UT05.1: Document Object Model (DOM)

Contenido

1. Introducción

El **DOM** (**Document Object Model** o en español **Modelo de Objetos del Documento**) es un API nativa de JavaScript que nos permite interactuar con documentos HTML o XML. Al ser nativa, este API está incluida en el intérprete de JavaScript del navegador, por lo que no necesitamos importar nada para poder utilizarla.

Con DOM representamos cada documento como un **árbol de nodos**, donde cada **nodo** representa una parte del documento, pudiendo ser un **elemento**, un **atributo**, una **cadena de texto**, incluso un **comentario**. La siguiente imagen representa el árbol de nodos de un documento, siendo la raíz un nodo que representa al propio **Documento**, y seguimos con el nodo que representa el elemento raíz de la página, el elemento *HTML*. Como vemos, también tenemos nodos atributos, nodos texto y el resto de los nodos elementos utilizados en la página.

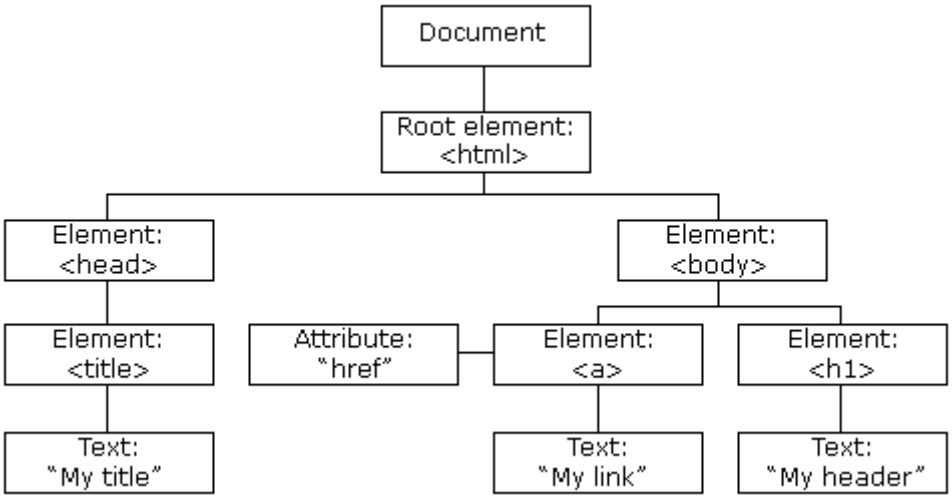

Imagen 1 Representación de árbol de nodos. Fuente w3school

1.1. ¿Qué podemos hacer con DOM?

DOM es un API que se puede dividir en función del tipo de documento que queremos manipular, un documento HTML o XML. En este apartado nos centraremos en el API que trabaja con documentos HTML. En cuanto a las versiones, la versión actual de DOM es la **3.0**.

Con este API podemos interactuar con cualquier nodo incluido en el documento. Algunas de las acciones que podemos realizar son:

- Cambiar los elementos HTML de una página.
- Cambiar los atributos de estos elementos.
- Cambiar los estilos CSS en la página.
- Añadir o eliminar tanto elementos como atributos.
- Reaccionar a los eventos ocurridos en la página.
- Crear nuevos elementos en la página.

Al trabajar con HTML, es más manejable utilizar interfaces que trabajan a nivel de elemento, en lugar de a nivel de nodo, aunque ambos están disponibles en el API. A nivel de elemento podemos interactuar con:

- **Atributos**: Corresponden con los atributos de la especificación de HTML.

- **Propiedades**: Nos da información sobre el elemento como por ejemplo su tamaño real, su contenido en formato de texto o en formato HTML, su estilo, sus relaciones con otros elementos como por ejemplo su padre.
- **Eventos**: cómo reaccionamos en base a los eventos ocurridos en un elemento concreto, asociando uno o más **manejadores de eventos**, funciones que son ejecutadas al dispararse el evento en el que están asignadas.

1.2. Tipos de datos

Todos los objetos de DOM heredan de `Node`, por lo que todos los tipos de datos tienen un interfaz común con la que podremos interactuar. En la siguiente tabla se muestran los principales tipos de datos que utiliza el API.

