



1

EVENTOS DE LA UI

2º DAW - Desarrollo Web en Entorno Cliente

Introducción a eventos en el navegador

Un evento es una señal de que algo ocurrió. Todos los nodos del DOM generan dichas señales (pero los eventos no están limitados sólo al DOM).

Aquí hay una lista con los eventos del DOM más utilizados, solo para echar un vistazo:

Eventos del mouse:

- click – cuando el mouse hace click sobre un elemento (los dispositivos touch lo generan con un toque).
- contextmenu – cuando el mouse hace click derecho sobre un elemento.
- mouseover / mouseout – cuando el cursor del mouse ingresa/abandona un elemento.
- mousedown / mouseup – cuando el botón del mouse es presionado/soltado sobre un elemento.
- mousemove – cuando el mouse se mueve.

Eventos del teclado:

- keydown / keyup – cuando se presiona/suelta una tecla.

Eventos del elemento form:

- submit – cuando el visitante envía un <form>.
- focus – cuando el visitante hace foco en un elemento, por ejemplo un <input>.

Eventos del documento:

- DOMContentLoaded --cuando el HTML es cargado y procesado, el DOM está completamente construido

Eventos del CSS:

- transitionend – cuando una animación CSS concluye.

Atributo HTML

Un handler puede ser establecido en el HTML con un atributo llamado `on<event>`. Por ejemplo, para asignar un handler click a un input podemos usar `onclick`, como aquí:

```
1 <input value="Haz click aquí" onclick="alert('¡Click!')" type="button">
```

Al hacer click, el código dentro de `onclick` se ejecuta.

Toma en cuenta que dentro de `onclick` usamos comillas simples, porque el atributo en sí va entre comillas dobles. Si olvidamos que el código está dentro del atributo y usamos comillas dobles dentro, así: `onclick="alert("Click!")"`, no funcionará correctamente.

Un atributo HTML no es un lugar conveniente para escribir un montón de código, así que mejor creamos una función JavaScript y la llamamos allí. Aquí un click ejecuta la función `countRabbits()`:

```
1 <script>
2   function countRabbits() {
3     for(let i=1; i<=3; i++) {
4       alert("Conejo número " + i);
5     }
6   }
7 </script>
8
9 <input type="button" onclick="countRabbits()" value="¡Cuenta los conejos!">
```

¡Cuenta los conejos!

Como sabemos, los nombres de los atributos HTML no distinguen entre mayúsculas y minúsculas, entonces `ONCLICK` funciona bien al igual que `onClick` y `onCLICK`... Pero usualmente los atributos van con minúsculas: **onclick**.

Propiedad del DOM

Podemos asignar un handler usando una propiedad del DOM `on<event>`. Por ejemplo, `elem.onclick`:

```
1 <input id="elem" type="button" value="Haz click en mí">
2 <script>
3   elem.onclick = function() {
4     alert('¡Gracias!');
5   };
6 </script>
```

Haz click en mí

Si el handler es asignado usando un atributo HTML entonces el navegador lo lee, crea una nueva función desde el contenido del atributo y lo escribe en la propiedad del DOM.

Esta forma en realidad es la misma que ya habíamos visto antes. Estás dos piezas de código funcionan igual:

1. Solo HTML:

```
1 <input type="button" onclick="alert('¡Click!')" value="Botón">
```

Botón

2. HTML + JS:

```
1 <input type="button" id="button" value="Botón">
2 <script>
3   button.onclick = function() {
4     alert('¡Click!');
5   };
6 </script>
```

Botón

En el primer ejemplo el atributo HTML es usado para inicializar el `button.onclick`, mientras que en el segundo ejemplo se usa el script. Esa es toda la diferencia.

Como solo hay una propiedad `onclick`, no podemos asignar más de un handler. En el siguiente ejemplo se agrega un handler con JavaScript que sobrescribe el handler existente:

```
1 <input type="button" id="elem" onclick="alert('Antes')" value="¡Haz click en mí!">
2 <script>
3   elem.onclick = function() { // sobrescribe el handler existente
4     alert('Después'); // solo se mostrará este
5   };
6 </script>
```

¡Haz click en mí!

Para eliminar un handler, asigna `elem.onclick = null`.

Posibles errores

Si estás empezando a trabajar con eventos, por favor, nota algunas sutilezas. Nosotros podemos establecer una función existente como un handler:

```
1 function sayThanks() {  
2     alert('¡Gracias!');  
3 }  
4  
5 elem.onclick = sayThanks;
```

Pero ten cuidado: la función debe ser asignada como `sayThanks`, no `sayThanks()`.

```
1 // correcto  
2 button.onclick = sayThanks;  
3  
4 // incorrecto  
5 button.onclick = sayThanks();
```

Si agregamos paréntesis, `sayThanks()` se convierte en una llamada de función. En ese caso la última línea toma el resultado de la ejecución de la función, que es `undefined` (ya que la función no devuelve nada), y lo asigna a `onclick`. Esto no funciona.

...Por otro lado, en el markup necesitamos los paréntesis:

```
1 <input type="button" id="button" onclick="sayThanks()">
```

La diferencia es fácil de explicar. Cuando el navegador lee el atributo crea una función handler con cuerpo a partir del contenido del atributo. Por lo que el markup genera esta propiedad:

```
1 button.onclick = function() {  
2   sayThanks(); // <-- el contenido del atributo va aquí  
3 };
```

No uses `setAttribute` para handlers. Tal llamada no funcionará:

```
1 // un click sobre <body> generará errores,  
2 // debido a que los atributos siempre son strings, la función se convierte en un string  
3 document.body.setAttribute('onclick', function() { alert(1) });
```

NOTA: Las mayúsculas en las propiedades DOM importan.

Asignar un handler a ***elem.onclick***, en lugar de `elem.ONCLICK`, ya que las propiedades DOM son sensibles a mayúsculas.

addEventListener

El problema fundamental de las formas ya mencionadas para asignar handlers es que no podemos asignar múltiples handlers a un solo evento. Digamos que una parte de nuestro código quiere resaltar un botón al hacer click, y otra quiere mostrar un mensaje en el mismo click.

Nos gustaría asignar dos handlers de eventos para eso. Pero una nueva propiedad DOM sobrescribirá la que ya existe:

```
1 input.onclick = function() { alert(1); }  
2 // ...  
3 input.onclick = function() { alert(2); } // el handler reemplaza el handler anterior
```

Los desarrolladores de estándares de la web entendieron eso hace mucho tiempo y sugirieron una forma alternativa de administrar los handlers utilizando los métodos especiales `addEventListener` y `removeEventListener`, que no tienen este problema. La sintaxis para agregar un handler:

```
1 element.addEventListener(event, handler, [options]);
```

- Evento: Nombre del evento, por ejemplo: "click".
- Handler: La función handler.
- Options: Un objeto adicional, opcional, con las propiedades:
 - `once`: si es `true` entonces el listener se remueve automáticamente después de activarlo.
 - `capture`: la fase en la que se controla el evento, que será cubierta en el capítulo Propagación y captura. Por razones históricas, `options` también puede ser `false/true`, lo que es igual a `{capture: false/true}`.
 - `passive`: si es `true` entonces el handler no llamará a `preventDefault()`, esto lo explicaremos más adelante en Acciones predeterminadas del navegador.

Para remover el handler, usa `removeEventListener`:

```
1 element.removeEventListener(event, handler, [options]);
```

Múltiples llamadas a `addEventListener` permiten agregar múltiples handlers:

```
1 <input id="elem" type="button" value="Haz click en mí"/>
2
3 <script>
4   function handler1() {
5     alert('¡Gracias!');
6   };
7
8   function handler2() {
9     alert('¡Gracias de nuevo!');
10  }
11
12  elem.onclick = () => alert("Hola");
13  elem.addEventListener("click", handler1); // Gracias!
14  elem.addEventListener("click", handler2); // Gracias de nuevo!
15 </script>
```

Como podemos ver en el ejemplo anterior, podemos establecer handlers tanto usando una propiedad DOM como `addEventListener` juntos. Pero por lo general solo usamos una de esas maneras.

Objeto del evento

Pero para manejar correctamente un evento necesitamos saber todavía más acerca de lo que está pasando. No solo si fue un “click” o un “teclazo”, sino ¿cuáles eran coordenadas del cursor, o qué tecla fue oprimida? Y así.

Cuando un evento ocurre, el navegador crea un objeto del evento, coloca los detalles dentro y los pasa como un argumento al handler. Aquí hay un ejemplo para obtener las coordenadas del cursor a partir del objeto del evento:

```
1 <input type="button" value="¡Haz click en mí!" id="elem">
2
3 <script>
4   elem.onclick = function(event) {
5     // muestra el tipo de evento, el elemento y las coordenadas del click
6     alert(event.type + " en el " + event.currentTarget);
7     alert("Coordenadas: " + event.clientX + ":" + event.clientY);
8   };
9 </script>
```

Algunas propiedades del objeto event:

- **event.type**: Tipo de evento, en este caso fue "click".
- **event.currentTarget**: Elemento que maneja el evento. Lo que exactamente igual a this, a menos que el handler sea una función de flecha o su this esté vinculado a otra cosa, entonces podemos obtener el elemento desde event.currentTarget.
- **event.clientX / event.clientY**: Coordenadas del cursor relativas a la ventana, para eventos de cursor.

Hay más propiedades. Muchas de ellas dependen del tipo de evento: los eventos del teclado tienen un conjunto de propiedades, y las de cursor, otro. Los estudiaremos después, cuando lleguemos a los detalles de diferentes eventos.

Objetos handlers: *handleEvent*

Podemos asignar no solo una función, sino un objeto como handler del evento usando `addEventListener`. Cuando el evento ocurre, el método `handleEvent` es llamado. Por ejemplo:

```
1 <button id="elem">Haz click en mí</button>
2
3 <script>
4   let obj = {
5     handleEvent(event) {
6       alert(event.type + " en " + event.currentTarget);
7     }
8   };
9
10  elem.addEventListener('click', obj);
11 </script>
```

Como podemos ver, cuando `addEventListener` recibe como handler a un objeto, llama a ***obj.handleEvent(event)*** en caso de un evento.

También podemos usar objetos de una clase personalizada:

```

1  <button id="elem">Haz click en mí</button>
2
3  <script>
4    class Menu {
5      handleEvent(event) {
6        switch(event.type) {
7          case 'mousedown':
8            elem.innerHTML = "Botón del mouse presionado";
9            break;
10         case 'mouseup':
11           elem.innerHTML += "...y soltado.";
12           break;
13         }
14       }
15     }
16
17     let menu = new Menu();
18
19     elem.addEventListener('mousedown', menu);
20     elem.addEventListener('mouseup', menu);
21   </script>

```

Aquí el mismo objeto maneja ambos eventos. Nota que necesitamos configurar explícitamente los eventos a escuchar usando `addEventListener`. El objeto `menu` solo obtiene `mousedown` y `mouseup` aquí, no hay ningún otro tipo de eventos.

