TEMA 8 – ITERACCIÓN CON EL USUARIO: EVENTOS

Cuando interaccionamos con el usuario se tenía la **limitación** del uso de ciertos **cuadros de diálogos ofrecidos por objetos como window o document**, en los que ya estaba predefinido el comportamiento de sus botones, como puede ser alert, que muestra un mensaje con un botón “aceptar” o prompt, que solicita al usuario que introduzca un valor mostrando un cuadro con un campo de texto y botones para “aceptar” y “cancelar”… Pero en las aplicaciones web estos métodos rara vez se utilizan. **JavaScript proporciona una vía muy ambia mucho más potente, personalizable y atractiva de interactuar con el usuario: la gestión de eventos.**

Los eventos y los manejadores (handlers) son elementos para capturar interacciones del usuario y realizar acciones en respuesta a estas interacciones. 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 es que este mecanismo es que es asíncrono, por lo tanto, no hay que esperar por el. Simplemente se prepara el código para que cuando se produzca el evento el programa lo capture y ejecute la lógica que le interese.

1. Captura de eventos:

Los **eventos** se asocian a elementos concretos del DOM, los programadores “escuchan” a ese elemento y **tienen preparado un programa para capturar el evento que lanza ese elemento.** Algunas de las características principales de los eventos son:

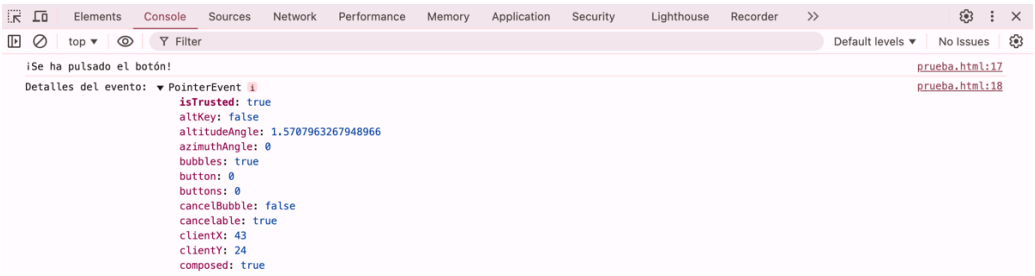
- Son disparados por el navegador o por el usuario.
- Pueden estar relacionados con elementos específicos del DOM, como un botón o un campo de entrada.
- El objeto Event contiene información detallada sobre el evento que ocurrió, como qué elemento lo originó y detalles específicos del tipo de evento.

Un **manejador**, o handler en inglés, es una función que se ejecuta cuando ocurre un evento específico. Cada tipo de evento (como click, submit, mouseover, etc.) puede tener asociado un manejador que define qué acción debe realizarse en respuesta al evento.

- Es una función que se asigna a un evento particular en un elemento del DOM.
- Define la acción o comportamiento que se debe llevar a cabo cuando el evento ocurre.
- Los manejadores pueden ser definidos directamente en el HTML, usando atrobutos como onclik, o pueden ser asignados dinámicamente desde JS.

Si queremos capturar los eventos directamente desde JS 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.

```
<button id="boton1">Haz clic aquí</button>
<script>
    const button = document.getElementById('boton1');

    // Definir el manejador para el evento 'click'
    function handleClick(event) {
        console.log('¡Se ha pulsado el botón!');
        console.log('Detalles del evento:', event);
    }

    // Asignar el manejador al evento 'click' del botón
    button.addEventListener('click', handleClick);
</script>
```



Algunas de las **propiedades** del objeto evento son:

Propiedad	Utilidad
altKey	Devuelve si la tecla [Alt] fue pulsada durante el evento.
Button	Devuelve el botón del ratón que activó el evento: 0: principal, 1: botón central, 2: secundario…

Target	Referencia al elemento que lanzó el evento.
Type	Nombre del evento
timeStamp	Devuelve el momento en el que se creó el evento
CharCode	Contiene el valor Unicode de la tacla que se pulsó (evento keypress).
ctrlKey	Devuelve si la recla [Ctrl] fue pulsada durante el evento.