Tipo de dato	Descripción
`Document`	Este objeto representa el elemento raíz de un documento. A partir de este objeto podemos navegar por el resto de nodos del documento.
`Element`	Referencia a un elemento o a un nodo de tipo de elemento `element` devuelto por un método del API. Los objetos `element` utilizan tanto la interfaz `Element`, así como la interfaz de nodo "`Node`" de la cual hereda.
`NodeList`	Colección de nodos sobre la que podemos iterar igual que hacemos con un array. Para acceder a sus ítems podemos utilizar la siguiente notación: - list.item (1) - lista [1]
`Attribute`	Referencia a un objeto que expone una interfaz particular los atributos.
`NamedNodeMap`	Colección de ítems accesibles tanto por nombre como por índice, aunque en realidad no guardan ningún orden en particular en la lista.

En el siguiente link podemos encontrar las interfaces de cada tipo de objeto:

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Cuando trabajamos a nivel de elementos, en función del tipo seleccionado tendremos interfaces diferentes, es decir, un elemento `TABLE` tiene métodos, propiedades y eventos diferentes a `DIV`, aunque tengan una base común entre ellos. Para conocer las diferencias es necesario consultar la referencia de cada tipo de elemento para saber qué información nos aporta. En el siguiente enlace podemos ver el API del tipo `element`.

https://developer.mozilla.org/es/docs/Web/API/Element

2. Selección de elementos

El primer paso al trabajar con DOM es seleccionar los elementos con los que vamos a trabajar. En este punto vamos a revisar los métodos de selección que nos ofrece el API.

Nota: La selección de un elemento siempre debe hacerse asignando a una variable para su posterior reutilización a través de la variable. Buscar el mismo elemento una y otra vez en nuestro código, hará que la ejecución de la función sea más lenta, por esta razón, debemos **minimizar el número de selecciones** en nuestro código. La primera acción en cualquier función que utilice el API de DOM debe ser asignar a variables la selección de todos los elementos que vamos a manipular en la función.

2.1. Uso de la consola

Los objetos de DOM tienen un gran número de métodos, eventos y propiedades. Para revisar los valores temporales de un objeto DOM podemos utilizar el método `dir()` del objeto `console`, el cual mostrará esta información de un objeto especificado.

Vamos a mostrar el contenido del objeto `document` de la página.

```
console.dir(document)
```

El resultado es la lista jerarquizada de propiedades del objeto.

Imagen 2 Ejecución de console.dir(document)

La imagen de la derecha muestra la jerarquía de objetos heredada a través de la propiedad `__proto__` del objeto `document`. Podemos ver que un document es un documento de tipo HTML (`HTMLDocument`) el cuál hereda de `Document`, que a su vez es un objeto `Node`.

Podemos revisar esta relación utilizando la consola del navegador, y revisar la interfaz de cada uno de los métodos, propiedades y eventos.

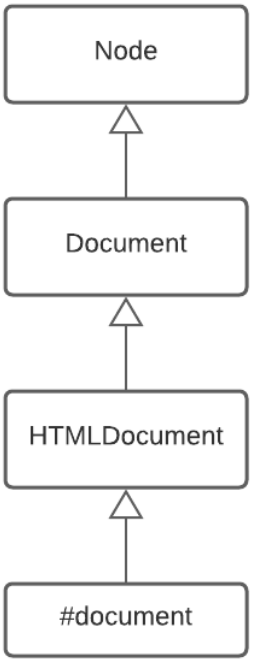

Imagen 3 Jerarquía de document

2.2. Método `getElementById()`

Este método del objeto `document` nos permite acceder a un elemento utilizando el identificador definido en la página HTML. Debemos recordar que el identificador debe ser único en el código HTML.

El siguiente código muestra la selección de elemento `H1` utilizando para ello su identificador, para luego mostrarlo en la consola.

```
let message = document.getElementById("message");
console.dir(message);
message.style.border = "2px solid red";
```

Una de las propiedades de cualquier elemento HTML es la propiedad `style`. Esta propiedad es un objeto que contiene todas las propiedades CSS que podemos asignar al elemento. Como hemos visto en el código, hemos modificado el borde del elemento seleccionado para que sea de color rojo. Las propiedades de `style` son nombradas con las de CSS pero la notación *camel-case* en lugar de utilizar guiones.

La siguiente imagen muestra la herencia de objetos que hemos seleccionado.

