PRACTICA 3 – UNIDAD 6 – UNIDAD 5

Duración estimada: 300 min.

Objetivos:

- **Eventos en JavaScript.**
- **Manejar estructura DOM**
- **Manejar estructura BOM**

Bibliografía: ***Internet***

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

Para utilizar los eventos de teclado en JavaScript, debes seguir los siguientes pasos:

- Detectar el evento de teclado
- Capturar la tecla pulsada
- Ejecutar la acción correspondiente

Para detectar el evento de teclado, debes usar el método **addEventListener()** y especificar el evento de teclado que deseas capturar, como "**keydown**", "**keyup**" o "**keypress**".

Detectar el evento de teclado

Diferencias entre keydown, keyup o keypress

- Keydown se lanza cuando se pulsa una tecla
- Keyup cuando se suelta la tecla
- Keypress cuando la tecla pulsada produce el valor

Esto quiere decir que por ejemplo si pulsamos la tecla **CTRL**, el evento **Keypress** no se va a ejecutar al no producir valor.

- Entre otras cosas es por eso que **Keypress *se considera ya obsoleto***.
- Cuando quieras capturar eventos deberías usar Keydown o Keyup.
- Así tendrás todo el rango de posibilidades cubierto.

Aquí tienes un ejemplo de cómo utilizar los eventos de teclado en JavaScript para mostrar u ocultar un menú cuando pulsa la tecla Escape:

```
document.addEventListener("keydown", function(event) {
  if (event.key === "Escape") {
    document.getElementById("menu").classList.toggle("hidden");
  }
});
```

Capturar la tecla pulsada : keyCode o key

Una vez que detectamos el evento, es hora de capturar la tecla pulsada. Para ello anteriormente se usaba la propiedad **keyCode**, pero es algo que ***ya con el tiempo se ha ido quedando obsoleta.***Lo ideal es usar la propiedad key del evento.La propiedad **key** nos va a indicar que tecla está siendo presionada

```
if (event.key === "Escape")
```

Cómo crear atajos de teclado personalizados

Crear atajos de teclado personalizados es una forma de mejorar la eficiencia del usuario en tu sitio web.Puedes crear atajos para cualquier acción que desees, como enviar un formulario, cambiar de página o mostrar un modal.

Para crear un atajo de teclado personalizado, debes detectar la combinación de teclas que deseas utilizar y luego ejecutar la acción correspondiente.

Ya que se consideran teclas modificadores, **ALT**, **CTRL** o **SHITF** tienen su propia propiedad que podremos escuchar

altKey, ctrlKey y shiftKey

Aquí tienes un ejemplo de cómo crear un atajo de teclado personalizado en JavaScript para enviar un formulario cuando se pulse **CTRL + ENTER**

```
document.addEventListener("keydown", function(event) {
  if (event.ctrlKey && event.key === "Enter") {
    document.getElementById("myForm").submit();
  }
```

```
});
```

Cómo detectar combinaciones de teclas

En algunos casos, es posible que desees detectar una combinación de teclas en lugar de una sola tecla.

Por ejemplo, podrías querer detectar la combinación "**Ctrl+C"** para copiar texto o **"Ctrl+V"** para pegar texto.

Para detectar combinaciones de teclas, debes utilizar condicionales que comprueben si se han pulsado varias teclas a la vez.

Aquí un ejemplo de cómo detectar la combinación "Ctrl+C" en JavaScript:

```
document.addEventListener("keydown", function(event) {
  if (event.ctrlKey && event.key === "c") {
    // Ejecutar la acción correspondiente
  }
});
```

Cómo detectar la tecla de escape

La tecla de escape es una de las teclas más utilizadas en los eventos de teclado. Por lo general, se utiliza para cerrar una ventana modal, salir de un menú o cancelar una operación.

En JavaScript, puedes detectar la tecla de escape utilizando también la propiedad key.

```
document.addEventListener("keydown", function(event) {
  if (event.key === "Escape") {
    // Ejecutar la acción correspondiente
  }
});
```

Detener la acción predeterminada

En algunos casos, puede que necesites evitar que el navegador ejecute la acción predeterminada de una tecla. Para ello, puedes usar el método preventDefault().

Ejemplo de código:

```
document.addEventListener('keydown', function(event) {
    if (event.key === 'Enter') {
        event.preventDefault(); // Evita el comportamiento predeterminado
        console.log('¡Enter fue presionada, pero la acción predeterminada fue evitada!');
    }
});
```

Mejores prácticas para manejar eventos de teclado

1. Usar debounce y throttle

Cuando trabajas con eventos de teclado en tiempo real, como en la búsqueda de autocompletado, es buena práctica implementar técnicas como debounce y throttle para optimizar el rendimiento. Estas técnicas limitan la cantidad de veces que una función se puede ejecutar en respuesta a un evento.