Por ejemplo:

```
console.log('Detalles del evento:', event);
console.log("AltKey: "+ event.altKey);
console.log("Button: " + event.button);
console.log("Target: " + event.target);
console.log("TimeStamp: " + event.timeStamp);
console.log("Type: "+ event.type);
```

```
Detalles del evento:                                    prueba.html:18
▶ PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressur
  e: 0, …}
AltKey: false                                           prueba.html:19
Button: 0                                               prueba.html:20
Target: [object HTMLButtonElement]                      prueba.html:21
TimeStamp: 1033.5999999940395                           prueba.html:22
Type: click                                             prueba.html:23
```

A continuación vamos a ver un ejemplo de introducir el código JS directamente en el atributo onmouseover y onmouseout del elemento <p>. Aunque es simple, no se recomienda porque mezcla lógica de presentación con el contenido y dificulta el mantenimiento.

```
<p onmouseover="this.style.background='#FF0000';" onmouseout="this.style.background='#FFFFFF';">HOLA</p>
```

Cuando pasamos el ratón por encima del span ocurre el evento:

HOLA

El ejemplo que os voy a mostrar a continuación, utiliza DOMContentLoaded, el cual es un evento que es fundamental en la gestión del DOM (Document Object Model). Esto garantiza que el código JS que manipula elementos del DOM se ejecute solo después de que la estructura básica de la página se haya cargado completamente.

Cuando se carga una página web, el navegador realiza dos tareas principales:

1. Cargar el HTML y construir el DOM.
2. Cargar recursos externos (como imágenes, hojas de estilo y script).

El evento **DOMContentLoaded** se dispara cuando el navegador ha terminado de construir el DOM, pero antes de cargar completamente los recursos externos. Esto significa que podemos empezar a manipular elementos del DOM sin esperar a que la página esté completamente cargada (ahorrando tiempo). Si por el contrario, intenmos manipular elmenetos del DOM antes de que el DOM este listo, obtendríamos un error de tipo: "Cannot read property addEventListener of null." Esto ocurre porque el elemento con id="hola" no existiría aún en el momento en que el script intenta acceder a él.

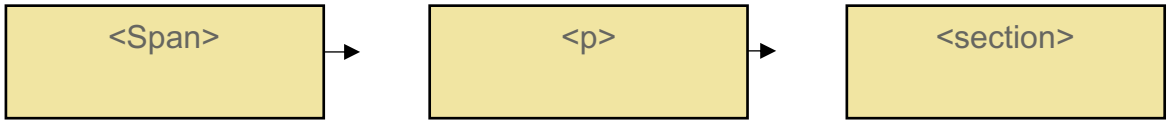
```
<script>
    document.addEventListener("DOMContentLoaded", function () {
        document.getElementById('hola').addEventListener('mouseover', manejador, false);
        document.getElementById('hola').addEventListener('mouseout', manejador, false);
    });

    function manejador(e) {
        console.log(e.type, e.target);
        if (e.type === 'mouseover') {
            this.style.background = '#FF0000';
        }
        if (e.type === 'mouseout') {
            this.style.background = '#FFFFFF';
        }
        if (e.target.id === 'hola') {
            console.log('¡Hola!');
        }
    }
</script>
```

2. 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.

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



Padre Abuelo

```html
<script>
    const miSeccion = document.getElementsByTagName("section")[0];
    const mip = document.getElementsByTagName("p")[0];
    const miSpan = document.getElementById("miSpan");

    miSeccion.addEventListener("click", function () {
        console.log("<section>: Capturado el evento");
    });

    mip.addEventListener("click", function () {
        console.log("<p>: Capturado el evento");
    });

    miSpan.addEventListener("click", function () {
        console.log("<span>: Capturado el evento");
    });

</script>
```

Si observamos la salida que muestra:

```
<span>: Capturado el evento                    prueba.html:30
<p>: Capturado el evento                       prueba.html:26
<section>: Capturado el evento                 prueba.html:22
```

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 es precisamente el comportamiento predeterminado de la programación.

No obstante, puede forzarse el cambio de 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 a fals

```
// Permitir el drop en la papelera y prevenir el comportamiento por defecto
papelera.addEventListener("dragover", (e) => {
    e.preventDefault();
});
```

e respeta el comportamiento por defecto, pero establecido a true invierte la lógica de ejecución:

```
miSeccion.addEventListener("click", function () {
    console.log("<section>: Capturado el evento");
}, true);

mip.addEventListener("click", function () {
    console.log("<p>: Capturado el evento");
},true);

miSpan.addEventListener("click", function () {
    console.log("<span>: Capturado el evento");
},true);
```

```
<section>: Capturado el evento                    prueba.html:22
<p>: Capturado el evento                          prueba.html:26
<span>: Capturado el evento                       prueba.html:30
```

