PRÁCTICA 9: INTERFACES GRAFICAS DE USUARIO – DELEGACION DE EVENTOS

Duración estimada: 100 min.

Objetivos:

- **Eventos en JavaScript.**

Bibliografía: ***Internet***

Funciones anónimas y funciones flecha

Funciones por expresión

```
// El segundo "saludar" (nombre de la función) se suele omitir: es redundante
const saludo = function saludar() {
  return "Hola";};
saludo(); // 'Hola'
```

En Javascript es muy habitual encontrarse códigos donde los programadores «guardan funciones» dentro de variables, para posteriormente «ejecutar dichas variables». Se trata de un enfoque diferente, creación de funciones por expresión, que fundamentalmente, hacen lo mismo con algunas diferencias:

Con este nuevo enfoque, estamos creando una función en el interior de una variable, lo que nos permitirá posteriormente ejecutar la variable (como si fuera una función). Observa que el nombre de la función (en este ejemplo: saludar) pasa a ser inútil, ya que si intentamos ejecutar saludar() nos dirá que no existe y si intentamos ejecutar saludo() funciona correctamente.

¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las funciones anónimas (o funciones lambda).

Funciones anónimas

Las funciones anónimas o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
  return "Hola";
};
saludo; // ƒ () { return 'Hola'; }
saludo(); // 'Hola'
```

Observa que en la última línea del ejemplo anterior, estamos ejecutando la variable, es decir, ejecutando la función que contiene la variable. Sin embargo, en la línea anterior hacemos referencia a la variable (sin ejecutarla, no hay paréntesis) y nos devuelve la función en sí.

Funciones flecha

Las Arrow functions, funciones flecha o «fat arrow» son una forma corta de escribir funciones que aparece en Javascript a partir de ECMAScript 6. Básicamente, se trata de reemplazar eliminar la palabra function y añadir => antes de abrir las llaves:

```
const func = function () {
  return "Función tradicional.";
};

const func = () => {
  return "Función flecha.";
};
```

Sin embargo, las funciones flechas tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves ({}).
- Además, en ese caso, automáticamente se hace un return de esa única línea, por lo que podemos omitir también el return.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: () =>.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: e =>.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: (a, b) =>.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: ({name: 'Manz'}).