Imagen 4 Objeto HTMLHeadingElement

2.3. Método `querySelector()`

Además del identificador, podemos seleccionar elementos utilizando **selectores de CSS**. Este método selecciona el **primer elemento** que cumple el selector especificado. Veamos algunos ejemplos.

El siguiente código modifica el color de fondo de varios elementos en la página. La primera instrucción es similar a `getElementById()` al utilizan un selector identificador. El resto de las instrucciones permiten descendientes y utilizar las pseudoclase de CSS.

```
document.querySelector('#message').style.background = "blue";
document.querySelector('.navbar .account').style.background = "yellow";
document.querySelector('li:first-child').style.background = "red";
document.querySelector('li:last-child').style.background = "green";
document.querySelector('li:nth-child(3)').style.background = "purple";
```

Este método es parte de la interfaz de los objetos element. El siguiente código seleccionamos el elemento con identificador *footer* y a partir de él buscaremos los elementos que coincidan con el selector.

```
let message = document.getElementById("footer");
message.querySelector('li:first-child').style.background = "red";
message.querySelector('li:last-child').style.background = "green";
message.querySelector('li:nth-child(3)').style.background = "purple";
```

Podemos consultar la especificación del método en: https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelector

2.4. Elementos de selección múltiple

Lo más habitual es que queramos seleccionar un conjunto de elementos relacionados por algún criterio en común. Los métodos de selección múltiple nos permiten seleccionar una colección de elemento devueltos en un objeto `NodeList` o en un objeto `HTMLCollection`.

La colección `NodeList` funciona como un array en cuanto a la notación de acceso a través de corchete o la propiedad `length`, pero realmente no son un array ya que en su interfaz no están la mayoría de los métodos de los arrary.

Debemos tener en cuenta que `NodeList` es **estático**, es decir, una vez hecha la selección la colección no es alterada, aunque en otro punto de la aplicación se haya añadido o eliminado un elemento que pudiera formar parte de la colección, por lo que si es utilizada como variable global tiene que tomarse con precaución.

`NodeList` puede trabajar como si fuera un iterador con los métodos `forEach()`, `entries()`, `keys()` y `values()`, aunque versiones antiguas no soportan esta implementación.

https://developer.mozilla.org/en-US/docs/Web/API/NodeList

Por el contrario, `HTMLCollection` es una colección genérica de elementos como ocurría con `arguments`, sin posibilidad de utilizar como si fuera un iterador, pero con la ventaja de que se actualizará siempre que se añadan nuevos elementos que puedan formar parte de esta colección, por lo tanto, se trata de una colección **dinámica**.

https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection

Los métodos de selección múltiple pueden ser utilizados desde el objeto `document` o en un objeto `element`.

2.4.1. Método `querySelectorAll()`

Es la evolución de `querySelector()` pero seleccionando una colección de elementos. Vamos a enumerar cada uno de los enlaces de interés de la página. Primero seleccionamos el elemento `ul` contenedor de los enlaces, para luego obtener la colección de `NodeList` con los enlaces a partir del selector. La propiedad `textContent` nos devuelve el contenido de los nodos de texto del elemento. Para iterar utilizamos el método `forEach()`.

```
let ul = document.querySelector('#footer .footer-links ul');
let itemList = ul.querySelectorAll("li a");
itemList.forEach(function (item, index){
  item.textContent = index + 1 + item.textContent;
});
```

https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelectorAll

2.4.2. Método `getElementsByTagName()`

Aunque es menos habitual, podemos seleccionar elementos mediante el nombre de tag de los elementos que queremos seleccionar, teniendo como resultado una colección `HTMLCollection`. En el siguiente ejemplo seleccionamos todos los elementos `a` contenidos en el `ul` de los enlaces de servicios del pie de la página, a cada uno de ellos le cambiamos la decoración para que estén subrayados. Para iterar con los elementos de la colección utilizamos un bucle `for-of`.

```
let ul = document.querySelector('#footer div.row div:nth-child(3) ul');
let itemList = ul.getElementsByTagName("a");
for (let link of itemList) {
  link.style.textDecoration = "underline";
}
```

https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByTagName

2.4.3. Método `getElementsByClassName()`

Este método devuelve una colección `HTMLCollection` con todos los elementos con un atributo `class` determinado. En el ejemplo accedemos a los elementos con `class` *footer-links* incluidos en el `footer` de la página. Para iterar utilizamos un bucle `for` clásico.

```
let footer = document.getElementById("footer");
let divs = footer.getElementsByClassName("footer-links");
for (let i = 0; i < divs.length; i++) {
  divs[i].style.border = "2px solid red";
}
```

https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByClassName

2.4.4. Método `closest()`

Los ancestros de un elemento son los elementos que contienen a un elemento, es decir, podrán ser el padre, el padre del padre y así sucesivamente. Todos los ancestros forman la cadena de padres desde el elemento hasta la raíz.