Esta inversión de la lógica de propagación puede resultar útil pen algunas ocasiones, pero en otras se hace necesario establecer criterios en los que la propagación se produzca o se deje de hacerlo para un mismo evento. Para evitar que un evento se propague por el árbol del DOM, puedo utilizar el método **StopPropagation**, que detiene el evento en el elemento actual y evita que continue hacia los elementos padres o hacia los elementos relacionados.

```
miSeccion.addEventListener("click", function () {
    contadorSeccion++;
    console.log("<section>: Capturado el evento y se ha pulsado " + contadorSeccion + " veces");
});

mip.addEventListener("click", function (e) {
    contadorParrafo++;
    console.log("<p>: Capturado el evento y se ha pulsado " + contadorParrafo + " veces");
    e.stopPropagation(); // Detenemos la propagación para que no llegue al párrafo o la sección

});

miSpan.addEventListener("click", function (e) {
    contadorSpan++;
    console.log("<span>: Capturado el evento y se ha pulsado " + contadorSpan + " veces");
    e.stopPropagation(); // Detenemos la propagación para que no llegue al párrafo o la sección

});
```

```
<p>: Capturado el evento y se ha pulsado 1 veces          prueba.html:34
<span>: Capturado el evento y se ha pulsado 1 veces       prueba.html:41
<p>: Capturado el evento y se ha pulsado 2 veces          prueba.html:34
```

3. Cancelación de eventos:

A la hora de gestionar eventos existen algunos **elementos** que tienen asignado por defecto un **evento de forma predeterminada**. Por ejemplo, un elemento <a> 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 esa manera, se puede capturar el evento y anular su comportamiento predeterminado, que es abrir la URL destino.

```
<body>
    <a href="https://www.google.com">Ir a Google</a>
</body>
<script>
    let miEnlace = document.getElementsByTagName("a")[0];
    miEnlace.addEventListener("click", function(e){
        e.preventDefault();
        console.log("El evento se ha producido");
    })
</script>
</html>
```

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, por ese su comportamiento predeterminado se acaba de anular.

Sin embargo, lo que se ha hecho es **cancelar el comportamiento por defecto**, no el evento en sí. **Para anular el evento completamente 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.

```html
<a href="https://www.google.com">Ir a Google</a>

<script>
    let miEnlace = document.getElementsByTagName("a")[0];

    function cb(event) {
        event.preventDefault(); // Evita el comportamiento por defecto del enlace
        console.log("Evento anulado en su primera ejecución");
        miEnlace.removeEventListener("click", cb); // Elimina el evento después de ejecutarse
    }

    miEnlace.addEventListener("click", cb);
</script>
```

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

4. Lanzar eventos:

En el desarrollo de aplicaciones web, además de capturar eventos lanzados por las acciones del usuario, también es posible crear y disparar eventos personalizados. Esto resulta útil para:

- Reutilizar **gestores de eventos** en diferentes situaciones.
- Simular **acciones del usuario** en pruebas o automatizaciones.
- Desencadenar procesos en componentes desacoplados.

Para **crear un evento y lanzarlo se utiliza el objeto Event y dispachEvent**, respectivamente. En el siguiente ejemplo se puede observar:

```html
<button id="btn-usuario">Clic Usuario</button>
<button id="btn-simular">Simular Evento</button>

<script>
    const botonUsuario = document.getElementById("btn-usuario");
    const botonSimular = document.getElementById("btn-simular");

    botonUsuario.addEventListener("miEventoPersonalizado", function () {
        alert("¡Evento personalizado activado!");
    });

    botonSimular.addEventListener("click", function () {
        const eventoPersonalizado = new Event("miEventoPersonalizado");
        botonUsuario.dispatchEvent(eventoPersonalizado); // Lanza el evento
    });
</script>
```

5. Tipos de eventos:

Existe un amplio catálogo de eventos que pueden capturarse por medio de JS. En esta sección se muestra una recopilación de los más comunes.

5.1. Eventos de formulario:

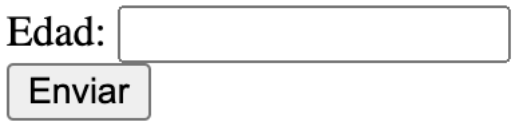
Los formularios representan las auténticas entradas de datos de las aplicaciones web. Más allá de capturar un clic en un elemento del DOM, existe todo un universo de controles para interactuar con el usuario.