2. Usar nombres de teclas (Key Names)

Siempre que sea posible, utiliza event.key en vez de los códigos de tecla. Los nombres de teclas son más legibles y menos propensos a errores.

3. Accesibilidad

Asegúrate de que tus manejadores de eventos de teclado consideren la accesibilidad. Permite que los usuarios naveguen por tu aplicación usando solamente el teclado y proporciona alternativas para las funciones que dependen de interacciones de teclado.

Throttle y Debounce

Throttle y **debounce** son técnicas utilizadas para controlar con qué frecuencia se ejecuta una función, especialmente cuando se manejan eventos que pueden activarse rápidamente, por ejemplo: el desplazamiento del ratón, el redimensionamiento de la pantalla o la escritura con el teclado.

Ambas técnicas ayudan a mejorar el rendimiento y a evitar llamadas innecesarias a funciones.

¿Qué diferencia hay entre Throttle y Debounce?

Throttle asegura que una función se ejecute a intervalos regulares, sin importar cuántas veces ocurra el evento. Esto es útil cuando quieres que una función se ejecute periódicamente, como al actualizar algo en la pantalla mientras el usuario está desplazándose con el ratón. El movimiento del ratón puede emitir muchísimos eventos, si para cada evento se dispara una función, la técnica throttle asegura que este evento sólo se lanzará en intervales definidos.

Debounce, por otro lado, espera hasta que el evento haya dejado de activarse durante un tiempo especificado antes de ejecutar la función. Es particularmente útil en escenarios como el manejo de entradas de búsqueda o campos autocompletables, donde deseas esperar a que el usuario termine de escribir antes de realizar una acción, como obtener resultados.

Throttle

La función throttle asegura que una función se ejecuta como mucho una vez en el intervalo especificado sin importar cuantas veces el evento se haya disparado.

En este ejemplo se llamará **logThrottle** cada 1 segundo aunque el intervalo simulado sea cada 200ms.

```
function throttle(func, delay) {
  let lastTime = 0;

  return function (...args) {
    const now = Date.now();
    if (now - lastTime >= delay) {
      func.apply(this, args);
      lastTime = now;
    }
  };
}

// Ejemplo de uso:
function logThrottle() {
  console.log("función llamada!");
}

const throttledLog = throttle(logThrottle, 1000); // Llama como mucho una vez cada 1 segundo

// Simular llamadas rápidas
setInterval(throttledLog, 200);
```

Debounce

La función **debounce** asegura que una función se ejecute solo después de que haya pasado un tiempo específico desde la última vez que se activó.

```
function debounce(func, delay) {
  let timer;

  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

// Ejemplo de uso:
function logDebounce() {
  console.log("función llamada!");
}

const debouncedLog = debounce(logDebounce, 1000); // Se ejecuta 1 segundo después de la última llamada

// Simular llamadas rápidas
let counter = 0;
const interval = setInterval(() => {
  debouncedLog();
  counter++;
  if (counter === 10) clearInterval(interval); // Para después de 10 llamadas rápidas
}, 200); // Se dispara cada 200ms
```