El método `closest()` toma como argumento un selector CSS y busca el ancestro más cercano que coincide con el selector. El propio elemento también se incluye en la cadena búsqueda.

En otras palabras, el método el elemento y cada uno de los padres. Si coincide con el selector, entonces la búsqueda se detiene y devuelve dicho elemento.

El siguiente ejemplo busca todos los padres que sean elementos DIV a partir del botón de *ver artículo*.

```
const element = document.getElementById('button');
let container = button.closest('div');
while (container) {
  container.style.border = '2px solid green';
  container = container.parentElement.closest('div');
}
```

https://developer.mozilla.org/en-US/docs/Web/API/Element/closest

2.4.5. Método `matches()`

El método comprueba si un elemento coincide con el selector CSS. Se utiliza para filtrar los elementos que nos interese de una colección.

El siguiente ejemplo filtra para los enlaces del menú de cabecera a partir de todos los enlaces encontrados en la página.

```
const links = document.querySelectorAll('a');
for (const link of links) {
  if (link.matches('.dropdown-item')) console.log(link.textContent);
}
```

https://developer.mozilla.org/en-US/docs/Web/API/Element/matches

2.4.6. `NodeList` vs `HTMLCollection`

La diferencia entre ambos tipos de colecciones es que `NodeList` es estática, cualquier cambio en el DOM de la página no actualiza su contenido, mientras que `HTMLCollection` si que reflejará los cambios.

En el siguiente ejemplo obtenemos ambos tipos de colecciones para todos los enlaces de la página. Comparamos el tamaño de las colecciones y vemos que ambas son iguales. Añadimos un nuevo enlace a la página y volvemos a comparar las colecciones, vemos que `HTMLCollection` tiene un tamaño mayor.

```
const nodelist = document.querySelectorAll('a');
const htmlCollection = document.getElementsByTagName('a');
console.log(`Tamaño NodeList: ${nodelist.length}`);
console.log(`Tamaño HTMLCollection: ${htmlCollection.length}`);
document.body.insertAdjacentHTML('beforeend', '<a>NodeList vs
HTMLCollection</a>');
console.log(`Tamaño NodeList: ${nodelist.length}`);
console.log(`Tamaño HTMLCollection: ${htmlCollection.length}`);
```

3. Manipulación de elementos

DOM nos permite manipular los elementos para:

- Modificar el contenido interno.
- Trabajar con sus atributos.
- Modificar su estilo.

3.1. Modificar el contenido

A nivel de `Node` tenemos la propiedad `textContent` que representa el contenido en formato de texto de un nodo y sus descendientes. En este contenido se incluye el contenido de elementos como `style` y `script`.

https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent

La propiedad `innerText` pertenece a los objetos `HTMLElement`, muestra el contenido leíble por el usuario. Básicamente obtendríamos el mismo resultado de seleccionar el elemento y copiarlo en el portapapeles. Si tuviéramos contenido oculto a través de la propiedad `visibility` de CSS no estaría seleccionado.

https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/innerText

Por último, la propiedad `innerHTML` también pertenece a `HTMLElement` y muestra el contenido de un elemento y sus descendientes en formato HTML.

https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML

El siguiente ejemplo muestra el contenido del banner de la página utilizando las 3 propiedades anteriores.

```
let message = document.getElementsByClassName("banner")[0];
alert (message.textContent);
alert (message.innerText);
alert (message.innerHTML);
console.log(message.textContent);
console.log(message.innerText);
console.log(message.innerHTML);
```