```
const func = () => "Función flecha."; // 0 parámetros: Devuelve "Función flecha"
const func = (e) => e + 1; // 1 parámetro: Devuelve el valor de e + 1
const func = (a, b) => a + b; // 2 parámetros: Devuelve el valor de a + b
```

Eventos de teclado

Existen tres eventos distintos relacionados con pulsar una tecla. Cuando pulsamos una tecla se producen tres eventos, que enunciados por orden son:

- keydown,
- keypress,
- keyup.

Hay dos tipos de códigos diferentes: los códigos de teclado (que representan la tecla del teclado que ha sido pulsada, tecla física) y los códigos de caracteres (número asociado a cada carácter, según el juego de caracteres Unicode). Tener en cuenta que desde el punto de vista de qué tecla física ha sido pulsada es lo mismo escribir una a que una A, ya que en ambos casos pulsamos la misma tecla en el teclado. En cambio desde el punto de vista del carácter, a es un carácter y A es otro carácter diferente (y cada uno tiene su código numérico diferente).

Recomendaciones para el manejo de eventos del teclado

- **Regla 1:** para determinar el carácter resultado de la pulsación de una tecla simple o de una combinación de teclas usaremos el evento **keypress** (no los eventos keydown ni keyup), determinando la tecla pulsada con la propiedad wich del objeto event.
- **Regla 2:** para determinar si ha sido pulsada una tecla no imprimible (como flecha de cursor, etc.), usaremos el evento **keyup**, determinando la tecla pulsada con la propiedad **keyCode** (código de la tecla física).
- **Regla 3**: no intentes controlar todo lo que ocurre con el teclado (porque quizás te vuelvas loco). Controla únicamente aquello que sea necesario y útil.

RECUPERAR UN CARÁCTER A PARTIR DE SU CÓDIGO NUMÉRICO

Cada carácter tiene un código numérico. Guiándonos por el juego de caracteres Unicode los códigos numéricos son 65 para la letra A (mayúscula), 66 para la letra B, …, 97 para la letra a (minúscula).

Teclas especiales como **ENTER** también tienen asignados códigos numéricos: 13 para **ENTER**, 37 **flecha izquierda del cursor**, 38 **flecha arriba del cursor**, 39 **flecha derecha del cursor**, 40 **flecha abajo del cursor**, etc.

Para recuperar el carácter asociado a un código usaremos un método estático del objeto predefinido String, con la siguiente sintaxis: String.fromCharCode(num1, num2, ..., numN);

PROPIEDADES Y MÉTODOS DEL OBJETO EVENT RELACIONADOS CON TECLADO

Las propiedades y métodos del objeto **Event** relacionados con el teclado se resumen en la siguiente tabla.

Tipo	Nombre	Descripción aprenderaprogramar.com
Propiedades relacionadas con el teclado	altKey	Contiene el valor booleano true si la tecla ALT estaba pulsada cuando se produjo el evento, o false en caso contrario.
	ctrlKey	Contiene el valor booleano true si la tecla CTRL estaba pulsada cuando se produjo el evento, o false en caso contrario.
	shiftKey	Contiene el valor booleano true si la tecla SHIFT estaba pulsada cuando se produjo el evento, o false en caso contrario.
	charCode	Devuelve el código del carácter unicode generado por el evento keypress. Se recomienda usar wich en lugar de charCode.

EJEMPLO: TRABAJANDO CON EVENTOS DE TECLADO: Escribe el siguiente código y comprueba sus resultados:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html><head><title>Ejemplo aprenderaprogramar.com</title><meta charset="utf-8">
<script type="text/javascript">
var eventoControlado = false;
window.onload = function() {
document.onkeypress = mostrarInformacionCaracter;
document.onkeyup = mostrarInformacionTecla; }

function mostrarInformacionCaracter(evObject) {
var msg = '';
```

```
var elCaracter = String.fromCharCode(evObject.which);

 if (evObject.which!=0 && evObject.which!=13) {
 msg = 'Tecla pulsada: ' + elCaracter;
 control.innerHTML += msg + '----------<br/>'; }
 else
{
msg = 'Pulsada tecla especial';
 control.innerHTML += msg + '----------<br/>';}
 eventoControlado=true;
}
function mostrarInformacionTecla(evObject) {
 var msg = ''; var teclaPulsada = evObject.keyCode;
 if (teclaPulsada == 20) {
msg = 'Pulsado caps lock (act/des mayúsculas)';}
 else if (teclaPulsada == 16) {
msg = 'Pulsado shift';}
 else if (eventoControlado == false) {
msg = 'Pulsada tecla especial';}
 if (msg) {
control.innerHTML += msg + '----------<br/>';}
 eventoControlado = false;
}
</script>
</head>
<body><div id="cabecera"><h2>Cursos aprenderaprogramar.com</h2><h3>Ejemplo JavaScript: pulse una
tecla</h3></div>
<div id="control"> </div>
</body></html>
```

Delegación de Eventos

1. **Definición**

   Delegación de eventos es un mecanismo a través del cual evitamos asignar **event listeners** a múltiples nodos específicos del DOM, asignando un **event listener** a solo un nodo padre que contiene el resto de estos nodos.

La delegación de eventos consiste en escuchar los eventos en el elemento padre para capturarlo cuando ocurra en sus hijos.

La captura y la propagación nos permiten implementar uno de los más poderosos patrones de manejo de eventos llamado delegación de eventos. La idea es que si tenemos muchos elementos manejados de manera similar podemos, en lugar de asignar un manejador a cada uno de ellos, poner un único manejador a su ancestro común.