El método `handleEvent` no tiene que hacer todo el trabajo por sí solo. En su lugar puede llamar a otros métodos específicos de eventos, como este:

```

1  <button id="elem">Haz click en mí</button>
2
3  <script>
4    class Menu {
5      handleEvent(event) {
6        // mousedown -> onMousedown
7        let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
8        this[method](event);
9      }
10
11      onMousedown() {
12        elem.innerHTML = "Botón del mouse presionado";
13      }
14
15      onMouseup() {
16        elem.innerHTML += "...y soltado.";
17      }
18    }
19
20    let menu = new Menu();
21    elem.addEventListener('mousedown', menu);
22    elem.addEventListener('mouseup', menu);
23  </script>

```

RESUMEN

Hay tres formas de También podemos usar objetos de una clase personalizada:

```

1 <button id="elem">Haz click en mí</button>
2
3 <script>
4   class Menu {
5     handleEvent(event) {
6       switch(event.type) {
7         case 'mousedown':
8           elem.innerHTML = "Botón del mouse presionado";
9           break;
10          case 'mouseup':
11            elem.innerHTML += "...y soltado.";
12            break;
13          }
14        }
15      }
16
17      let menu = new Menu();
18
19      elem.addEventListener('mousedown', menu);
20      elem.addEventListener('mouseup', menu);
21 </script>

```

Aquí el mismo objeto maneja ambos eventos. Nota que necesitamos configurar explícitamente los eventos a escuchar usando `addEventListener`. El objeto `menu` solo obtiene `mousedown` y `mouseup` aquí, no hay ningún otro tipo de eventos.

El método `handleEvent` no tiene que hacer todo el trabajo por sí solo. En su lugar puede llamar a otros métodos específicos de eventos, como este:

```

1 <button id="elem">Haz click en mí</button>
2
3 <script>
4   class Menu {
5     handleEvent(event) {
6       // mousedown -> onMousedown
7       let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
8       this[method](event);
9     }
10
11     onMousedown() {
12       elem.innerHTML = "Botón del mouse presionado";
13     }
14
15     onMouseup() {
16       elem.innerHTML += "...y soltado.";
17     }
18   }
19
20   let menu = new Menu();
21   elem.addEventListener('mousedown', menu);
22   elem.addEventListener('mouseup', menu);
23 </script>

```

Asignar handlers:

1. Atributos HTML: `onclick="..."`.
2. Propiedades del DOM: `elem.onclick = function`.
3. Métodos: `elem.addEventListener(event, handler[, phase])` para agregar ó `removeEventListener` para remover.

Los atributos HTML se usan con moderación, porque JavaScript en medio de una etiqueta HTML luce un poco extraño y ajeno. Además no podemos escribir montones de código ahí. Las propiedades del DOM son buenas para usar, pero no podemos asignar más de un handler a un evento en particular. En la mayoría de casos esta limitación no es apremiante.

La última forma es la más flexible, pero también es la más larga para escribir. Unos pocos eventos solo funcionan con ésta, por ejemplo `transitionend` y `DOMContentLoaded` (que veremos después). Además `addEventListener` soporta objetos como handlers de eventos. En este caso `handleEvent` es llamado en caso del evento.

No importa como asignes el handler, este obtiene un objeto como primer argumento. Este objeto contiene los detalles sobre lo que pasó. Vamos a aprender más sobre eventos en general y sobre diferentes tipos de eventos en los siguientes capítulos.

TAREAS

Ocultar con un click

Agrega JavaScript al button para hacer que `<div id="text">` desaparezca al clickearlo.

El demo:

Haz click para desaparecer el texto
Texto

Abrir un entorno controlado para la tarea.

Ocultarse

Crea un botón que se oculte a sí mismo al darle un click.

Así: Click para esconder

Crear un menú deslizante

Crea un menú que se abra/colapse al hacer click:

▼ Sweeties (click me)!

- Cake
- Donut
- Honey

P.D. El HTML/CSS del documento fuente se debe modificar.

[Abrir un entorno controlado para la tarea.](#)

Propagación y captura

Vamos a empezar con un ejemplo. Este manejador está asignado a `<div>`, pero también se ejecuta si haces clic a cualquier elemento anidado como `` ó `<code>`:

```
1 <div onclick="alert('¡El manejador!')">
2   <em>Si haces clic en<code>EM</code>, el manejador en <code>DIV</code> es ejecutado.
3 </div>
```

Si haces clic enEM, el manejador en DIV es ejecutado.

Propagación

El principio de propagación es simple. Cuando un evento ocurre en un elemento, este primero ejecuta los manejadores que tiene asignados, luego los manejadores de su padre, y así hasta otros ancestros.

Digamos que tenemos 3 elementos anidados `FORM > DIV > P` con un manejador en cada uno de ellos:

```
1 <style>
2   body * {
3     margin: 10px;
4     border: 1px solid blue;
5   }
6 </style>
7
8 <form onclick="alert('form')">FORM
9   <div onclick="alert('div')">DIV
10    <p onclick="alert('p')">P</p>
11  </div>
12 </form>
```

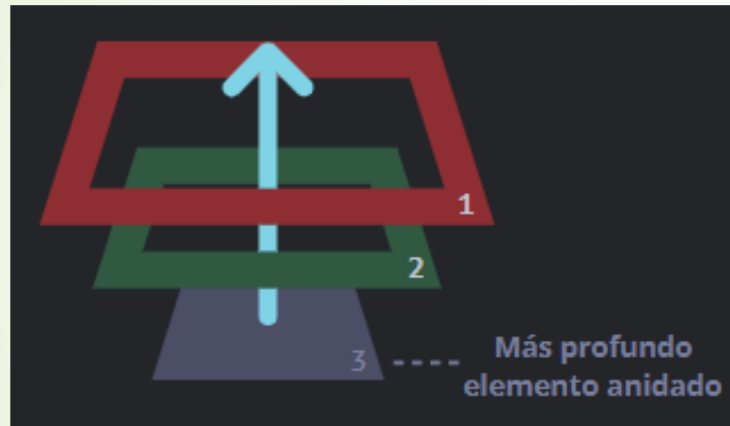
FORM

DIV

P

Un clic en el elemento del interior <p> primero ejecuta onclick:

1. En ese <p>.
2. Luego en el <div> de arriba.
3. Luego en el <form> de más arriba.
4. Y así sucesivamente hasta el objeto document.



Así si hacemos clic en <p>, entonces veremos 3 alertas: ***p*** → ***div*** → ***form***. Este proceso se conoce como “propagación” porque los eventos “se propagan” desde el elemento más al interior, a través de los padres, como una burbuja en el agua.

Event.target

Un manejador en un elemento padre siempre puede obtener los detalles sobre dónde realmente ocurrió el evento. El elemento anidado más profundo que causó el evento es llamado elemento objetivo, accesible como ***event.target***.

Nota la diferencia de this (= ***event.currentTarget***):

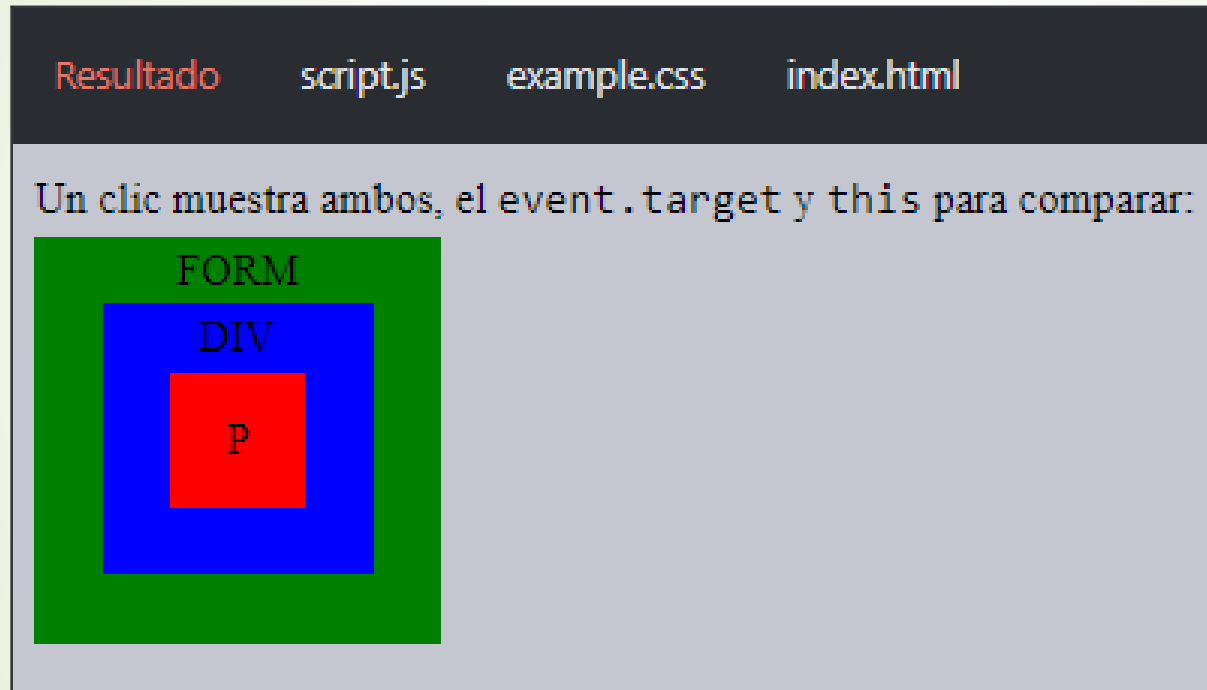
- ***event.target*** – es el elemento “objetivo” que inició el evento, no cambia a través de todo el proceso de propagación.
- this – es el elemento “actual”, el que tiene un manejador ejecutándose en el momento.

Por ejemplo, si tenemos un solo manejador ***form.onclick***, este puede atrapar todos los clicks dentro del formulario. No importa dónde el clic se hizo, se propaga hasta el <form> y ejecuta el manejador. En el manejador ***form.onclick***:

- this (=event.currentTarget) es el elemento <form>, porque el manejador se ejecutó en él.
- event.target es el elemento actual dentro de el formulario al que se le hizo clic.

Ejemplo

Mira esto. Ejemplo → <https://plnkr.co/edit/Ffx43GvPqLxJxTfH?p=preview>



Es posible que `event.target` sea igual a `this`: ocurre cuando el clic se hace directamente en el elemento `<form>`.

Detener la propagación

Una propagación de evento empieza desde el elemento objetivo hacia arriba. Normalmente este continúa hasta `<html>` y luego hacia el objeto `document`, algunos eventos incluso alcanzan `window`, llamando a todos los manejadores en el camino.