Desde el punto de vista del DOM, todos los elementos de un formulario 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 documentos. De esta forma, el primer formulario que encuentra estará **localizable** en forms[0], el segundo en forms[1] y así sucesivamente. Cuando se invoca a **document.forms** se obtiene una colección de objetos forms con los objetos de cada formulario.

```
<body>
    <form id="formulario-ejemplo">
        <label for="campo1">Campo 1:</label>
        <input type="text" id="campo1" name="campo1">
        <br>
        <label for="campo2">Campo 2:</label>
        <input type="text" id="campo2" name="campo2">
        <br>
        <button type="submit">Enviar</button>
    </form>

    <script>
        document.addEventListener("DOMContentLoaded", function () {
            // Accedemos al primer formulario de la página
            const formulario = document.forms[0];
            console.log("Primer formulario de la página:", formulario);

            // Acceso a los elementos dentro del formulario
            const campo1 = formulario.elements['campo1'];
            const campo2 = formulario.elements['campo2'];
            console.log("Campo 1:", campo1);
            console.log("Campo 2:", campo2);
        });
    </script>
```

El evento más importante relacionado con los formularios, sin duda, es **submit**, que permite **enviar los datos que contiene el formulario a su destino**. **Hay que tener cuidado en este punto con las validaciones de datos, pues hay que hacer las ante de que se produzca el**

envío del formulario. Existen muchos framework de CSS, que incorporan pequeñas utilidades JS para realizar esto automáticamente como Bootstrap, 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:

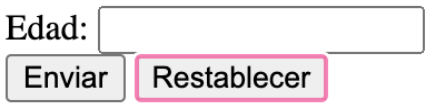


La idea es **validar** que la edad sea mayor que 18 y menor que 65 antes de que los dato se envíen a u destino. **Para ello, puede capturarse el evento submit, realizar la comprobación y si la edad no cumple con las restricciones, cancelar el envío y mostrar un mensaje de error:**

```
document.addEventListener("DOMContentLoaded", function () {
    const formulario = document.getElementById("formulario-validacion");

    formulario.addEventListener("submit", function (event) {
        const edad = formulario.elements['edad'].value;

        if (edad < 18 || edad > 65) {
            event.preventDefault(); // Cancela el envío
            alert("La edad debe estar entre 18 y 65 años.");
        } else {
            alert("Formulario enviado con éxito.");
        }
    });
});
```

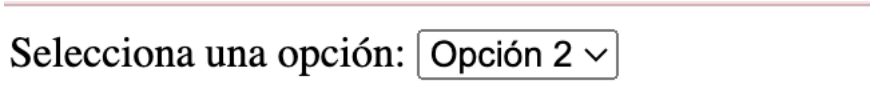
Para conocer el valor que ha introducido el usuario en el campo edad se recurre al contenido del objeto documents.forms[0], que contiene una propiedad **elements cuyo contenido es una entrada por cada control del formulario, y cada uno de estos, a su vez, otra lista de propiedades como value.**

Otro evento que se puede aplicar sobre los formularios es el de dejar el formulario en su estado de partida con sus valores iniciales. Para lanzarlo o capturarlo debe incorporarse el evento **reset**.

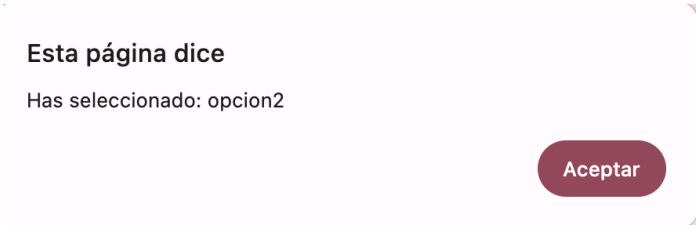



```
formulario.addEventListener("reset", function () {
    alert("Formulario restablecido a sus valores iniciales.");
});
```

Otro evento de gran utilidad es change, que se lanza cuando se realiza un cambio de valor en un control del formulario y, además, abandona el foco. Con **esta coletilla hay que tener cuidado por el evento no se lanzará al cambiar el valor de un control sino junto cuando el foco abandona el control que se ha cambiado.**




```
<form id="formulario-change">
    <label for="opcion">Selecciona una opción:</label>
    <select id="opcion" name="opcion">
      <option value="opcion1">Opción 1</option>
      <option value="opcion2">Opción 2</option>
      <option value="opcion3">Opción 3</option>
    </select>
  </form>

  <script>
    document.addEventListener("DOMContentLoaded", function () {
      const select = document.getElementById("opcion");

      select.addEventListener("change", function () {
        alert(`Has seleccionado: ${select.value}`);
      });
    });
  </script>
```



Y por último, un evento que ya hemos visto en clase es el evento **focus** cuando se gana el foco sobre un elemento y el evento **blur** cuando se pierde el foco.

