disponibles para gestionar con JavaScript:

<https://developer.mozilla.org/es/docs/Web/Events#category%C3%ADas>



Importante

Cuando se asocian gestores de eventos a elementos del DOM se está haciendo a los elementos que están presentes en ese instante. Pero el DOM es una estructura dinámica que puede cambiarse según las necesidades. Es preciso tener mucha precaución a la hora de definir eventos para una serie de elementos, porque si en ese conjunto de elementos se inserta uno de forma dinámica **no tendrá asociado el evento que se había definido**.