Pero cualquier manejador podría decidir que el evento se ha procesado por completo y detener su propagación. El método para esto es **`event.stopPropagation()`**. Por ejemplo, aquí **`body.onclick`** no funciona si haces clic en `<button>`:

```
1 <body onclick="alert(`No se propagó hasta aquí`)">
2   <button onclick="event.stopPropagation()">Haz clic</button>
3 </body>
```

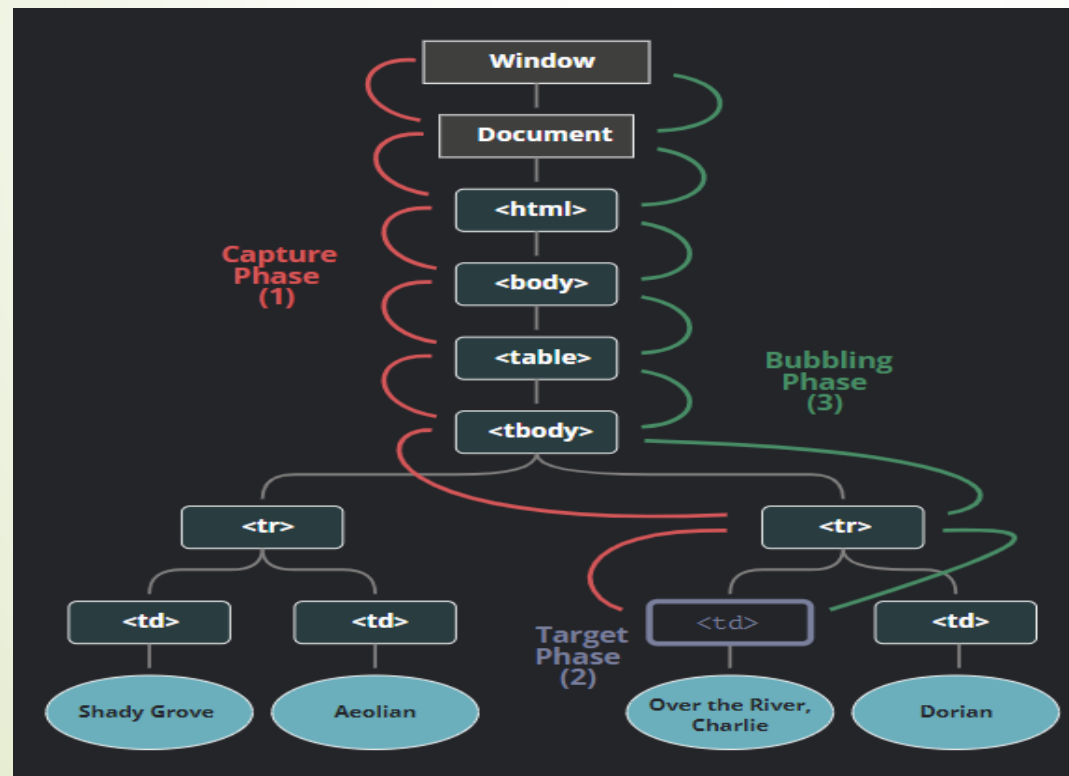
Haz clic

Captura

Hay otra fase en el procesamiento de eventos llamada “captura”. Es raro usarla en código real, pero a veces puede ser útil. El estándar de eventos del DOM describe 3 fases de la propagación de eventos:

- Fase de captura – el evento desciende al elemento.
- Fase de objetivo – el evento alcanza al elemento.
- Fase de propagación – el evento se propaga hacia arriba del elemento.

Aquí (tomada de la especificación), tenemos la imagen de las fases de captura (1), objetivo (2), y propagación (3), de un evento click en un `<td>` dentro de una tabla:



Se explica así: por un clic en <td> el evento va primero a través de la cadena de ancestros hacia el elemento (fase de captura), luego alcanza el objetivo y se desencadena ahí (fase de objetivo), y por último va hacia arriba (fase de propagación), ejecutando los manejadores en su camino.

Hasta ahora solo hablamos de la propagación, porque la fase de captura es raramente usada. De hecho, la fase de captura es invisible para nosotros, porque los manejadores agregados que usan la propiedad on<event>, ó usan atributos HTML, ó **`addEventListener(event, handler)`** de dos argumentos, no ven la fase de captura, únicamente se ejecutan en la 2da y 3ra fase.

Para atrapar un evento en la fase de captura, necesitamos preparar la opción capture como true en el manejador:

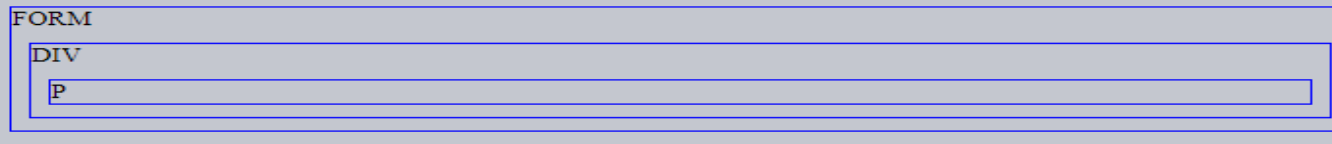
```
1 elem.addEventListener(..., {capture: true})  
2  
3 // o solamente "true". Es una forma más corta de {capture: true}  
4 elem.addEventListener(..., true)
```

Hay dos posibles valores para la opción capture:

- Si es false (por defecto), entonces el manejador es preparado para la fase de propagación.
- Si es true, entonces el manejador es preparado para la fase de captura.

Es de notar que mientras formalmente hay 3 fases, la 2da fase (“la fase de objetivo”: el evento alcanzó el elemento) no es manejada de forma separada; los manejadores en ambas fases, la de captura y propagación, se disparan en esa fase. Veamos ambas fases, captura y propagación, en acción:

```
1 <style>
2   body * {
3     margin: 10px;
4     border: 1px solid blue;
5   }
6 </style>
7
8 <form>FORM
9   <div>DIV
10    <p>P</p>
11  </div>
12 </form>
13
14 <script>
15   for(let elem of document.querySelectorAll('*')) {
16     elem.addEventListener("click", e => alert(`Captura: ${elem.tagName}`), true);
17     elem.addEventListener("click", e => alert(`Propagación: ${elem.tagName}`));
18   }
19 </script>
```



El código prepara manejadores de clic en cada elemento en el documento para ver cuáles están funcionando. Si haces clic en `<p>`, verás que la secuencia es:

1. HTML → BODY → FORM → DIV (fase de captura, el primer detector o “listener”);
2. P → DIV → FORM → BODY → HTML (fase de propagación, el segundo detector).

Nota que P aparece dos veces, porque establecimos dos listeners: captura y propagación. Se disparan en el objetivo al final de la primera fase y al principio de la segunda fase. Hay un propiedad ***event.eventPhase*** que nos dice el número de fase en la que el evento fue capturado. Pero es raramente usada, ya que usualmente lo sabemos en el manejador.

RESUMEN

- Cuando ocurre un evento, el elemento más anidado dónde ocurrió se reconoce como el “elemento objetivo” (`event.target`).
 - Luego el evento se mueve hacia abajo desde el documento raíz hacia `event.target`, llamando a los manejadores en el camino asignados con `addEventListener(..., true)` (`true` es una abreviación para `{capture: true}`).
 - Luego los manejadores son llamados en el elemento objetivo mismo.
 - Luego el evento se propaga desde `event.target` hacia la raíz, llamando a los manejadores que se asignaron usando `on<event>`, atributos HTML y `addEventListener` sin el 3er argumento o con el 3er argumento `false`/`{capture:false}`.
- Cada manejador puede acceder a las propiedades del objeto `event`:
 - `event.target` – el elemento más profundo que originó el evento.
 - `event.currentTarget` (`=this`) – el elemento actual que maneja el evento (el que tiene al manejador en él)
 - `event.eventPhase` – la fase actual (`captura=1`, `objetivo=2`, `propagación=3`).
- Cualquier manejador de evento puede detener el evento al llamar `event.stopPropagation()`, pero no es recomendado porque no podemos realmente asegurar que no lo necesitaremos más adelante, quizá para completar diferentes cosas.
- La fase de captura raramente es usada, usualmente manejamos los eventos en la propagación. Y hay una explicación lógica para ello. En el mundo real, cuando un accidente ocurre, las autoridades locales reaccionan primero. Ellos conocen mejor el área dónde ocurrió. Luego, si es necesario, las autoridades de alto nivel.
- Lo mismo para los manejadores de eventos. El código que se prepara en el manejador de un elemento en particular conoce el máximo de detalles sobre el elemento y qué hace. Un manejador en un `<td>` particular puede ser adecuado para ese exacto `<td>`, conocer todo sobre él, entonces debe tener su oportunidad primero. Luego su padre inmediato también conoce sobre el contexto, pero un poco menos, y así sucesivamente hasta el elemento de arriba que maneja conceptos generales y se ejecuta al final.

Eventos de Mouse

- En este capítulo vamos a entrar en más detalles sobre los eventos del mouse y sus propiedades. Ten en cuenta que tales eventos pueden provenir no sólo del “dispositivo mouse”, sino también de otros dispositivos, como teléfonos y tabletas, donde se emulan por compatibilidad.
- **Tipos de eventos del mouse**
- Ya hemos visto algunos de estos eventos:
 - **mousedown/mouseup**: Se oprime/suelta el botón del ratón sobre un elemento.
 - **mouseover/mouseout**: El puntero del mouse se mueve sobre/sale de un elemento.
 - **mousemove**: Cualquier movimiento del mouse sobre un elemento activa el evento.
 - **Click**: Se activa después de mousedown y un mouseup enseguida sobre el mismo elemento si se usó el botón.
 - **Dblclick**: Se activa después de dos clicks seguidos sobre el mismo elemento. Hoy en día se usa raramente.
 - **Contextmenu**: Se activa al pulsar el botón derecho del ratón. Existen otras formas de abrir el menú contextual, por ejemplo: usando un comando especial de teclado también puede activarse, de manera que no es exactamente un evento exclusivo del mouse.

El orden de los eventos

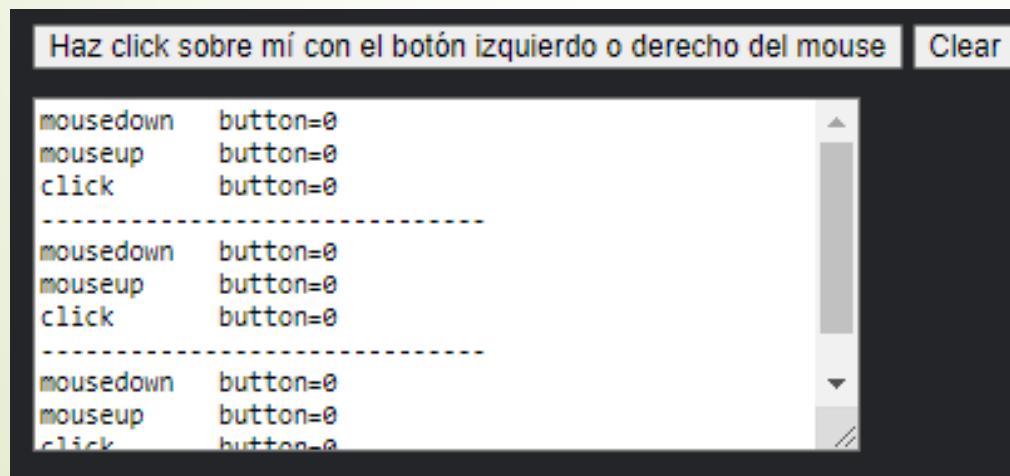
Como pudiste ver en la lista anterior, una acción del usuario puede desencadenar varios eventos. Por ejemplo, un click izquierdo primero activa **mousedown** cuando se presiona el botón, enseguida **mouseup** y **click** cuando se suelta.