Como hemos comentado, lo ideal es trabajar a nivel de elemento y no hacerlo con las propiedades de los objetos `Node`. En el siguiente código manipulamos las propiedades `innerText` e `innerHTML`. En el primer caso, el contenido es añadido en formato de texto, mientras que, en el segundo, el contenido es previamente parseado para formar parte del árbol DOM de la página.

```
let message = document.getElementById("message");
let message2 = document.getElementById("message2");
message.innerText = "<span>1 span</span> <span>2 span</span>";
message2.innerHTML = "<span>1 span</span> <span>2 span</span>";
```

3.2. Trabajar con atributos

Podemos interactuar con los atributos de un elemento. Tenemos básicamente cuatro métodos:

- `getAttibute()`: recupera el valor de un atributo. https://developer.mozilla.org/en-US/docs/Web/API/Element/getAttribute
- `setAttribute()`: fija el valor para un atributo. https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute
- `hasAttibute()`: devuelve un booleano indicando si el atributo está definido en el elemento. https://developer.mozilla.org/en-US/docs/Web/API/Element/hasAttribute
- `removeAttibute()`: elimina un atributo de un elemento. https://developer.mozilla.org/en-US/docs/Web/API/Element/removeAttribute

En el siguiente código iteramos sobre los enlaces de interés de *footer* de la página, modificando el atributo `href` para que contenga el URL completa de la página, y los atributos `alt` y `title`.

```
let ul = document.querySelector('#footer .footer-links ul');
let itemList = ul.querySelectorAll("li a");
itemList.forEach(function (item, index){
  if (item.getAttribute('href').indexOf('#') === -1)
    item.setAttribute('href', document.documentURI + '#');
  else
  item.setAttribute('href', document.documentURI);
  if (!item.hasAttribute('title')){
    item.setAttribute('title', 'Título ' + index);
    item.setAttribute('alt',item.innerText);
  }
});
```

Además de los métodos mencionados tenemos las siguientes propiedades y métodos:

- `attributes`: Proporciona un objeto `NamedNodeMap` que nos permite recorrer los atributos utilizando su nombre.
- `hasAttributes()`: El método indica si el elemento tiene definidos atributos.

Podemos acceder a los atributos de un elemento y modificarlos a través del nombre de atributo, pero siempre teniendo en cuenta que el tipo de dato de los valores es un string. En este ejemplo vemos como generar un identificador para el elemento `ul` de los enlaces de interés, y lo mostramos en una función `alert()`.

```
let ul = document.querySelector('#footer .footer-links ul');
ul.id = "enlaces-interes";
alert(ul.id);
```

3.3. Modificar el estilo

Además de trabajar con la propiedad `style` de los elementos como hemos hecho en algunos de los ejemplos, podemos trabajar en función de las clases CSS que estamos aplicando al

elemento. La propiedad de solo lectura `classList` es un objeto que recoge la lista de clases aplicadas al elemento. https://developer.mozilla.org/en-US/docs/Web/API/Element/classList

Los métodos que ofrece el objeto `classList` son:

- `add()`: Añade una nueva clase al elemento.
- `remove()`: Borra una clase del elemento.
- `replace()`: Reemplaza una clase por otra.
- `toggle()`: Sirve de interruptor, si la clase que añadimos existe la elimina, y si no existe la añade.

Vamos a ver una serie de ejemplo para jugar con estos métodos. En primer lugar, vamos a seleccionar los elementos `i` utilizados para generar los iconos de servicios del *footer* de la página.

En el primer ejemplo solamente vamos a seleccionar los servicios en posición impar, para cambiar el icono de flecha derecha por la flecha izquierda. Además, vamos a añadir una fuente de color utilizando la clase de Bootstrap *text-primary*.

```
let ul = document.querySelector('#footer div.row div:nth-child(3) ul');
let itemList = ul.querySelectorAll("li:nth-child(odd) a i");
for (let i of itemList) {
  i.classList.replace('fa-angle-right', 'fa-angle-left');
  i.classList.add('text-primary');
}
```

En el segundo ejemplo seleccionamos todos los elementos, para intercambiar la clase *text-primary*.

```
let ul = document.querySelector('#footer div.row div:nth-child(3) ul');
let itemList = ul.querySelectorAll("li a i");
for (let i of itemList) {
  i.classList.toggle('text-primary');
}
```