Ejemplo:

```
const button = document.querySelector(".btn");

const changeTextButton = () => {

 button.classList.toggle("clicked");

};

button.addEventListener("click", changeTextButton);
```

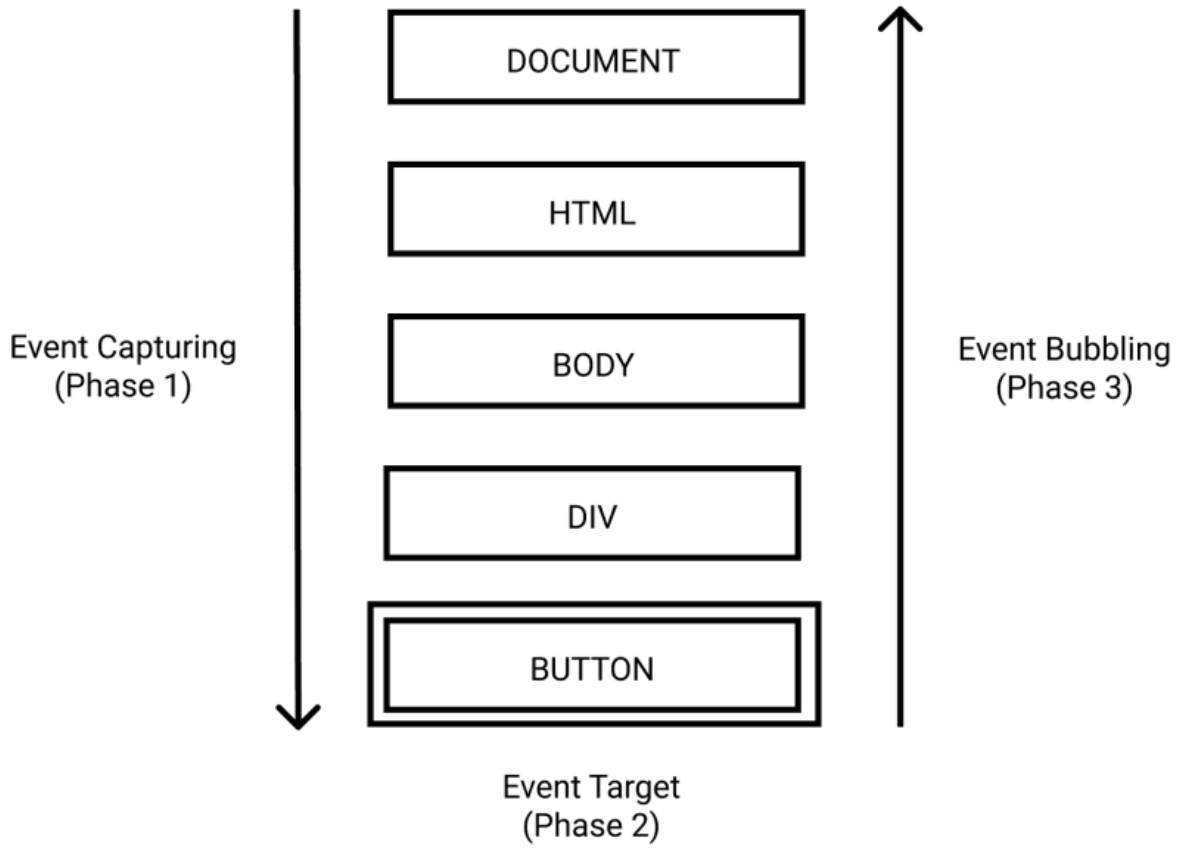
El método **addEventListener** es la forma más habitual que usamos para registrar eventos. Recibe dos parámetros: el tipo de evento que queremos “escuchar” y la acción (o función) que queremos ejecutar cuando el evento suceda.