En casos así, el orden es fijo. Es decir, los controladores son llamados en el siguiente orden **mousedown** → **mouseup** → **click**.

Haz click en el botón abajo y verás los eventos. Intenta con doble click también.

En el testeador de abajo todos los eventos quedan registrados. Si hay mas de un segundo de retraso entre cada uno de ellos quedan separados por una línea horizontal.

Además podemos ver la propiedad de `button` que nos permite detectar el botón del mouse.



El botón del mouse

Los eventos relacionados con clics siempre tienen la propiedad `button`, esta nos permite conocer el botón exacto del mouse. Normalmente no la usamos para eventos `click` y `contextmenu` events, porque sabemos que ocurren solo con click izquierdo y derecho respectivamente.

Por otro lado, los controladores `mousedown` y `mouseup` pueden necesitar **`event.button`** ya que estos eventos se activan con cualquier botón, y `button` nos permitirá distinguir entre “mousedown derecho” y “mousedown izquierdo”. Los valores posibles para **`event.button`** son:

Estado del botón	<code>event.button</code>
Botón izquierdo (primario)	0
Botón central (auxiliar)	1
Botón derecho (secundario)	2
Botón X1 (atrás)	3
Botón X2 (adelante)	4

La mayoría de los dispositivos de ratón sólo tienen los botones izquierdo y derecho, por lo que los valores posibles son 0 o 2. Los dispositivos táctiles también generan eventos similares cuando se toca sobre ellos.

También hay una propiedad **`event.buttons`** que guarda todos los botones presionados actuales en un solo entero, un bit por botón. En la práctica, esta propiedad es raramente utilizada. Puedes encontrar más detalles en MDN si alguna vez lo necesitas.

Modificadores: shift, alt, ctrl y meta

Todos los eventos del mouse incluyen la información sobre las teclas modificadoras presionadas.

Propiedades del evento:

- shiftKey: Shift
- altKey: Alt (p Opt para Mac)
- ctrlKey: Ctrl
- metaKey: Cmd para Mac

Su valor es true si la tecla fue presionada durante el evento. Por ejemplo, el botón abajo solo funciona con Alt+Shift+click:

```
1 <button id="button">Alt+Shift+¡Click sobre mí!</button>
2
3 <script>
4   button.onclick = function(event) {
5     if (event.altKey && event.shiftKey) {
6       alert('¡Genial!');
7     }
8   };
9 </script>
```

Alt+Shift+¡Click sobre mí!

Coordenadas: clientX/Y, pageX/Y

Todos los eventos del ratón proporcionan coordenadas en dos sabores:

1. Relativas a la ventana: clientX y clientY.
2. Relativos al documento: pageX y pageY.

En resumen, las coordenadas relativas al documento pageX/Y se cuentan desde la esquina superior izquierda del documento y no cambian cuando se desplaza la página, mientras que clientX/Y se cuentan desde la esquina superior actual. Cambian cuando se desplaza la página.

Por ejemplo, si tenemos una ventana del tamaño 500x500, y el mouse está en la esquina superior izquierda, entonces clientX y clientY son 0, sin importar cómo se desplace la página.

Y si el mouse está en el centro, entonces clientX y clientY son 250, No importa en qué parte del documento se encuentren. Esto es similar a position:fixed en ese aspecto.

Mueve el mouse sobre el campo de entrada para ver clientX/clientY (el ejemplo está dentro del iframe, así que las coordenadas son relativas al iframe):

```
<input onmousemove="this.value=event.clientX+':'+event.clientY" value="Pasa el mouse sobre mí">
```



Previniendo la selección en mousedown

El doble clic del mouse tiene un efecto secundario que puede ser molesto en algunas interfaces: selecciona texto. Por ejemplo, un doble clic en el texto de abajo lo selecciona además de activar nuestro controlador:

```
1 <span ondblclick="alert('dblclick')">Haz doble click en mí</span>
```

Haz doble click en mí

Si se pulsa el botón izquierdo del ratón y, sin soltarlo, mueve el ratón, también hace la selección, a menudo no deseado. En este caso particular, la forma más razonable es evitar la acción del navegador mousedown. Esto evita ambas selecciones:

```
1 Antes...  
2 <b ondblclick="alert('Click!')" onmousedown="return false">  
3   Haz doble click en mí  
4 </b>  
5 ...Después
```

Antes... Haz doble click en mí ...Después

Ahora el elemento en negrita no se selecciona con doble clic, y al mantener presionado el botón izquierdo y arrastrar no se iniciará la selección.

Tenga en cuenta: el texto dentro de él todavía es seleccionable. Sin embargo, la selección no debe comenzar en el texto en sí, sino antes o después. Por lo general, eso está bien para los usuarios.

RESUMEN

- ▶ Los eventos del mouse tienen las siguientes propiedades:
 - ▶ Botón: `button`.
 - ▶ Teclas modificadoras (true si fueron presionadas): `altKey`, `ctrlKey`, `shiftKey` y `metaKey` (Mac).
 - ▶ Si quieres controlar las acciones de la tecla Ctrl no te olvides de los usuarios de Mac que generalmente usan Cmd, de manera que es mejor verificar con la condicional: `if (e.metaKey || e.ctrlKey)`.
 - ▶ Coordenadas relativas a la ventana: `clientX/clientY`.
 - ▶ Coordenadas relativas al documento: `pageX/pageY`.
- ▶ La acción predeterminada del navegador `mousedown` es la selección del texto, si no es bueno para la interfaz, entonces debe evitarse.

TAREAS

Lista seleccionable

Cree una lista donde los elementos son seleccionables, como en los administradores de archivos.

Un clic en un elemento de la lista selecciona solo ese elemento (agrega la clase `.selected`), deselecta todos los demás.

Si se hace un clic con `Ctrl` (`Cmd` para Mac), el estado seleccionado/deseleccionado cambia para ese solo elemento, los otros elementos no se modifican.

Demo:

Haz click en un elemento de la lista para seleccionarlo

- Christopher Robin
- Winnie-the-Pooh
- Tigger
- Kanga
- Rabbit. Just rabbit.

PD: Para esta tarea, podemos suponer que los elementos de la lista son solo de texto. No hay etiquetas anidadas.

PPD: Evita la selección nativa del navegador del texto en los clics.

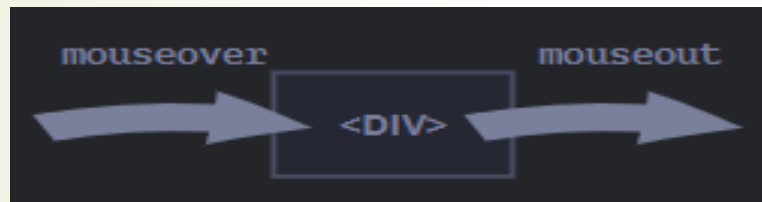
[Abrir un entorno controlado para la tarea.](#)

Moviendo el mouse: mouseover/out, mouseenter/leave

Entremos en detalle sobre los eventos que suceden cuando el mouse se mueve entre elementos.

Eventos mouseover/mouseout, relatedTarget

El evento mouseover se produce cuando el cursor del mouse aparece sobre un elemento y mouseout cuando se va.



Estos eventos son especiales porque tienen la propiedad `relatedTarget`. Esta propiedad complementa a `target`. Cuando el puntero del mouse deja un elemento por otro, uno de ellos se convierte en `target` y el otro en `relatedTarget`. Para `mouseover`:

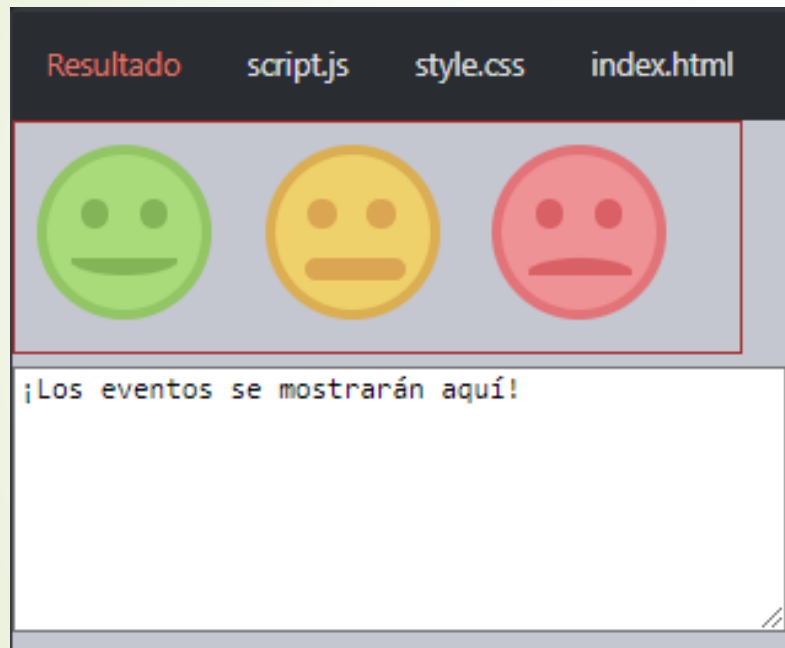
- `event.target` – Es el elemento al que se acerca el mouse.
- `event.relatedTarget` – Es el elemento de donde proviene el mouse (`relatedTarget` → `target`).

Para `mouseout` sucede al contrario:

- `event.target` – Es el elemento que el mouse dejó.
- `event.relatedTarget` – es el nuevo elemento bajo el cursor por cuál el cursor dejó al anterior (`target` → `relatedTarget`).

En el siguiente ejemplo, cada cara y sus características son elementos separados. Puedes ver en el área de texto los eventos que ocurren cuando mueves el mouse. Cada evento tiene la información sobre ambas propiedades: target y relatedTarget.

Ejemplo → <https://plnkr.co/edit/Vw00IYL6VVM5T65p?p=preview&preview>



Saltando elementos

El evento `mousemove` se activa cuando el mouse se mueve, pero eso no significa que cada píxel nos lleve a un evento. El navegador verifica la posición del mouse de vez en cuando y si nota cambios entonces activan los eventos.

Eso significa que si el visitante mueve el mouse muy rápido, entonces algunos elementos del DOM podrían estar siendo ignorados:



Si el mouse se mueve muy rápido de los elementos `#FROM` a `#TO`, como se muestra arriba, entonces los elementos intermedios `<div>` (o algunos de ellos) podrían ser ignorados. El evento `mouseout` se podría activar en `#FROM` e inmediatamente `mouseover` en `#TO`.

Eso es bueno para el rendimiento porque puede haber muchos elementos intermedios. Realmente no queremos procesar todo lo que sucede dentro y fuera de cada uno. Por otro lado, debemos tener en cuenta que el puntero del mouse no “visita” todos los elementos en el camino. Los puede “saltar”.