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);
```

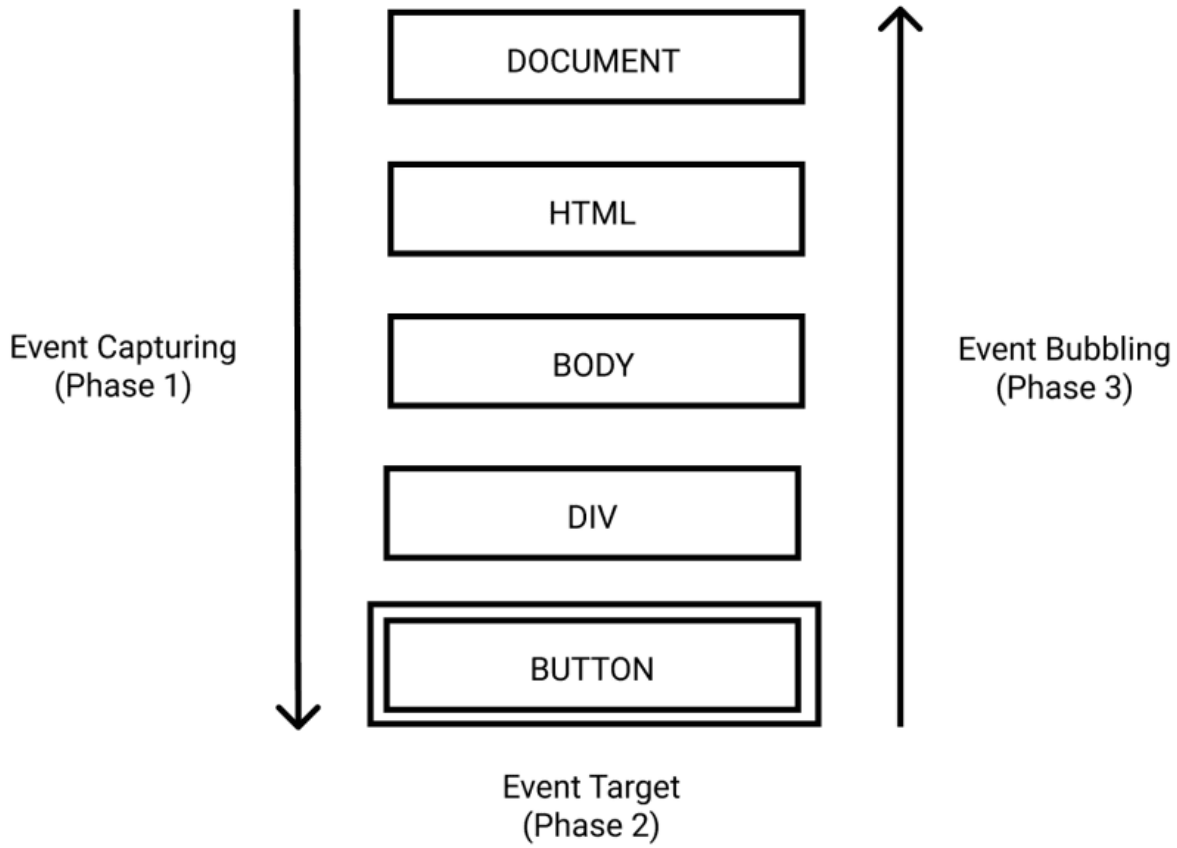
El método **addEventListener** es la forma más habitual que usamos para registrar eventos. Recibe dos parámetros: el tipo de evento que queremos "escuchar" y la acción (o función) que queremos ejecutar cuando el evento suceda.

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>
```

*** ***

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:

```
<div id=»div»> <p id=»p»> I'm a batman </p></div>
```

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 Eventcome 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
);
```

BROWSER OBJECT MODEL

1. Introducción a BOM:

Las versiones 3.0 de los navegadores Internet Explorer y Netscape Navigator introdujeron el concepto de Browser Object Model o BOM, que permite acceder y modificar las propiedades de las ventanas del propio navegador.

Mediante BOM, es posible redimensionar y mover la ventana del navegador, modificar el texto que se muestra en la barra de estado y realizar muchas otras manipulaciones no relacionadas con el contenido de la página HTML.

- Crear, mover, redimensionar y cerrar ventanas de navegador.
- Obtener información sobre el propio navegador.
- Propiedades de la página actual y de la pantalla del usuario.
- Gestión de cookies.
- Objetos ActiveX en Internet Explorer.

2. El objeto window:

El objeto **window** representa la ventana completa del navegador. Mediante este objeto, es posible mover, redimensionar y manipular la ventana actual del navegador. Incluso es posible abrir y cerrar nuevas ventanas de navegador.

Como todos los demás objetos heredan directa o indirectamente del objeto window, no es necesario indicarlo de forma explícita en el código JavaScript. En otras palabras:

```
window.forms[0] === forms[0]

window.document === document
```

BOM define cuatro métodos para manipular el tamaño y la posición de la ventana:

- **moveBy(x, y)** desplaza la posición de la ventana x píxel hacia la derecha y y píxel hacia abajo. Se permiten desplazamientos negativos para mover la ventana hacia la izquierda o hacia arriba.
- **moveTo(x, y)** desplaza la ventana del navegador hasta que la esquina superior izquierda se encuentre en la posición (x, y) de la pantalla del usuario. Se permiten desplazamientos negativos, aunque ello suponga que parte de la ventana no se visualiza en la pantalla.
- r**esizeBy(x, y)** redimensiona la ventana del navegador de forma que su nueva anchura sea igual a (anchura_anterior + x) y su nueva altura sea igual a (altura_anterior + y). Se pueden emplear valores negativos para reducir la anchura y/o altura de la ventana.

- **resizeTo(x, y)** redimensiona la ventana del navegador hasta que su anchura sea igual a x y su altura sea igual a y. No se permiten valores negativos.

Los navegadores son cada vez menos permisivos con la modificación mediante JavaScript de las propiedades de sus ventanas. De hecho, la mayoría de navegadores permite a los usuarios bloquear el uso de JavaScript para realizar cambios de este tipo. De esta forma, una aplicación nunca debe suponer que este tipo de funciones están disponibles y funcionan de forma correcta.

Además de desplazar y redimensionar la ventana del navegador, es posible averiguar la posición y tamaño actual de la ventana. Sin embargo, la ausencia de un estándar para BOM provoca que cada navegador implemente su propio método:

Internet Explorer proporciona las propiedades **window.screenLeft y window.screenTop** para obtener las coordenadas de la posición de la ventana. No es posible obtener el tamaño de la ventana completa, sino solamente del área visible de la página (es decir, sin barra de estado ni menús). Las propiedades que proporcionan estas dimensiones son d**ocument.body.offsetWidth** y **document.body.offsetHeight.**

Los navegadores de la familia **Mozilla**, **Safari** y **Opera** proporcionan las propiedades **window.screenX** y **window.screenY** para obtener la posición de la ventana. El tamaño de la zona visible de la ventana se obtiene mediante **window.innerWidth** y **window.innerHeight**, mientras que el tamaño total de la ventana se obtiene mediante **window.outerWidth** y **window.outerHeight**.

3. Control de tiempos:

Al contrario que otros lenguajes de programación, JavaScript no incorpora un método **wait()** que detenga la ejecución del programa durante un tiempo determinado. Sin embargo, JavaScript proporciona los métodos **setTimeout()** y **setInterval()** que se pueden emplear para realizar tareas similares.

El método **setTimeout()** permite ejecutar una función al transcurrir un determinado periodo de tiempo:

```
setTimeout("console.log('Han transcurrido 3 segundos desde que me programaron')", 3000);

function muestraMensaje() {
  console.log("Han transcurrido 3 segundos desde que me programaron");
}

setTimeout(muestraMensaje, 3000);
```

Como es habitual, cuando s***e indica la referencia a la función no se incluyen los paréntesis***, ya que de otro modo, ***se ejecuta la función en el mismo instante en que se establece el intervalo de ejecución.***

Cuando se establece una cuenta atrás, la función **setTimeout()** devuelve el identificador de esa nueva cuenta atrás. Empleando ese identificador y la función **clearTimeout()** es posible impedir que se ejecute el código pendiente:

```
function muestraMensaje() {
  console.log("Han transcurrido 3 segundos desde que me programaron");
}
var id = setTimeout(muestraMensaje, 3000);
// Antes de que transcurran 3 segundos, se decide eliminar la ejecución pendiente
clearTimeout(id);
```

Además de programar la ejecución futura de una función, JavaScript también permite establecer la ejecución periódica y repetitiva de una función. El método necesario es **setInterval()** y su funcionamiento es idéntico al mostrado para **setTimeout()**:

```
function muestraMensaje() {
  console.log("Este mensaje se muestra cada segundo");
}
setInterval(muestraMensaje, 1000);
```

De forma análoga a ***clearTimeout()***, también existe un método que permite eliminar una repetición periódica y que en este caso se denomina ***clearInterval()***:

```
function muestraMensaje() {
  console.log("Este mensaje se muestra cada segundo");
}
var id = setInterval(muestraMensaje, 1000);

// Despues de ejecutarse un determinado número de veces, se elimina el intervalo
clearInterval(id);
```

4. El objeto document:

El objeto document es el único que pertenece tanto al DOM (como se vio en el capítulo anterior) como al BOM. Desde el punto de vista del BOM, el objeto document proporciona información sobre la propia página HTML.

Algunas de las propiedades más importantes definidas por el objeto document son:

Propiedad	Descripción
lastModified	La fecha de la última modificación de la página
referrer	La URL desde la que se accedió a la página (es decir, la página anterior en el array history)
title	El texto de la etiqueta <title>
URL	La URL de la página actual del navegador

Array	Descripción
anchors	Contiene todas las "anclas" de la página (los enlaces de tipo)
applets	Contiene todos los applets de la página
embeds	Contiene todos los objetos embebidos en la página mediante la etiqueta <embed>
forms	Contiene todos los formularios de la página
images	Contiene todas las imágenes de la página
links	Contiene todos los enlaces de la página (los elementos de tipo)

Los elementos de cada array del objeto document se pueden acceder mediante su índice numérico o mediante el nombre del elemento en la página HTML. Si se considera por ejemplo la siguiente página HTML:

```
<html>
  <head><title>Pagina de ejemplo</title></head>
  <body>
    <p>Primer parrafo de la pagina</p>
    <a href="otra_pagina.html">Un enlace</a>
    <img src="logo.jpg" name="logotipo"/>
    <form method="post" name="consultas">
      <input type="text" name="id" />
      <input type="submit" value="Enviar">
    </form>
  </body>
</html>
```

Para acceder a los elementos de la página se pueden emplear las funciones DOM o los objetos de BOM:

Párrafo: document.getElementsByTagName("p")

Enlace: document.links[0]

Imagen: document.images[0] o document.images["logotipo"]

Formulario: document.forms[0] o document.forms["consultas"]

Una vez obtenida la referencia al elemento, se puede acceder al valor de sus atributos HTML utilizando las propiedades de DOM. De esta forma, el método del formulario se obtiene mediante document.forms["consultas"].method y la ruta de la imagen es document.images[0].src.

5. El objeto location:

El objeto location es uno de los objetos más útiles del BOM. Debido a la falta de estandarización, location es una propiedad tanto del objeto window como del objeto document. El objeto location representa la URL de la página HTML que se muestra en la ventana del navegador y proporciona varias propiedades útiles para el manejo de la URL:

Propiedad	Descripción
hash	El contenido de la URL que se encuentra después del signo # (para los enlaces de las anclas) http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion hash = #seccion
host	El nombre del servidor http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion host = www.ejemplo.com
hostname	La mayoría de las veces coincide con host, aunque en ocasiones, se eliminan las www del principio http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion hostname = www.ejemplo.com
href	La URL completa de la página actual http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion URL = http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion
pathname	Todo el contenido que se encuentra después del host http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion pathname = /ruta1/ruta2/pagina.html
port	Si se especifica en la URL, el puerto accedido http://www.ejemplo.com:8080/ruta1/ruta2/pagina.html#seccion port = 8080 La mayoría de URL no proporcionan un puerto, por lo que su contenido es vacío http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion port = (vacío)
protocol	El protocolo empleado por la URL, es decir, todo lo que se encuentra antes de las dos barras inclinadas // http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion protocol = http:
search	Todo el contenido que se encuentra tras el símbolo ?, es decir, la consulta o *"query string"* http://www.ejemplo.com/pagina.php?variable1=valor1&variable2=valor2 search = ?variable1=valor1&variable2=valor2

De todas las propiedades, la más utilizada es **location.href**, que permite obtener o establecer la dirección de la página que se muestra en la ventana del navegador.

Además de las propiedades de la tabla anterior, el objeto location contiene numerosos métodos y funciones. Algunos de los métodos más útiles son los siguientes:

// Método assign()

```
location.assign("http://www.ejemplo.com");  // Equivalente a location.href = "http://www.ejemplo.com"
```

// Método replace()

```
location.replace("http://www.ejemplo.com");
// Similar a assign(), salvo que se borra la página actual del array history del navegador
```

// Método reload()

```
location.reload(true);
/* Recarga la página. Si el argumento es true, se carga la página desde el servidor.
Si es false, se carga desde la cache del navegador */
```

6. El objeto navigator:

El objeto **navigator** es uno de los primeros objetos que incluyó el BOM y permite obtener información sobre el propio navegador. En Internet Explorer, el objeto navigator también se puede acceder a través del objeto clientInformation.

El objeto **navigator** se emplea habitualmente para detectar el tipo y/o versión del navegador en las aplicaciones cuyo código difiere para cada navegador. Además, se emplea para detectar si el navegador tiene habilitadas las cookies y Java y también para comprobar los plugins disponibles en el navegador.

7. El objeto screen:

El objeto **screen** se utiliza para obtener información sobre la pantalla del usuario. Uno de los datos más importantes que proporciona el objeto **screen** es la resolución del monitor en el que se están visualizando las páginas.

Las siguientes propiedades están disponibles en el objeto screen:

Propiedad	Descripción
availHeight	Altura de pantalla disponible para las ventanas
availWidth	Anchura de pantalla disponible para las ventanas
colorDepth	Profundidad de color de la pantalla (32 bits normalmente)
height	Altura total de la pantalla en píxel
width	Anchura total de la pantalla en píxel

La altura/anchura de pantalla disponible para las ventanas es menor que la altura/anchura total de la pantalla, ya que se tiene en cuenta el tamaño de los elementos del sistema operativo como por ejemplo la barra de tareas y los bordes de las ventanas del navegador.

Además de la elaboración de estadísticas de los equipos de los usuarios, las propiedades del objeto screen se utilizan por ejemplo para determinar cómo y cuanto se puede redimensionar una ventana y para colocar una ventana centrada en la pantalla del usuario.

El siguiente ejemplo redimensiona una nueva ventana al tamaño máximo posible según la pantalla del usuario:

```
window.moveTo(0, 0);
window.resizeTo(screen.availWidth, screen.availHeight);
```

El siguiente ejemplo muestra algunas de las opciones de manejo de BOM

```
//Manejo de BOM (Browser Object Model)
function getBom(){
        console.log(window.innerHeight);
        console.log(window.innerWidth);
        console.log(window.location);
```