5.2. Eventos de arrastrar y soltar:

Los navegadores permiten una serie de características para gestionar la funcionalidad de arrastrar y soltar. Esto permite al usuario hacer **clic** y mantener presionado el botón del ratón/mouse sobre un elemento, **arrastrarlo a otra ubicación y soltarlo para colocar el elemento allí.** Al puntero le seguirá una representación transparente de lo que se está arrastrando durante la operación.

Cuando comienza una **operación** de **arrastre y soltar**, se puede proporcionar una serie de datos:

- Los **datos** que se van a **arrastrar**, que pueden ser de varios formatos, (texto, imágenes, archivos, etc..)
- Es una **imagen** que aparece junto al puntero durante la operación. Puede ser personalizada por el desarrollador o si no se especifica, el navegador genera una imagen por defecto basada en el elemento que se está arrastrando.
- **Efectos de arrastre, que indicarán lo que ocurrirá al finalizar la operación de arrastre.** Los efectos posibles son:
    - **Copy:** para indicar que los datos que se arrastran se copiarán desde su ubicación actual a la ubicación de destino
    - **Move:** para indicar que los datos que se arrastran serán movidos a una nueva ubicación.
    - **Link:** para indicar que se creará algún tipo de relación o conexión entre la ubicación actual y la ubicación de destino.

El desarrollador puede modificar estos efectos durante la operación y definir qué acciones están permitidas en cada ubicación de destino. **Se utilizan una serie de eventos que se ejecutan durante las diversas etapas de la operación de arrastre y colocación**. Ten en cuenta que se ejecutan solo los eventos de arrastre, los eventos del ratón como **mousemove** no se ejecutan durante una operación de arrastre.

- **Dragstart**: Se dispara al comenzar la operación de arrastre. Es el momento adecuado para:
  - Configurar los datos que se van a transferir.
  - Personalizar la imagen que aparecerá junto al puntero.
- **Dragenter**: Se dispara cuando el puntero entra por primera vez en un elemento de destino válido. Sirve para mostrar visualmente que la ubicación acepta el elemento arrastrado. Por ejemplo, resaltar un área.
- **Dragover**: Se dispara mientras el puntero está sobre el elemento de destino durante la operación de arrastre. Es necesario evitar el comportamiento predeterminado con e.preventDefault() para que el evento drop se pueda disparar.
- **Dragleave**: Se dispara cuando el puntero abandona el área de un elemento de destino. Es útil para deshacer cambios realizados en el dragenter.
- **Drag**: Se dispara en el elemento de origen mientras se realiza la operación de arrastre.
- **Drop**: Se dispara en el elemento de destino cuando se suelta el elemento arrastrado. Es el momento adecuado para:
  - Obtener los datos transferidos por e.dataTrandfer.getData
  - Insertar o manejar el contenido en el destino.
- **Dragend:** Se dispara en el elemento de origen al finalizar la operación de arrastre (ya sea porque tuvo éxito o porque fue cancelada).

El objeto **DataTransfer** se **utiliza para almacenar y gestionar los datos que se transfieren durante una operación de arrastrar y soltar**. Este objeto puede contener uno o más elementos de datos, cada uno asociado con uno o varios tipos de datos. Algunas de las propiedades que podemos utilizar en un objeto de tipo DataTransfer son:

- **DataTransfer.dropeffect.** Obtiene o establece el tipo de operación de arrastrar y soltar actualmente seleccionada.