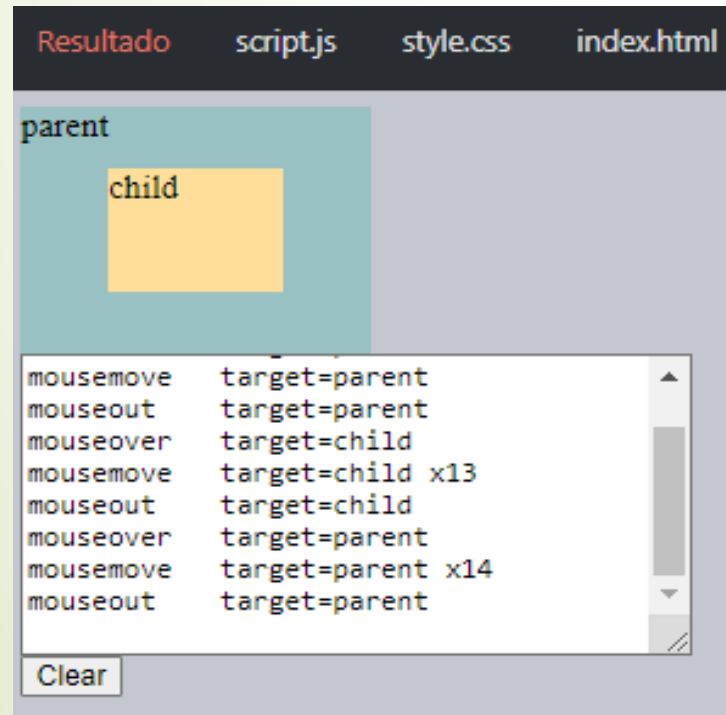
En particular, es posible que el puntero salte dentro de la mitad de la página desde la ventana. En ese caso `relatedTarget` es `null`, porque vino de “la nada”:



Puedes verlo “en vivo” en un testeador a continuación. Este HTML tiene dos elementos: el `<div id="child">` está adentro del `<div id="parent">`. Si mueves el mouse rápidamente sobre ellos entonces probablemente solo el div hijo active los eventos, o probablemente el padre, o probablemente no ocurran eventos en lo absoluto.

También prueba a mover el cursor hacia el div hijo y luego muévelo rápidamente hacia abajo a través del padre. Si el movimiento es lo suficientemente rápido entonces el padre será ignorado. El mouse cruzará el elemento padre sin notarlo.

Ejemplo → <https://plnkr.co/edit/bM3i0aWJJXVwaEk0?p=preview&preview>

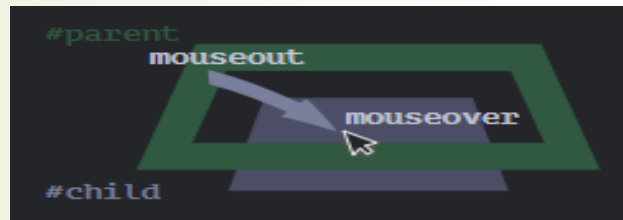


Mouseout, cuando se deja un elemento por uno anidado

Una característica importante de mouseout – se activa cuando el cursor se mueve de un elemento hacia su descendiente (elemento anidado o interno). Por ejemplo de #parent a #child en este HTML:

```
1 <div id="parent">
2   <div id="child">...</div>
3 </div>
```

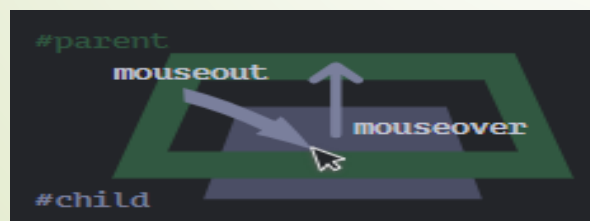
Si estamos sobre #parent y luego movemos el cursor hacia dentro de #child, ¡vamos a obtener mouseout en #parent!



Eso puede parecer extraño, pero puede explicarse fácilmente. De acuerdo con la lógica del navegador, el cursor podría estar sobre un elemento individual en cualquier momento – el anidado y el más alto según el z-index.

Entonces si se dirige hacia otro elemento (incluso uno anidado), está dejando al anterior. Por favor, note otro importante detalle sobre el procesamiento de eventos.

El evento mouseover se aparece en un un elemento anidado (brota o nace, por decirlo así). Entonces si #parent tiene el controlador mouseover, se activa:



Puedes verlo muy bien a continuación: `<div id="child">` está dentro de `<div id="parent">`. Hay controladores `mouseover/out` en el elemento `#parent` que arrojan los detalles de los eventos. Si mueves el mouse de `#parent` a `#child`, verás dos eventos sobre `#parent`:

- `mouseout [target: parent]` (dejó al padre), luego
- `mouseover [target: child]` (vino hacia el hijo, y este evento brotó).

Ejemplo → <https://plnkr.co/edit/7DUexgB77IoGlcU?p=preview&preview>

Como se muestra, cuando el cursor se mueve del elemento `#parent` a `#child`, los dos controladores se activan en el elemento padre: `mouseout` y `mouseover`:

```
1 parent.onmouseout = function(event) {  
2   /* event.target: elemento padre */  
3 };  
4 parent.onmouseover = function(event) {  
5   /* event.target: elemento hijo (brota) */  
6 };
```

Si no examinamos `event.target` dentro de los controladores podría parecer que el cursor dejó el elemento `#parent` y volvió a él inmediatamente. Pero ese no es el caso. El cursor aún está sobre el elemento padre, simplemente se adentró más en el elemento hijo.

Si hay algunas acciones al abandonar el elemento padre, por ejemplo: una animación se ejecuta con `parent.onmouseout`, usualmente no la queremos cuando el cursor se adentre más sobre `#parent`. Para evitar esto lo que podemos hacer es checar `relatedTarget` en el controlador y si el mouse aún permanece dentro del elemento entonces ignorar dicho evento.

Alternativamente podemos usar otros eventos: `mouseenter` y `mouseleave`, los cuales cubriremos a continuación, ya que con ellos no hay tales problemas.

Eventos mouseenter y mouseleave

Los eventos mouseenter/mouseleave son como mouseover/mouseout. Se activan cuando el cursor del mouse entra/sale del elemento. Pero hay dos diferencias importantes:

- Las transiciones hacia/desde los descendientes no se cuentan.
- Los eventos mouseenter/mouseleave no brotan.

Son eventos extremadamente simples. Cuando el cursor entra en un elemento mouseenter se activa. La ubicación exacta del cursor dentro del elemento o sus descendientes no importa.

Cuando el cursor deja el elemento mouseleave se activa. Este ejemplo es similar al anterior, pero ahora el elemento tiene mouseenter/mouseleave en lugar de mouseover/mouseout.

Como puedes ver, los únicos eventos generados son los relacionados con mover el puntero dentro y fuera del elemento superior. No pasa nada cuando el puntero va hacia el descendiente y regresa. Las transiciones entre descendientes se ignoran.

Ejemplo → <https://plnkr.co/edit/bceOAg7VVFKvCY5J?p=preview&preview>



Delegación de eventos

Los eventos `mouseenter/leave` son muy simples de usar. Pero no brotan por sí solos. Por lo tanto no podemos usar la delegación de eventos con ellos. Imagina que queremos manejar entrada/salida para celdas de tabla y hay cientos de celdas.

La solución natural sería: ajustar el controlador en `<table>` y manejar los eventos desde ahí. Pero `mouseenter/leave` no aparece. Entonces si cada evento sucede en `<td>`, solamente un controlador `<td>` es capaz de detectarlo.

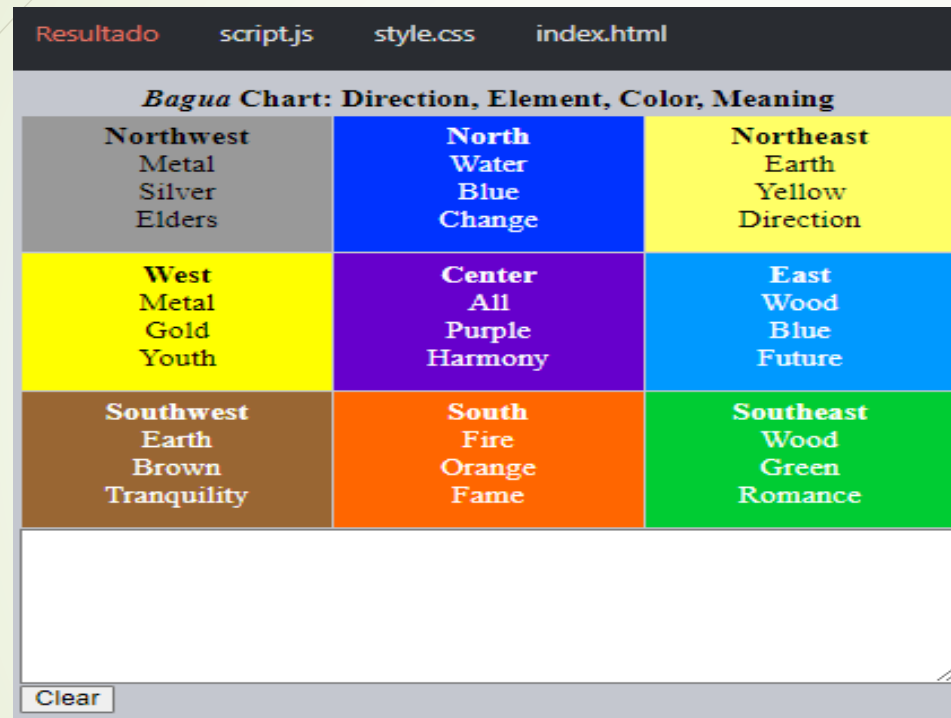
Los controladores `mouseenter/leave` en `<table>` solamente se activan cuando el cursor entra/deja la tabla completa. Es imposible obtener alguna información sobre las transiciones dentro de ella. Pues usemos `mouseover/mouseout`.

Comencemos con controladores simples que resaltan el elemento debajo del mouse:

```
1 // Resaltemos un elemento debajo del cursor
2 table.onmouseover = function(event) {
3   let target = event.target;
4   target.style.background = 'pink';
5 };
6
7 table.onmouseout = function(event) {
8   let target = event.target;
9   target.style.background = '';
10 };
```

Aquí se muestran en acción. A medida que el mouse recorre los elementos de esta tabla, se resalta la actual:

Ejemplo → <https://plnkr.co/edit/Ngnstzd3u6OS3cyM?p=preview&preview>



The screenshot shows a web browser window with a tab labeled 'Resultado'. The page content is a Bagua Chart table titled 'Bagua Chart: Direction, Element, Color, Meaning'. The table has 3 rows and 3 columns. The first row contains Northwest, North, and Northeast. The second row contains West, Center, and East. The third row contains Southwest, South, and Southeast. The 'North' cell is highlighted in blue. Below the table is a large empty text area and a 'Clear' button.

Bagua Chart: Direction, Element, Color, Meaning		
Northwest Metal Silver Elders	North Water Blue Change	Northeast Earth Yellow Direction
West Metal Gold Youth	Center All Purple Harmony	East Wood Blue Future
Southwest Earth Brown Tranquility	South Fire Orange Fame	Southeast Wood Green Romance

Clear

En nuestro caso nos gustaría manejar las transiciones entre las celdas de la tabla `<td>`: entradas y salidas de una celda a otra. Otras transiciones, como dentro de una celda o fuera de cualquiera de ellas no nos interesan. Vamos a filtrarlas. Esto es lo que podemos hacer:

- Recordar el elemento `<td>` resaltado actualmente en una variable, llamémosla `currentElem`.
- En `mouseover` ignoraremos el evento si permanecemos dentro del `<td>` actual.
- En `mouseout` ignoraremos el evento si no hemos dejado el `<td>` actual.