```
}

function redirec(url){
        window.location.href=url;
}
getBom();
redirec("http://www.google.es");
```

8. Eventos asociados a la carga de la pagina: DOMContentLoaded, load, beforeunload, unload

El ciclo de vida de una página HTML tiene tres eventos importantes:

- **DOMContentLoaded:** El navegador HTML está completamente cargado y el árbol DOM está construido, pero es posible que los recursos externos como <img> y hojas de estilo aún no se hayan cargado.
- **load:** No solo se cargó el HTML, sino también todos los recursos externos: imágenes, estilos, etc.
- **beforeunload/unload** : El usuario sale de la pagina.

Cada evento puede ser útil:

- **Evento DOMContentLoaded** – DOM está listo, por lo que el controlador puede buscar nodos DOM, inicializar la interfaz.
- **Evento load** – se cargan recursos externos, por lo que se aplican estilos, se conocen tamaños de imagen, etc.
- **Evento beforeunload** – el usuario se va: podemos comprobar si el usuario guardó los cambios y preguntarle si realmente quiere irse.
- **Evento unload** – el usuario casi se fue, pero aún podemos iniciar algunas operaciones, como enviar estadísticas.

Exploremos los detalles de estos eventos.

- **DOMContentLoaded:** El evento DOMContentLoaded ocurre en el objeto document. Debemos usar addEventListener para capturarlo:

```
document.addEventListener("DOMContentLoaded", ready);
// no "document.onDOMContentLoaded = …" No funciona
```

Por ejemplo:

```
*<script>
  function ready() {
    alert('DOM is ready');

    // la imagen aún no está cargada (a menos que se haya almacenado en caché), por lo que el tamaño es 0x0
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  }

  document.addEventListener("DOMContentLoaded", ready);
```

```
</script>

<img id="img" src="coche.jpg">
```

En el ejemplo, el controlador del evento DOMContentLoaded se ejecuta cuando el documento está cargado, por lo que puede ver todos los elementos, incluido el <img> que está después de él. Pero no espera a que se cargue la imagen. Entonces, alert muestra los tamaños en cero.

A primera vista, el evento **DOMContentLoaded** es muy simple. El árbol DOM está listo – aquí está el evento. Sin embargo, hay algunas peculiaridades.

- **DOMContentLoaded y scripts:** Cuando el navegador procesa un documento HTML y se encuentra con una etiqueta <script>, debe ejecutarla antes de continuar construyendo el DOM. Esa es una precaución, ya que los scripts pueden querer modificar el DOM, e incluso hacer document.write en él, por lo que **DOMContentLoaded** tiene que esperar.

Entonces DOMContentLoaded siempre ocurre después de tales scripts:

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    alert("DOM listo!");
  });
</script>

<script src="prueba.js"></script>

<script>
  alert("Libreria cargada, linea de script ejecutada");
</script>
```

En el ejemplo anterior, primero vemos “Biblioteca cargada …” y luego “¡DOM listo!” (se ejecutan todos los scripts).