2. ¿Cómo funcionan los eventos?

Cuando clicamos en un botón, se desencadenan las siguientes fases:

- **Fase de captura:** el evento empieza en el **window**, **document** y en el **root** y luego entra dentro de cada uno de los hijos.
- **Fase target:** el evento se lanza sobre el elemento en el que se hace click.
- **Fase bubble:** finalmente el evento sube hacia los ancestros desde el target hasta llegar de nuevo al **root**, **document** y **window**.

En esta imagen podemos ver cómo se propagan las diferentes fases de los eventos.



Desde el punto de vista del código, un evento no es más que un objeto donde se almacena información relacionada con el mismo.

Algunas de las propiedades que podemos encontrar en el objeto evento son las siguientes:

- **event.type**: es el tipo de evento que hemos lanzado, en el caso del ejemplo sería el **click**, pero podría ser cualquier otro.
- **event.target:** es el ***elemento en el que se ha disparado el evento***.
- **defaultPrevented:** indica si el evento puede ser cancelado con event.preventDefault.
- **event.Phase:** indica qué fase del flujo de eventos se está procesando.
- **event.currentTarget:** es el elemento en el que declaramos el evento, es decir, en que hemos declarado el addEventListener.

Ahora que ya sabemos qué son los eventos y cómo funcionan, vamos a ver cómo podemos trabajar con ellos. Veamos este ejemplo en el que mostramos una tabla con varias filas y columnas y, en cada celda, un botón de borrar. En este caso, siguiendo el ejemplo anterior, tendríamos que hacer algo similar a esto para capturar el evento de cada botón:

```
const buttons = document.querySelectorAll(‘.btn_deleted’);
buttons.forEach(button => {
   button.addEventListener('click', () => console.log('Clicked!'));
});
```

¿Qué tendríamos en este caso?

- Tendríamos un evento por cada botón. Imaginaros que tenemos 40 filas, tendríamos 40 botones con 40 eventos.
- Además, si esta tabla tuviese paginación, cada vez que cambiemos de página tendríamos que volver a declarar los eventos, ya que al cambiar de página, los perderíamos.
- Otro problema añadido es que si alguna de esas celdas se renderiza de forma dinámica tendríamos que esperar a que el elemento estuviera renderizado en el DOM ya que si el botón no existe aún no se puede declarar el evento asociado.

Para ayudarnos con estos dos problemas podemos usar la técnica de **delegación de eventos.**

La **delegación de eventos** consiste en declarar un evento en el contenedor del elemento sobre el que queremos lanzarlo. Al estar declarado en el padre, ¿cómo sabe cuándo tiene que lanzarse el evento al hacer click en un botón determinado?

Cuando lanzamos un evento, ***el navegador crea un objeto evento, donde recoge las propiedades de dicho evento***. Utilizando dicha información, comprobamos a través del ***target*** cuál es el elemento que lanza el evento. En este caso, lo que hacemos es comprobar si el target tiene la clase ‘buttonClass’ y si la tiene se lanzará el evento.

Para entenderlo de forma fácil, **con la delegación de eventos tenemos un elemento que está continuamente escuchando si se lanza o no un evento y que tendrá que cumplir la condición que le pongamos**.

Como podemos ver, la delegación de eventos es útil para escuchar eventos en múltiples elementos, con tan solo un manejador. Al delegar en el padre, los eventos estarán siempre ahí. Desde el punto de vista del rendimiento de nuestra aplicación, puede parecer contraproducente tener un elemento escuchando todos los eventos, pero es justo todo lo contrario No tiene el mismo coste lanzar 40 eventos que uno, con lo cual nuestra performance mejorará

3. Event Bubbling

En JavaScript existe un concepto denominado **event bubbling** que se da sobre todo con nodos del DOM que están anidados unos dentro de otros.

Se da cuando activamos el evento de un elemento, y si su nodo padre tiene registrado otro evento, este último se activara automáticamente y así ira escalando en la jerarquía del DOM.