Aquí hay un ejemplo de código que explica todas las situaciones posibles:

```
1 // Los elementos <td> bajo el mouse justo ahora(si es que hay)
2 let currentElem = null;
3
4 table.onmouseover = function(event) {
5     // antes de ingresar un nuevo elemento, el mouse siempre abandonará al anterior
6     // si currentElem está establecido, no abandonamos el <td> anterior,
7     // hay un mouseover dentro de él, ignoramos el evento
8     if (currentElem) return;
9
10    let target = event.target.closest('td');
11
12    // si no hay movimientos dentro de un <td> - lo ignoramos
13    if (!target) return;
14
15    //si hay movimientos dentro de un <td>, pero afuera de una tabla(posiblemente en
16    // lo ignoramos
17    if (!table.contains(target)) return;
18
19    // ¡Genial! ingresamos a un nuevo <td>
20    currentElem = target;
21    onEnter(currentElem);
22 };
```

```
24
25 table.onmouseout = function(event) {
26     // si estamos afuera de algún <td> ahora, entonces ignoramos el evento
27     // puede haber movimientos dentro de una tabla, pero fuera de <td>,
28     // por ejemplo: de un <tr> a otro <tr>
29     if (!currentElem) return;
30
31     // abandonamos el elemento - ¿pero hacia dónde? ¿podría ser hacia un descendiente?
32     let relatedTarget = event.relatedTarget;
33
34     while (relatedTarget) {
35         // vamos a la cadena de padres y verificamos - si aún estamos dentro de currentElem
36         // entonces hay una transición interna - la ignoramos
37         if (relatedTarget == currentElem) return;
38
39         relatedTarget = relatedTarget.parentNode;
40     }
41
42     // abandonamos el <td>.
43     onLeave(currentElem);
44     currentElem = null;
45 };
```



```
46
47 // algunas funciones para manejar entradas/salidas de un elemento
48 function onEnter(elem) {
49     elem.style.background = 'pink';
50
51     // lo mostramos en el área de texto
52     text.value += `over -> ${currentElem.tagName}.${currentElem.className}\n`;
53     text.scrollTop = 1e6;
54 }
55
56 function onLeave(elem) {
57     elem.style.background = '';
58
59     // lo mostramos en el area de texto
60     text.value += `out <- ${elem.tagName}.${elem.className}\n`;
61     text.scrollTop = 1e6;
62 }
```


Una vez más, las características importantes son:

1. Utilizar la delegación de eventos para manejar la entrada/salida de cualquier <td> dentro de la tabla. Pues depende de mouseover/out en lugar de mouseenter/leave que no broten y por lo tanto no permita ninguna delegación.
2. Los eventos adicionales, como moverse entre descendientes de <td> son filtrados, así que onEnter/Leave solamente se ejecuta si el cursor ingresa a <td> o lo deja absolutamente.

Aquí está el ejemplo completo con todos los detalles:

Ejemplo → <https://plnkr.co/edit/Os5aXEeDhQBM6njI?p=preview&preview>

Resultado	script.js	style.css	index.html
Bagua Chart: Direction, Element, Color, Meaning			
Northwest Metal Silver Elders	North Water Blue Change	Northeast Earth Yellow Direction	
West Metal Gold Youth	Center All Purple Harmony	East Wood Blue Future	
Southwest Earth Brown Tranquility	South Fire Orange Fame	Southeast Wood Green Romance	
Clear			

RESUMEN

- Hemos cubierto ***mouseover***, ***mouseout***, ***mousemove***, ***mouseenter*** y ***mouseleave***. Estas cosas son buenas de destacar:
 - Un movimiento rápido del mouse puede omitir elementos intermedios.
 - Los eventos ***mouseover/out*** y ***mouseenter/leave*** tienen una propiedad adicional: `relatedTarget`. Es el elemento de donde venimos o hacia donde vamos, complementario con `target`.
- Los eventos ***mouseover/out*** se activan incluso cuando vamos de un elemento padre a su descendiente. El navegador asume que de el mouse solo puede estar sobre un elemento a la vez – el más interno.
- Los eventos ***mouseenter/leave*** son diferentes en ese aspecto: solo se activan cuando el mouse viene hacia el elemento o lo deja como un todo. Así que no se aparecen de repente.

TAREAS

Comportamiento mejorado de un tooltip

Escribe JavaScript que muestre un tooltip sobre un elemento con el atributo `data-tooltip`. El valor de este atributo debe convertirse en el texto del tooltip.

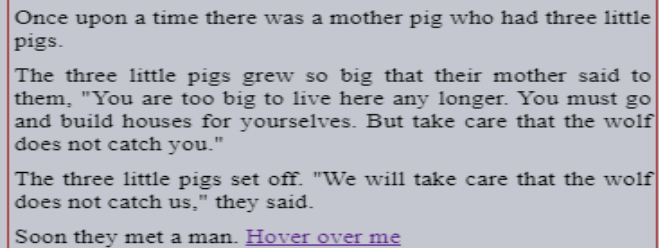
Es como la tarea Comportamiento: Tooltip, pero aquí los elementos anotados se pueden anidar. Los tooltips más internos se muestran.

Solamente un tooltip puede aparecer a la vez.

Por ejemplo:

```
<div data-tooltip="Aquí – está el interior de la casa" id="house">  
  <div data-tooltip="Aquí – está el techo" id="roof"></div>  
  <a href="https://en.wikipedia.org/wiki/The_Three_Little_Pigs" data-  
    tooltip="Continúa leyendo...">Colócate sobre mí</a>  
</div>
```

El resultado en el iframe:



Once upon a time there was a mother pig who had three little pigs.

The three little pigs grew so big that their mother said to them, "You are too big to live here any longer. You must go and build houses for yourselves. But take care that the wolf does not catch you."

The three little pigs set off. "We will take care that the wolf does not catch us," they said.

Soon they met a man. [Hover over me](#)

[Hover over me](#)

[Abrir un entorno controlado para la tarea.](#)

Arrastrar y Soltar con eventos del ratón

- Arrastrar y Soltar es una excelente solución de interfaz. Tomar algo, arrastrar y soltarlo es una forma clara y simple de hacer muchas cosas, desde copiar y mover documentos (como en los manejadores de archivos) hasta ordenar (arrastrando ítems al carrito). En el estándar moderno de HTML hay una sección sobre Arrastrar y Soltar con eventos especiales tales como dragstart, dragend, y así por el estilo.
- Estos eventos nos permiten soportar tipos especiales de Arrastrar y Soltar, como manejar el arrastrado de archivos desde el manejador de archivos del Sistema Operativo y soltarlo en la ventana del navegador. Así JavaScript puede acceder al contenido de dichos archivos.
- Pero los eventos nativos de arrastrar tienen limitaciones. Por ejemplo, no nos deja evitar el arrastre desde cierta área. Tampoco podemos hacer que el arrastre sea solamente “horizontal” o “vertical”. Y hay muchas otras tareas de “Arrastrar y Soltar” que no pueden hacerse utilizándolos. Además, el soporte para dichos eventos es muy pobre en dispositivos móviles. Así que aquí veremos cómo implementar “Arrastrar y Soltar” usando eventos del ratón.

Algoritmo de “Arrastrar y Soltar”

El algoritmo básico de Arrastrar y Soltar se ve así:

- En mousedown – preparar el elemento para moverlo, si es necesario (quizá creando un clon de este, añadiéndole una clase, o lo que sea).
- En mousemove – moverlo cambiando left/top con **position:absolute**.
- En mouseup – realizar todas las acciones relacionadas con finalizar el Arrastrar y Soltar.

Esto es lo básico. Luego veremos como añadir características, como resaltar los elementos subyacentes mientras arrastramos sobre ellos. Aquí esta la implementación de arrastrar una pelota:

```
1 ball.onmousedown = function(event) {
2   // (1) preparar para mover: hacerlo absoluto y ponerlo sobre todo con el z-index
3   ball.style.position = 'absolute';
4   ball.style.zIndex = 1000;
5
6   // quitar cualquier padre actual y moverlo directamente a body
7   // para posicionarlo relativo al body
8   document.body.append(ball);
9
10  // centrar la pelota en las coordenadas (pageX, pageY)
11  function moveAt(pageX, pageY) {
12    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
13    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
14  }
15
16  // mover nuestra pelota posicionada absolutamente bajo el puntero
17  moveAt(event.pageX, event.pageY);
18
19  function onMouseMove(event) {
20    moveAt(event.pageX, event.pageY);
21  }
22
23  // (2) mover la pelota con mousemove
24  document.addEventListener('mousemove', onMouseMove);
25
26  // (3) soltar la pelota, quitar cualquier manejador de eventos innecesario
27  ball.onmouseup = function() {
28    document.removeEventListener('mousemove', onMouseMove);
29    ball.onmouseup = null;
30  };
31
32 };
```


Si trasladamos el código para su ejecución, nos enteramos de algo extraño. Al inicio de arrastrar y soltar, la pelota se duplica: empezamos a arrastrar su “clon”. Coger el código anterior y ejecutarlo para ver su funcionamiento, trata de arrastrar con el ratón y verás dicho comportamiento.

Esto es porque el navegador tiene su propio soporte para arrastrar y soltar para imágenes y otros elementos. Se ejecuta automáticamente y entra en conflicto con el nuestro. Para deshabilitarlo:

```
1 ball.ondragstart = function() {  
2     return false;  
3 };
```

Otro aspecto importante: seguimos mousemove en document, no en ball. Desde el primer momento debe verse que el ratón está siempre sobre la pelota, y podemos poner mousemove en ella.

Pero como recordamos, mousemove se dispara a menudo, pero no por cada pixel. Así que después de un movimiento rápido el puntero puede saltar de la pelota a algún lugar en el medio del documento (o incluso fuera de la ventana). Así que tenemos que escuchar en document para captarlo.