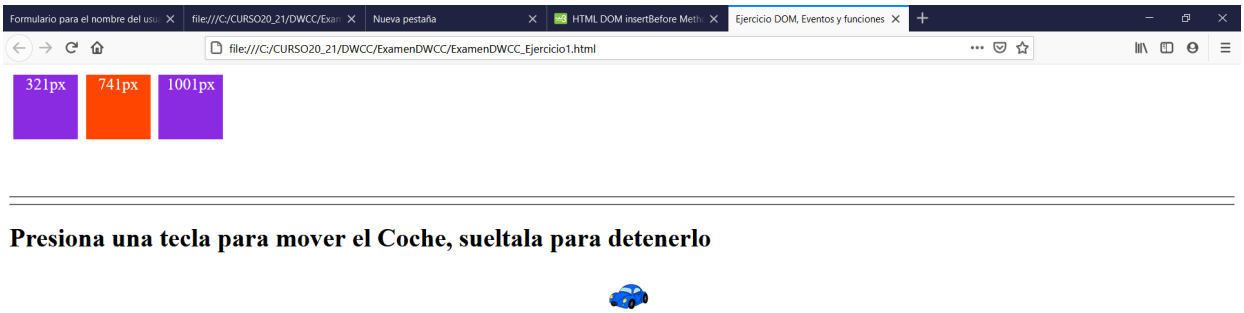
- **DOMContentLoaded y estilos:** Las hojas de estilo externas no afectan a DOM, por lo que DOMContentLoaded no las espera. Pero hay una trampa. Si tenemos un script después del estilo, entonces ese script debe esperar hasta que se cargue la hoja de estilo:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // el script no se ejecuta hasta que se cargue la hoja de estilo
  alert(getComputedStyle(document.body).marginTop);
</script>
```

La razón de esto es que el script puede querer obtener coordenadas y otras propiedades de elementos dependientes del estilo, como en el ejemplo anterior. Naturalmente, tiene que esperar a que se carguen los estilos. Como DOMContentLoaded espera los scripts, ahora también espera a los estilos que están antes que ellos.

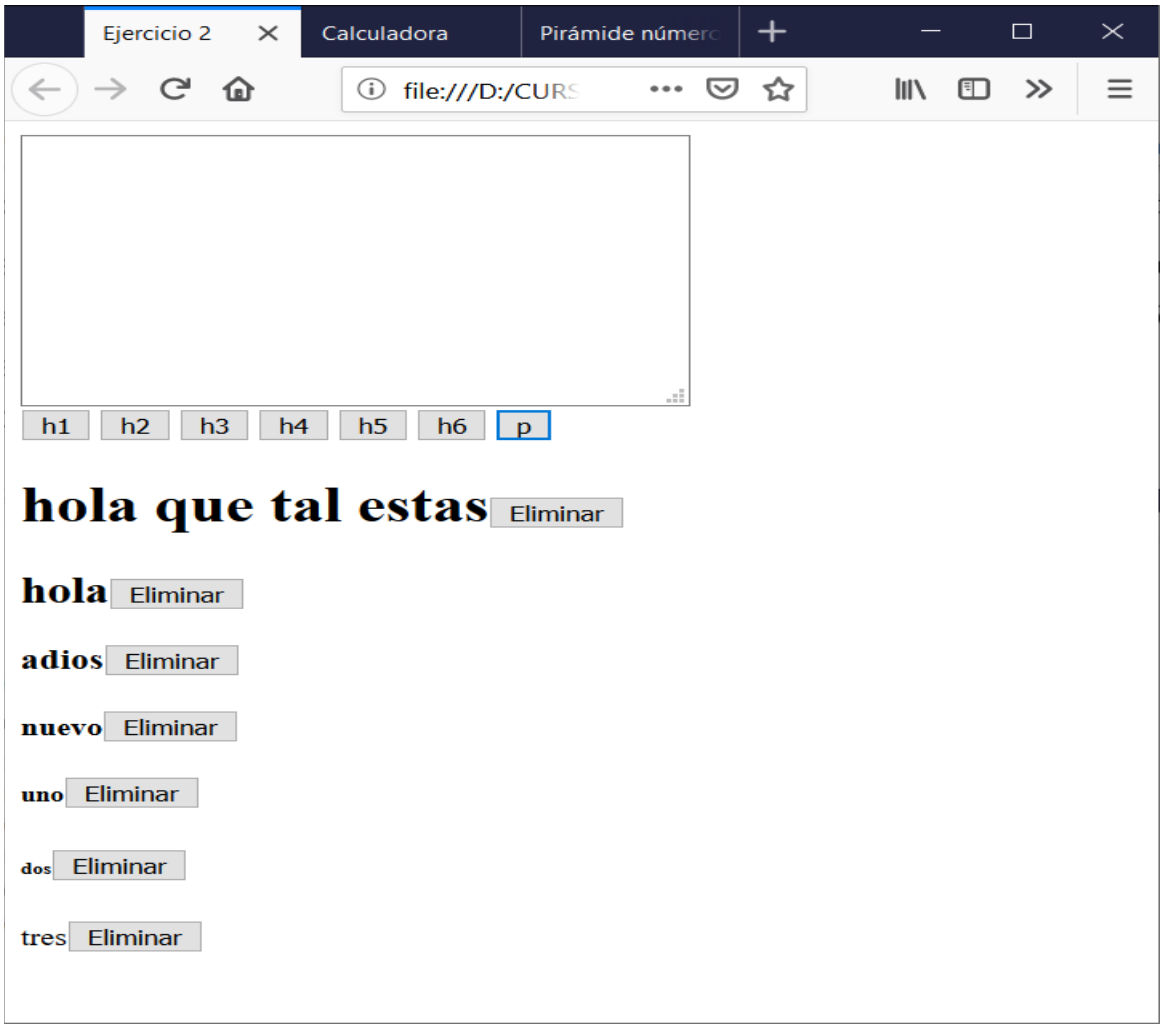
Secuencia y desarrollo:

1. **Ejercicio 1: A partir del fichero p3_Ejercicio1, escribe el código JavaScript que:**



1. **Funcionamiento:**
   1. Al pulsar una tecla (onkeydown), se moverá la imagen mientras la tecla esté pulsada, al soltarla (onkeyup) se parará.
   2. Al soltar la tecla se crea una capa div con la posición actual del coche.
   3. Al hacer clic sobre dicha capa, el coche se desplaza hacia esa posición y cambia el color.
2. **Ejecución**
   1. **Mover coche al pulsar la tecla**
   2. **Parar al final de la ventana**
   3. **Crear caja con formato clase**
   4. **Funcionar sólo con la tecla D**
   5. **Desplazar coche posición caja**
   6. **Cambiar color caja**
   7. **Volver al color inicial resto de cajas**