    - **None:** No se permite ninguna acción.
    - **Copy:** Copia los datos de la ubicación de destino.
    - **Link:** Crea una conexión o enlace entre origen y destino.
    - **Move:** Mueve los datos al destino.
- **DataTransfer.effectAllowed.** Proporciona todos los tipos de operaciones posibles. Defe ser uno de los siguientes none, copy, copylink, copyMove, link, linkMove, move, all o uninitialized.
- **DataTransfer.files.** Contiene una lista de todos los archivos locales disponibles en la transferencia de datos. Si la operación de arrastre no implica arrastrar archivos, esta propiedad es una lista vacía.
- **DataTransfer.items.** Genera una DataTransferItemList objeto que es una lista de todos los elementos de arrastre. Este objeto es de solo lectura.
- **DataTransfer.types.** Una matriz de strings que tiene los formatos que se establecieron en el dragstart evento. Este objeto es de solo lectura.

Algunos de los **métodos** que podemos utilizar en un objeto de tipo **DataTransfer** son:

- **DataTransfer.clearData().** Elimina los datos asociados con un tipo determinado. El argumento de tipo es opcional:
    - Si no se especifican, se eliminan los datos asociados con todos los tipos.
    - Si no existen datos para el tipo especificado, o la transferencia de datos no contiene datos, este método no tiene ningún efecto.
- **DataTransfer.getData().** Recupera los datos de un tipo determinado o una cadena vacía si los datos de este tipo no existen o la transferencia de datos no contiene datos.
- **DataTransfer.setData().** Permite configurar los datos para un tipo determinado.
    - Si los datos para el tipo no existen, se agregan al final, de modo que el último elemento de la lista será el nuevo formato.
    - Si ya existen datos para el tipo, son reemplazados en la misma posición.
- **DataTransfer.setDragImage().** Configura la imagen que se usa para arrastrar si queremos una personalizada.

A continuación, vamos a realizar un **ejemplo** sobre el evento de arrastrar y soltar (drag and drop) en la que un usuario puede mover una imagen de una ubicación inicial a una ubicación destino, y al soltarlo se cambie de imagen automáticamente.


```html
<body>
    <h1>Probando el arrastre</h1>
    <img id="cerilla" class="cerilla" src="cerilla.jpg" draggable="true">
    <h1>Vamos a quemar los papeles</h1>
    <img id="papelera" class="papelera" src="papelera.jpg" alt="papelera">
    <script src="quemar.js"></script>
</body>
```

Nos debemos fijar en que la imagen de la cerilla tiene la propiedad **draggable** activada, eso quiere decir, que la cerilla se puede arrastrar. Cuando se hace click en al cerilla, cambia la imagen para simular que está encendida:

```javascript
cerilla.addEventListener("click", () => {
    cerilla.src = "cerillaEncendida.jpg";
});
```

Cuando se comienza a arrastrar la cerilla, se establece un dato en el objeto DataTransfer que se usará durante la operación de arrastre.

```
cerilla.addEventListener("dragstart", (e) => {
    e.dataTransfer.setData("image/jpeg", "cerilla"); // Establecer datos para el arrastre
});
```

Por defecto, no se puede soltar nada en un área destino. Para habilitarlo, debemos evitar el comportamiento por defecto en el evento dragover.

```
papelera.addEventListener("dragover", (e) => {
    e.preventDefault();
});
```

Cuando la cerilla se suelta en la papelera, cambian ambas imágenes:

- La imagen de la papelera cambia a una con fuego.
- La imagen de la cerilla vuelve a su estado inicial (apagada).

```
papelera.addEventListener("drop", (e) => {
    papelera.src = "papeleraFuego.jpg";
    cerilla.src = "cerilla.jpg"; // Cambiar la imagen de la cerilla a apagada al soltarla
});
```

5.3. Otros eventos:

Evento	Momento en el que se activa
Carga de elementos	
Abort	Cuando se cancela la carga de un elemento
Error	Cuando se produce un error en la carga
Load	Cuando se ha cargado el documento y los elementos externos que incorpora
Progress	Cuando se está produciendo la carga
Ventana	
Scroll	Cuando se desplaza la ventana por medio de las barras de desplazamiento
Resize	Cuando se modifica el tamaño de la ventana
Impresión	
Afterprint	Cuando ha comenzado la impresión o se ha cerrado la previsualización de la impresora
Beforeprint	Cuando va a comenzar la impresión o se ha abierto la previsualización de la impresión
Portapapeles	
Copy	Cuando se va a copiar junto antes de ejecutar la acción.
Cut	Cuando se va a cortar junto antes de ejecutar la acción.
Paste	Cuando se va a pegar junto antes de ejecutar la acción.

Aquí se puede ver una lista con todos los eventos disponibles para gestionar en JS:
https://developer.mozilla.org/es/docs/Web/Events