Posicionamiento correcto

En los ejemplos de arriba la pelota siempre se mueve, de manera que su centro queda debajo del puntero:

```
1 ball.style.left = pageX - ball.offsetWidth / 2 + 'px';  
2 ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Nada mal, pero hay un efecto secundario. Para iniciar el arrastrar y soltar, podemos hacer mousedown en cualquier lugar de la pelota. Pero si la “tomamos” por el borde, entonces la pelota “salta” de repente para centrarse bajo el puntero del ratón.

Sería mejor si mantenemos la posición inicial del elemento, relativo al puntero. Por ejemplo, si empezamos a arrastrar por el borde de la pelota, entonces el puntero debería quedarse sobre el borde mientras se arrastra.



Vamos a actualizar nuestro algoritmo:

1. Cuando un visitante presiona el botón (mousedown) – recordar la distancia del puntero a la esquina superior izquierda de la pelota in variables shiftX/shiftY. Mantendremos esa distancia mientras arrastramos. Para obtener esas posiciones podemos restar las coordenadas:

```
1 // onmousedown
2 let shiftX = event.clientX - ball.getBoundingClientRect().left;
3 let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

2. Entonces mientras arrastra posicionamos la pelota en la misma posición relativa al puntero, de esta forma:

```
1 // onmousemove
2 // la pelota tiene position:absolute
3 ball.style.left = event.pageX - shiftX + 'px';
4 ball.style.top = event.pageY - shiftY + 'px';
```

El código final con mejor posicionamiento:

```
1  ball.onmousedown = function(event) {
2
3    let shiftX = event.clientX - ball.getBoundingClientRect().left;
4    let shiftY = event.clientY - ball.getBoundingClientRect().top;
5
6    ball.style.position = 'absolute';
7    ball.style.zIndex = 1000;
8    document.body.append(ball);
9
10   moveAt(event.pageX, event.pageY);
11
12   // mueve la pelota a las coordenadas (pageX, pageY)
13   // tomando la posición inicial en cuenta
14   function moveAt(pageX, pageY) {
15     ball.style.left = pageIndex - shiftX + 'px';
16     ball.style.top = pageY - shiftY + 'px';
17   }
18
19   function onMouseMove(event) {
20     moveAt(event.pageX, event.pageY);
21   }
22
23   // mueve la pelota con mousemove
24   document.addEventListener('mousemove', onMouseMove);
25
26   // suelta la pelota, elimina el manejador innecesario
27   ball.onmouseup = function() {
28     document.removeEventListener('mousemove', onMouseMove);
29     ball.onmouseup = null;
30   };
31
32 };
33
34 ball.ondragstart = function() {
35   return false;
36 };
```

La diferencia es notable especialmente si arrastramos la pelota por su esquina inferior derecha. En el ejemplo anterior la pelota “salta” bajo el puntero. Ahora sigue el puntero fluidamente desde su posición actual.

Objetivos receptores potenciales (droppables)

En los ejemplos anteriores la pelota debe ser soltada simplemente “en cualquier lugar” para quedarse. En la vida real normalmente tomamos un elemento para soltarlo en otro. Por ejemplo, un “archivo” en una “carpeta” o algo más.

Hablando abstracto, tomamos un elemento “arrastrable” y lo soltamos sobre un elemento “receptor”. Necesitamos saber:

- dónde el elemento fue soltado al final del Arrastrar y Soltar – para hacer la acción correspondiente,
- y, preferiblemente, saber el receptor sobre el que estamos arrastrando, para resaltarlo.

La solución es algo interesante y un poco complicado, así que vamos a cubrirlo aquí. ¿Cuál puede ser la primera idea? ¿Probablemente configurar `mouseover/mouseup` en receptores potenciales? Pero eso no funciona.

El problema es que, mientras estamos arrastrando, el elemento arrastrable siempre está encima de los demás elementos. Y los eventos del ratón solo suceden en el elemento superior, no en los que están debajo. Por ejemplo, debajo hay dos elementos `<div>`, el rojo sobre el azul (totalmente cubierto). No hay forma de captar un evento en el azul, porque el rojo está encima:

```
1  <style>
2    div {
3      width: 50px;
4      height: 50px;
5      position: absolute;
6      top: 0;
7    }
8  </style>
9  <div style="background:blue" onmouseover="alert('nunca funciona')"></div>
10 <div style="background:red" onmouseover="alert('sobre el rojo!')"></div>
```



Lo mismo con un elemento arrastrable. La pelota está siempre sobre los demás elementos, así que los eventos pasan en él. Cualquier manejador que pongamos en los elementos de abajo, no funcionará. Por eso la idea inicial de poner manejadores en receptores potenciales no funciona en práctica. No se activarán.

Entonces, ¿Qué hacer? Existe un método llamado `document.elementFromPoint(clientX, clientY)`. Este devuelve el elemento más anidado en las coordenadas relativas a la ventana proporcionada (o null si las coordenadas están fuera de la ventana). Si hay muchos elementos superpuestos en las mismas coordenadas, se devuelve el que está en el tope.

Podemos utilizarlo en cualquiera de nuestros manejadores para detectar los receptores potenciales bajo el puntero, de esta forma:

```
1 // en un manejador de evento del ratón
2 ball.hidden = true; // (*) ocultar el elemento que arrastramos
3
4 let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
5 // elemBelow es el elemento debajo de la pelota, puede ser receptor
6
7 ball.hidden = false;
```

Favor notar: necesitamos ocultar la pelota antes de llamar (*). De otra forma usualmente tendremos una pelota con esas coordenadas, ya que es el elemento superior bajo el puntero: `elemBelow=ball`. Así que lo ocultamos e inmediatamente lo mostramos de nuevo.

Podemos usar este código para verificar el elemento sobre el que estamos “flotando” en todo momento. Y manejar la caída cuando sucede. Un código extendido de onMouseMove para hallar elementos “receptores”:

```
1 // elemento potencialmente arrastrable sobre el que flotamos ahora mismo
2 let currentDraggable = null;
3
4 function onMouseMove(event) {
5     moveAt(event.pageX, event.pageY);
6
7     ball.hidden = true;
8     let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
9     ball.hidden = false;
10
11     // los eventos mousemove se pueden activar fuera de la ventana (cuando la pelota se
12     // si clientX/clientY están fuera de la ventana, entonces elementFromPoint devuelve
13     if (!elemBelow) return;
14
15     // receptores potenciales se etiquetan con la clase "draggable" (puede tener otra ló
16     let draggableBelow = elemBelow.closest('.draggable');
17
18     if (currentDraggable !== draggableBelow) {
19         // estamos flotando dentro o afuera
20         // nota: ambos valores pueden ser null
21         //   currentDraggable=null si no estábamos sobre un receptor antes de este evento
22         //   draggableBelow=null si no estamos sobre un receptor ahora, durante este event
23
24         if (currentDraggable) {
25             // la lógica para procesar "flying out" del receptor (elimina el resaltado)
26             leaveDraggable(currentDraggable);
27         }
28         currentDraggable = draggableBelow;
29         if (currentDraggable) {
30             // la lógica para procesar "flying in" del receptor
31             enterDraggable(currentDraggable);
32         }
33     }
34 }
```


En el siguiente ejemplo cuando la pelota se arrastra sobre la portería, esta se resalta.

Ejemplo → <https://plnkr.co/edit/6GjSD6KXPkxZgleD?p=preview&preview>



Ahora tenemos el “destino” actual, sobre el que estamos flotando, en la variable `currentDroppable` durante el proceso completo y podemos usarlo para resaltar o cualquier otra cosa.

RESUMEN

- Consideramos un algoritmo básico de Arrastrar y Soltar. Los componentes clave:
 1. Flujo de eventos: `ball.mousedown` → `document.mousemove` → `ball.mouseup` (no olvides cancelar el `ondragstart` nativo).
 2. El inicio del arrastrado – recuerda la posición inicial del puntero relativo al elemento: `shiftX/shiftY` y lo mantiene durante el arrastrado.
 3. Detectar elementos arrastrables bajo el puntero usando `document.elementFromPoint`.
- Podemos poner mucho sobre esta base.
 - Con `mouseup` podemos intelectualmente finalizar el arrastre: cambiar datos, mover elementos alrededor.
 - Podemos resaltar los elementos sobre los que estamos volando.
 - Podemos limitar el arrastrado a cierta área o dirección.
 - Podemos usar delegación de eventos para `mousedown/up`. Un manejador de eventos para un área grande que compruebe `event.target` puede manejar Arrastrar y Soltar para cientos de elementos.
 - Y así por el estilo.

TAREAS

Control deslizante

Crea un control deslizante:

Arrastra el pasador azul con el ratón y muévelo.

Detalles importantes:

Cuando el botón del ratón es presionado, durante el arrastrado del ratón puedes ir por arriba o debajo de la barra deslizante. Ésta seguirá funcionando (es lo conveniente para el usuario).

Si el ratón se mueve muy rápido hacia la izquierda o la derecha, el pasador se detiene exactamente en el borde.

[Abrir un entorno controlado para la tarea.](#)

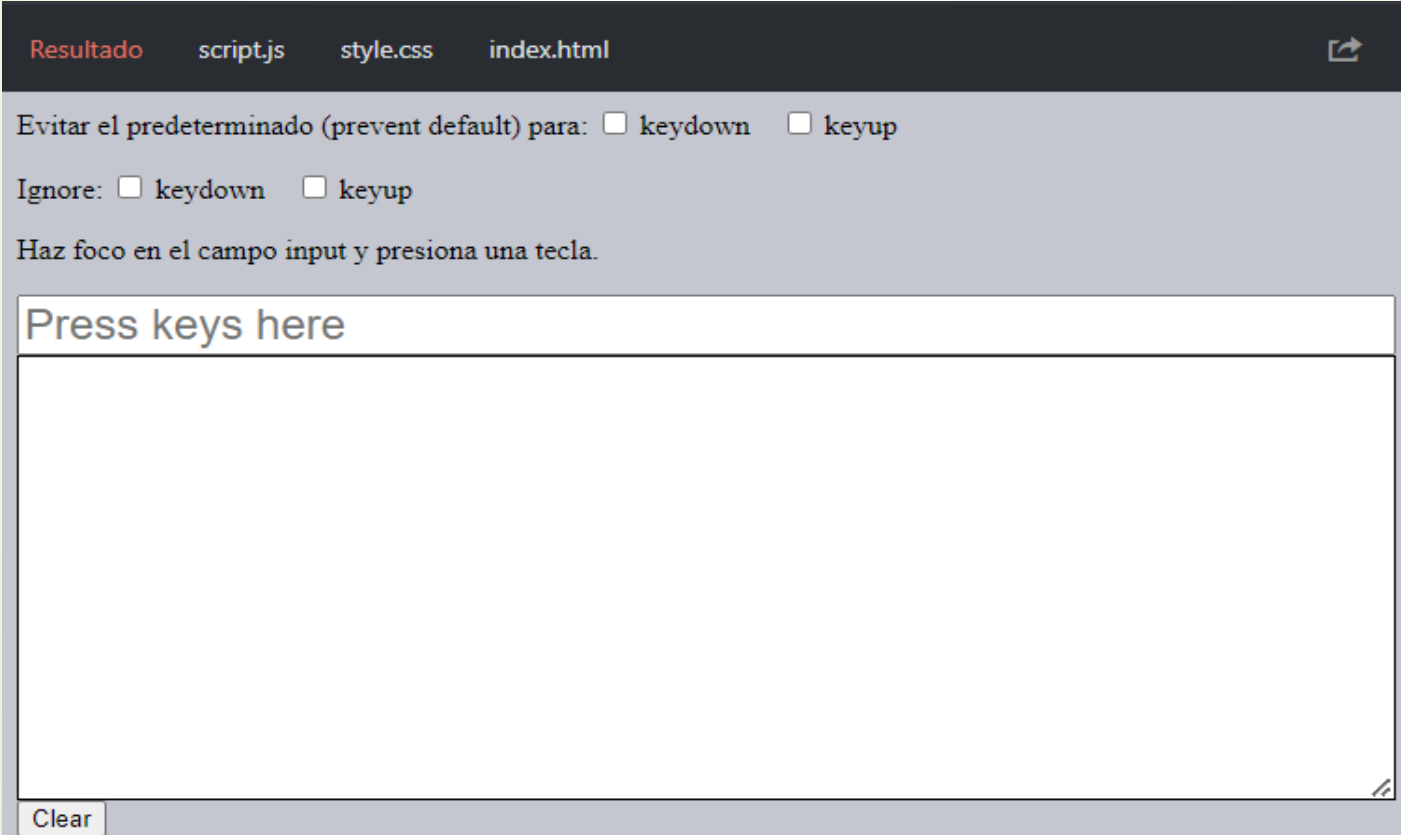
Teclado: keydown y keyup

- Antes de llegar al teclado, por favor ten en cuenta que en los dispositivos modernos hay otras formas de “ingresar algo”. Por ejemplo, el uso de reconocimiento de voz (especialmente en dispositivos móviles) o copiar/pegar con el mouse.
- Entonces, si queremos hacer el seguimiento de cualquier ingreso en un campo `<input>`, los eventos de teclado no son suficientes. Existe otro evento llamado `input` para detectar cambios en un campo `<input>` producidos por cualquier medio. Y puede ser una mejor opción para esa tarea. Lo estudiaremos más adelante, en el capítulo Eventos: `change`, `input`, `cut`, `copy`, `paste`.
- Los eventos de teclado solo deberían ser usados cuando queremos manejar acciones de teclado (también cuentan los teclados virtuales). Por ejemplo, para reaccionar a las teclas de flecha Up y Down o a atajos de teclado “hotkeys” (incluyendo combinaciones de teclas).