- El **event bubbling** es una caracteristica un poco extraña pero necesaria de comprender para hacer apps mas robustas.
- El **event bubbling** tiene tres faces: captura, target y bubbling.
- El **event bubbling** se da desde lo mas profundo del arbol DOM y sale hacia los elementos mas externos.
- El **modo captura** se da desde los elementos mas externos y va entrando a los elementos mas profundos.
- El modo captura se activa pasando al listener un tercer parámetro que es un objeto.
- La mejor manera de detener una propagación de eventos es usando el método **stopPropagation()**

```
button.addEventListener("click", (e) => {
        e.stopPropagation()
        alert("Clicked!")
})
```

Ejemplo:

CSS

```
const button = document.querySelector("#button");
const hijo = document.querySelector("#hijo");

button.addEventListener("click", () => {
  alert("Clicked!")
})

hijo.addEventListener("click", () => {
  alert("Clicked hijo!")
})
```

HTML

```
<div id="padre">
  <div id="hijo">
    <button id="button">
      Click me!
    </button>
  </div>
</div>
```

4. Event Capturing

JavaScript es un lenguaje de programación orientado a eventos, que son los elementos fundamentales para reaccionar a las acciones del usuario en la web.

El concepto del DOM se refiere a un árbol jerárquico de etiquetas o nodos HTML. Es decir, se manejan nodos dentro de nodos. Esto implica que un evento insertado en un nodo determinado afectará de un modo u otro a los demás nodos en su rama, ya sean sus padres o sus hijos.

El concepto de **event bubbling** en el DOM dice que la propagación de un evento generalmente sucede hacia arriba en el árbol jerárquico. Es decir, si haces clic en un elemento que está dentro de otro, estás a su vez haciendo clic en el elemento contenedor, su etiqueta HTML padre. Este comportamiento de tipo **event bubbling en el DOM es de lo más cotidiano.**

Como sabemos esto nos permitirá hacer que un nodo que no tiene necesariamente un evento pueda escucharlo y reaccionar a él.

Si queremos hacer que se haga a la inversa, hacia abajo, lo logramos con el comportamiento **event capturing**

Para hacer un **event capturing** en el DOM, deberemos incluir un parámetro más dentro del método addEventListener.

Entonces, para crear un event capturing en el DOM, necesitamos un tercer parámetro dentro del método **addEventListener**. ¿Cuál es este parámetro? Pues un simple valor booleano **true**. Este valor hace que la propagación del evento capturado se desarrolle al revés, es decir, de arriba hacia abajo.

A continuación, te mostramos unas cuantas líneas de código que nos ayudarán a entender el concepto de event capturing en el DOM. Allí podrás ver que estamos creando un elemento HTML div que contiene, a su vez, un elemento HTML de tipo p o párrafo. Esta línea la estaremos creando directamente sobre nuestro archivo HTML:

*** I'm a batman ***

Ahora, podemos usar el método para seleccionar nodos en el DOM getElementById para seleccionar los nodos anteriores directamente desde un archivo JavaScript:

document.getElementById(«p»).addEventListener («click, function ( ) {alert («p»);), true);
document.getElementById(«div»).addEventListener («click, function ( ) {alert («div»);), true);

Las líneas de código anteriores harán que se genere un **event capturing** en el DOM, sobrescribiendo el comportamiento natural de la propagación de eventos. En el contexto de este ejemplo, esto haría que primero se ejecute el alert del elemento HTML div y luego el del elemento HTML p. El motivo de esto es que la propagación se ejecutará de arriba hacia abajo, es decir, primero se dará el evento en el padre (div) y luego en el hijo (p).

El concepto de event capturing en el DOM suele ser más teórico que práctico. Esto significa que un desarrollador web normalmente no suele usar este cambio de propagación, a menos de que sea en momentos muy puntuales.