2. **Ejercicio 2:** ***A partir del fichero*** **p3_Ejercicio2,** ***escribe el código JavaScript que :*** Permita ir generando texto en la página: párrafos y encabezados. Al generar cada párrafo o encabezado, deberá facilitar un botón que, al pulsarlo, eliminará dicho párrafo o encabezado. ***Los elementos habrán de ser creados en el árbol DOM, no se permite la utilización de los métodos innerHTML, outerHTML ni insertAdjacentHTML.***

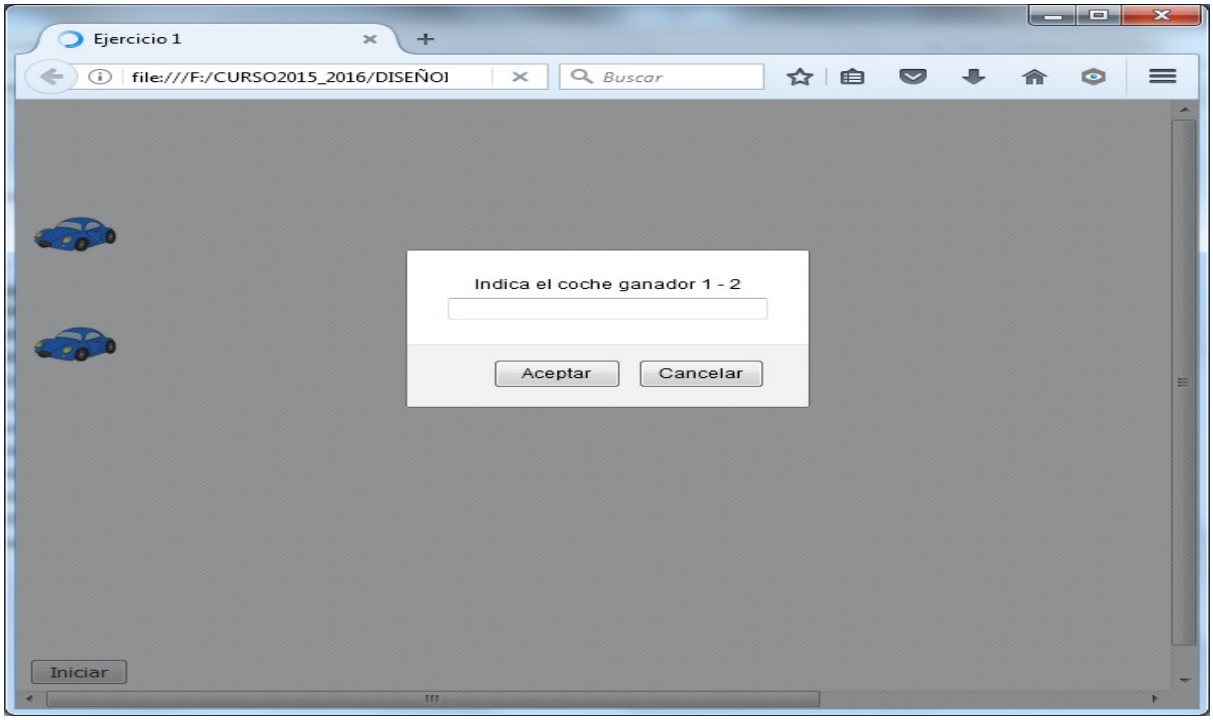


3. **Ejercicio 3 :** Implementa la siguiente página web en la que dos coches realizan una carrera. El usuario elige el coche que piensa que gana y lo guarda en una variable. A continuación se pulsa el botón de inicio y empiezan a moverse. La cantidad que hace que un coche se mueva más que otro se obtiene con la función **random**. Si el coche que llega antes es el que elige el usuario sale una alerta "Ha Ganado".

FUNCIÓN RANDOM: Math.Random → Devuelve un número pseudoaleatorio de punto flotante entre 0 (inclusive) y 1 (exclusivo)

OBTENER UN NUMERO ALEATORIO ENTRE DOS VALORES:

function getRandomArbitrary(min, max) {
return Math.random() * (max - min) + min;}



Requisitos:

1. Debes comprobar si los estilos han sido inicializados antes de comenzar la carrera.
2. Utilizar los métodos :
    1. ***Element.setAttribute***: Establece el valor de un atributo en el elemento indicado. Si el atributo ya existe, el valor es actualizado, en caso contrario, el nuevo atributo es añadido con el nombre y valor indicado.
    2. Para obtener el valor actual de un atributo, se utiliza ***getAttribute()***;
    3. **Para eliminar un atributo, se llama a *removeAttribute()*.**