Teststand

Para entender mejor los eventos de teclado, puedes usar “teststand” aquí abajo. Prueba diferentes combinaciones de tecla en el campo de texto.

Ejemplo → <https://plnkr.co/edit/rbCCCj4wX5fcuBh8?p=preview&preview>



The screenshot shows a web application interface for testing keyboard events. At the top, there is a dark header bar with tabs labeled "Resultado" (highlighted in red), "script.js", "style.css", and "index.html". To the right of these tabs is a share icon. Below the header, the main content area has a light gray background. It contains the following text and controls:

- "Evitar el predeterminado (prevent default) para: ☐ keydown ☐ keyup"
- "Ignore: ☐ keydown ☐ keyup"
- "Haz foco en el campo input y presiona una tecla."
- A large text input field with the placeholder text "Press keys here".
- A "Clear" button at the bottom left of the input field.

KEYDOWN Y KEYUP

Los eventos keydown ocurren cuando se presiona una tecla, y **keyup** cuando se suelta.

event.code y **event.key**

La propiedad `key` del objeto de evento permite obtener el carácter, mientras que la propiedad `code` del evento permite obtener el “código físico de la tecla”. Por ejemplo, la misma tecla Z puede ser presionada con o sin Shift. Esto nos da dos caracteres diferentes: z minúscula y Z mayúscula.

event.key es el carácter exacto, y será diferente. Pero **event.code** es el mismo:

Tecla	<code>event.key</code>	<code>event.code</code>
Z	z (minúscula)	KeyZ
Shift+Z	Z (mayúscula)	KeyZ

Si un usuario trabaja con diferentes lenguajes, el cambio a otro lenguaje podría producir un carácter totalmente diferente a "Z". Este se volverá el valor de `event.key`, mientras que `event.code` es siempre el mismo: "KeyZ".

¿Qué pasa si una tecla no da ningún carácter? Por ejemplo, Shift o F1 u otras. Para estas teclas, `event.key` es aproximadamente lo mismo que `event.code`:

Key	<code>event.key</code>	<code>event.code</code>
F1	F1	F1
Backspace	Backspace	Backspace
Shift	Shift	ShiftRight or ShiftLeft

Ten en cuenta que `event.code` especifica con exactitud la tecla que es presionada. Por ejemplo, la mayoría de los teclados tienen dos teclas Shift: una a la izquierda y otra a la derecha. `event.code` nos dice exactamente cuál fue presionada, en cambio `event.key` es responsable del “significado” de la tecla: lo que “es” (una “Mayúscula”).

Digamos que queremos manejar un atajo de teclado: Ctrl+Z (o Cmd+Z en Mac). La mayoría de los editores de texto “cuelgan” la acción “Undo” en él. Podemos configurar un “listener” para escuchar el evento `keydown` y verificar qué tecla es presionada.

Hay un dilema aquí: en ese “listener”, ¿debemos verificar el valor de `event.key` o el de `event.code`? Por un lado, el valor de `event.key` es un carácter que cambia dependiendo del lenguaje. Si el visitante tiene varios lenguajes en el sistema operativo y los cambia, la misma tecla dará diferentes caracteres. Entonces tiene sentido chequear `event.code` que es siempre el mismo. Como aquí:

```
1 document.addEventListener('keydown', function(event) {
2   if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
3     alert('Undo!')
4   }
5 });
```

Por otro lado, hay un problema con `event.code`. Para diferentes distribuciones de teclado, la misma tecla puede tener diferentes caracteres. Por ejemplo, aquí abajo mostramos la distribución de EE.UU. “QWERTY” y la alemana “QWERTZ” (de Wikipedia):



Para la misma tecla, la distribución norteamericana tiene “Z”, mientras que la alemana tiene “Y” (las letras son intercambiadas). Efectivamente, `event.code` será igual a `KeyZ` para las personas con distribución de teclas alemana cuando presionen Y.

Si chequeamos `event.code == 'KeyZ'` en nuestro código, las personas con distribución alemana pasarán el test cuando presionen Y. Esto suena realmente extraño, y lo es. La especificación explícitamente menciona este comportamiento.

Entonces, ***event.code*** puede coincidir con un carácter equivocado en una distribución inesperada. Las mismas letras en diferentes distribuciones pueden mapear a diferentes teclas físicas, llevando a diferentes códigos. Afortunadamente, ello solo ocurre en algunos códigos, por ejemplo `keyA`, `keyQ`, `keyZ` (que ya hemos visto), y no ocurre con teclas especiales como Shift. Puedes encontrar la lista en la especificación.

Para un seguimiento confiable de caracteres que dependen de la distribución, `event.key` puede ser una mejor opción. Por otro lado, `event.code` tiene el beneficio de quedar siempre igual, ligado a la ubicación física de la tecla. Así los atajos de teclado que dependen de él funcionan bien aunque cambie el lenguaje.

¿Queremos manejar teclas que dependen de la distribución? Entonces `event.key` es lo adecuado. ¿O queremos que un atajo funcione en el mismo lugar incluso si cambia el lenguaje? Entonces ***event.code*** puede ser mejor.

Autorepetición

Si una tecla es presionada durante suficiente tiempo, comienza a “autorepetirse”: keydown se dispara una y otra vez, y cuando es soltada finalmente se obtiene keyup. Por ello es normal tener muchos keydown y un solo keyup.

Para eventos disparados por autorepetición, el objeto de evento tiene la propiedad ***event.repeat*** establecida a true.

Acciones predeterminadas

Las acciones predeterminadas varían, al haber muchas cosas posibles que pueden ser iniciadas por el teclado. Por ejemplo:

- Un carácter aparece en la pantalla (el resultado más obvio).
- Un carácter es borrado (tecla Delete).
- Un avance de página (tecla PageDown).
- El navegador abre el diálogo “guardar página” (Ctrl+S)

Evitar la acción predeterminada en `keydown` puede cancelar la mayoría de ellos, con la excepción de las teclas especiales basadas en el sistema operativo. Por ejemplo, en Windows la tecla `Alt+F4` cierra la ventana actual del navegador. Y no hay forma de detenerla por medio de “evitar la acción predeterminada” de JavaScript. Por ejemplo, el `<input>` debajo espera un número telefónico, entonces no acepta teclas excepto dígitos, `+`, `()` or `-`:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || ['+', '(', ')', '-'].includes(key);
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Teléfono, por favor"
type="tel">
```


Aquí el manejador onkeydown usa checkPhoneKey para chequear la tecla presionada. Si es válida (de 0..9 o uno de +-()), entonces devuelve true, de otro modo, false.

Como ya sabemos, el valor false devuelto por el manejador de eventos, asignado usando una propiedad DOM o un atributo tal como lo hicimos arriba, evita la acción predeterminada; entonces nada aparece en <input> para las teclas que no pasan el test. (El valor true no afecta en nada, solo importa el valor false)

Ten en cuenta que las teclas especiales como Backspace, Left, Right, no funcionan en el input. Este es un efecto secundario del filtro estricto que hace checkPhoneKey. Estas teclas hacen que devuelva false. Aliviamos un poco el filtro permitiendo la tecla de flecha [Left], [Right], y [Delete], [Backspace]:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') ||
    ['+', '(', ')', '-', 'ArrowLeft', 'ArrowRight', 'Delete', 'Backspace'].includes(key);
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Teléfono, por favor"
type="tel">
```


Código heredado

En el pasado existía un evento ***keypress***, y también las propiedades del objeto evento ***keyCode***, ***charCode***, ***which***.

Al trabajar con ellos había tantas incompatibilidades entre los navegadores que los desarrolladores de la especificación no tuvieron otra alternativa que declararlos obsoletos y crear nuevos y modernos eventos (los descritos arriba en este capítulo). El viejo código todavía funciona porque los navegadores aún lo soportan, pero no hay necesidad de usarlos más, en absoluto.

RESUMEN

- Presionar una tecla siempre genera un evento de teclado, sean teclas de símbolos o teclas especiales como Shift o Ctrl y demás. La única excepción es la tecla Fn que a veces está presente en teclados de laptops. No hay un evento de teclado para ella porque suele estar implementado en un nivel más bajo que el del sistema operativo.
- Eventos de teclado:
 - **keydown** – al presionar la tecla (comienza a autorepetir si la tecla queda presionada por un tiempo),
 - **keyup** – al soltar la tecla.
- Principales propiedades de evento de teclado:
 - **code** – el código de tecla “key code” (“KeyA”, “ArrowLeft” y demás), especifica la ubicación física de la tecla en el teclado.
 - **key** – el carácter (“A”, “a” y demás). Para las teclas que no son de caracteres como Esc, suele tener el mismo valor que code.
- En el pasado, los eventos de teclado eran usados para detectar cambios en los campos de formulario. Esto no es confiable, porque el ingreso puede venir desde varias fuentes. Para manejar cualquier ingreso tenemos los eventos input y change (tratados en el capítulo Eventos: change, input, cut, copy, paste). Ellos se disparan después de cualquier clase de ingreso, incluyendo copiar/pegar y el reconocimiento de voz.

TAREAS

Extendiendo atajos de teclado

Crea una función `runOnKeys(func, code1, code2, ... code_n)` que ejecute `func` al presionar simultáneamente las teclas con códigos `code1, code2, ..., code_n`.

Por ejemplo, el siguiente código muestra un `alert` cuando "Q" y "W" se presionan juntas (en cualquier lenguaje, con o sin mayúscula)

```
runOnKeys(  
  () => alert("¡Hola!"),  
  "KeyQ",  
  "KeyW"  
);
```

[Demo en nueva ventana](#)