5. EJEMPLO DELEGACIÓN DE EVENTOS. COMO PODEMOS SOLUCIONAR UN PROBLEMA EN EL QUE VARIOS ELEMENTOS EJECUTEN LA MISMA FUNCIÓN CON Y SIN DELEGACIÓN DE EVENTOS.

Si queremos que varios elementos escuchen el mismo **eventType** y ejecuten el mismo **eventHandler,** que es la función que se ejecutará cuando el eventType se dispare. (por ejemplo, en una galería de fotos poder hacer clic en cualquiera de las miniaturas para mostrar la versión ampliada), nuestro primer razonamiento sería iterar sobre ellos.

```
var galleryImg = document.querySelectorAll('.gallery-item');

for(let i = 0; i < galleryImg.length; i++) {
  galleryImg[i].addEventListener('eventType', () =>
    // eventHandler
  )
}
```

Sin embargo, el principal problema con estos enfoques es que **solo** aplica a los **elementos existentes en el DOM** al cargar la página. Con lo cual, si añadimos elementos **dinámicamente**, estos no se verán afectados por el evento.

Por otro lado, si evitamos la iteración, podemos ganar rendimiento. Para eso utilizamos la **Delegación de eventos** que ***consiste en escuchar el evento en el elemento padre solamente para luego capturarlo cuando ocurra en sus hijos***. Esto gracias a un comportamiento de los eventos llamado **Bubbling**.

Bubbling

El bubbling (burbujeo) es una fase de los eventos (la otra es **capturing**) que significa que cuando un evento ocurre en el DOM, es capturado en el elemento HTML más profundo posible, y de ahi se va elevando hacia sus padres en orden de jerarquía hasta llegar al objeto global (window). Por lo tanto, cuando un evento ocurre en un elemento, también está ocurriendo en sus padres.

Delegación de eventos

Ahora que sabemos que si un evento ocurre en un hijo, también está ocurriendo en sus padres. Volvamos al ejemplo de la galería. Ahora no necesitamos iterar sobre los elementos, sino escuchar el evento en el padre.

Tomemos en cuenta el siguiente HTML

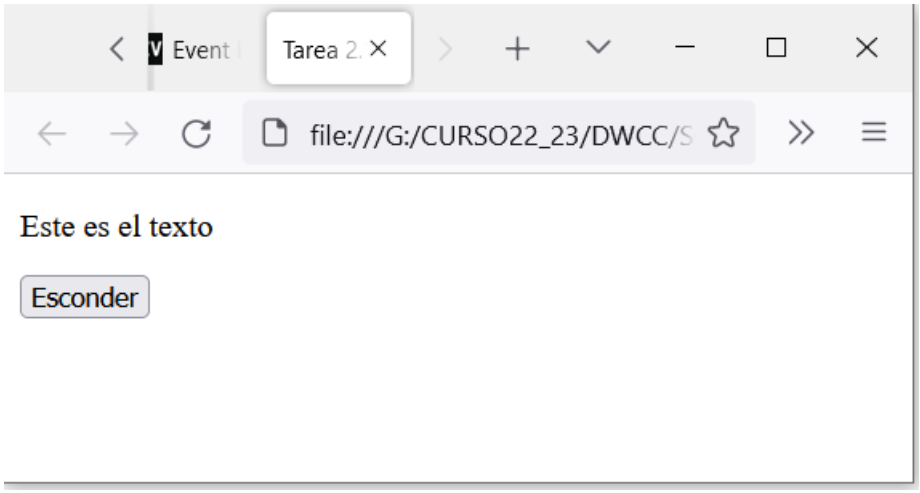
```
<div class="gallery-container">
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
  <a href="" class="gallery-item"><img src="" alt=""></a>
</div>
```

Tenemos nueve elementos gallery-item. Pero podrían ser más. Sin embargo, como ya conocemos la delegación, escucharemos el evento una sola vez en el padre. Recuerda que podemos pasar el objeto Eventcomo parámetro (lo nombraremos e en nuestros ejemplos), al event Handler para asi obtener el target que es el elemento que dispara el evento.

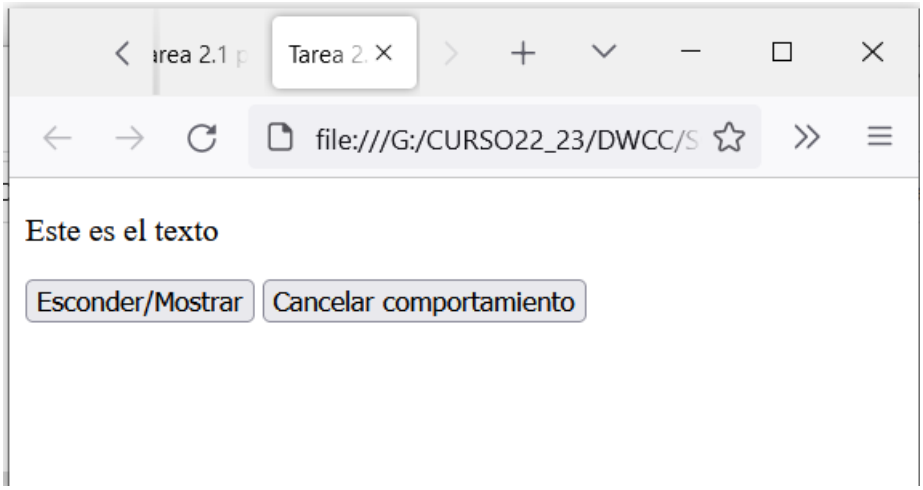
```
let gallery = document.querySelector('.gallery-item');

gallery.addEventListener('eventType', e =>
  console.log(e.target) // nos imprime en consola el elemento que dispara el evento
);
```

Secuencia / Desarrollo:

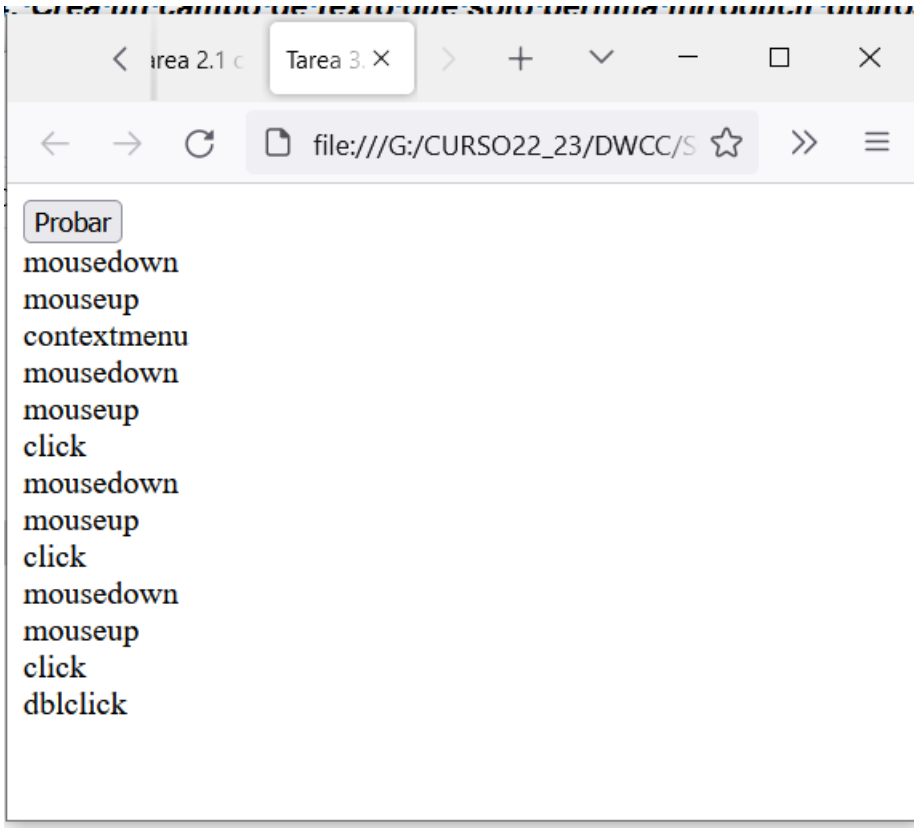
1. **Ejercicio 1: Utilizando todas las formas de asignación de eventos, crea un botón que esconda un texto y otro que lo muestre.**



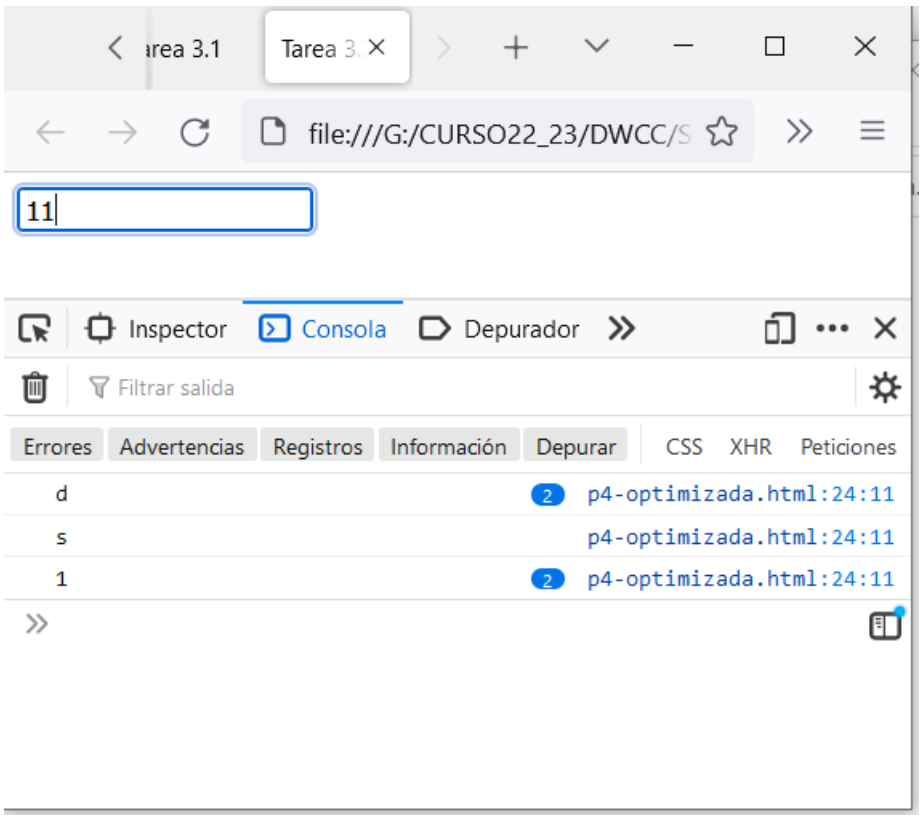
2. **Ejercicio 2. Modifica el ejercicio anterior y en la opción de detectores de eventos, cambia el botón para que esconda/muestre el texto. Añade otro botón que desactive el primero.**



3. **Ejercicio 3: Muestra en un textarea los eventos producidos por el ratón en un botón (mousedown, mouseup, click, contextmenu, dblclick).**



4. ***Ejercicio 4: Crea un campo de texto que sólo permita introducir dígitos.***



5. ***Ejercicio 5: Crea un div que contenga un textarea en el que:***
   - ***Tengamos que pulsar Ctrl-E para escribir en él.***
   - ***Ctrl-S para guardar y***
   - ***Esc para salir sin guardar.***