En el último ejemplo, seleccionamos todos los elementos, y vamos comprobando si tienen una clase *bi-chevron-right*, si es así la eliminamos y añadimos *bi-chevron-left*.

```
let ul = document.querySelector('#footer div.row div:nth-child(3) ul');
let itemList = ul.querySelectorAll("li a i");
for (const i of itemList) {
  if (i.classList.contains('bi-chevron-right')) {
    i.classList.remove('bi-chevron-right');
    i.classList.add('bi-chevron-left');
  }
}
```

3.4. Acceso a atributos personalizados

Los atributos personalizados nos permiten crear nuestros propios atributos dentro del código HTML de nuestra página. En el siguiente ejemplo hemos creado dos atributos personalizados para añadir información del tipo de categorías y un código. Como vemos, los atributos personalizados deben comenzar con el prefijo "`data-`".

```
<div data-type="list" data-code="14" id="categories"
class="container cat-list">
```

Cada objeto de tipo elemento dispone de una propiedad denominada `dataset` que contiene un objeto literal con todas las propiedades que hayamos definido como atributos personalizados, en este caso *type* y *code*. En el siguiente código vemos cómo podemos acceder a los atributos personalizados, y cómo podemos modificar su contenido.

```
let categories = document.getElementById('categories');
console.dir(categories);
console.log('Type: ' + categories.dataset.type);
console.log('Code: ' + categories.dataset.code);
categories.dataset.code = 41;
console.log('New code: ' + categories.dataset.code);
```

El contenido de las propiedades de `dataset` es siempre un *string*, independientemente como hayamos almacenado el dato. Lógicamente, los atributos personalizados solo admiten datos primitivos.

4. Relación entre elementos

La estructura en árbol de la página implica que podemos tener una relación de "parentesco" entre los elementos, lo que nos permite navegar entre los elementos de la página utilizando estas relaciones. Aunque el API permite trabajar tanto a nivel de nodo como a nivel de elemento, vamos a centrarnos en la segunda opción al estar trabajando con documentos HTML.

- `parentElement`: Referencia el elemento padre de un elemento.
- `children`: Devuelve una colección `HTMLCollection` con los elementos hijos de un elemento.
- `firstElementChild`: De los elementos hijos, referencia el primer elemento.
- `lastElementChild`: De la misma forma que el anterior, devuelve el último elemento hijo.
- `nextElementSibling`: Devuelve el siguiente hermano en la estructura de un elemento dado.
- `previousElementSibling`: Devolvería el elemento previo.

En este primer ejemplo vamos a selecciona el elemento `ul` correspondientes a los enlaces de interés del *footer*. Una vez seleccionado vamos a asignar colores a los bordes de cada uno de los elementos relacionados con la lista.

```
let ul = document.querySelector('#footer div.row div:nth-child(2) ul');
ul.parentElement.style.border = "2px solid red";
for (let item of ul.children){
  item.style.border = "2px solid green";
}
ul.firstElementChild.style.border = "2px solid blue";
ul.lastElementChild.style.border = "2px solid blue";
```

En el segundo ejemplo trabajaremos con los hermanos previos y posteriores de un elemento. En los enlaces de servicios, seleccionamos el de posición 3. Iteramos sobre los hermanos para cambiar el borde de cada uno de ellos.

```
let li = document.querySelector('#footer div.row div:nth-
child(3) ul li:nth-child(3)');
let previous = li.previousElementSibling;
while (previous){
  previous.style.border = "2px solid red";
  previous = previous.previousElementSibling;
}
let next = li.nextElementSibling;
while (next){
  next.style.border = "2px solid blue";
  next = next.nextElementSibling;
}
```

Por último, podemos combinar estas propiedades para navegar entre los elementos. Vamos a recorrer los diferentes elementos padres a partir de la lista de enlaces de interés.

```
let ul = document.querySelector('#footer div.row div:nth-child(2) ul');
ul.parentElement.style.border = "2px solid red";
ul.parentElement.parentElement.style.border = "2px solid red";
ul.parentElement.parentElement.parentElement.style.border = "2px solid re
d";
ul.parentElement.parentElement.parentElement.parentElement.style.border =
 "2px solid red";
ul.parentElement.parentElement.parentElement.parentElement.parentElement.
style.border = "2px solid red";
```

5. Crear elementos en la página

En este punto vamos a trabajar cómo crear nuevos elementos y añadirlos en la página, y cómo podemos borrarlos.

5.1. Crear y añadir elementos

El objeto `document` tiene dos métodos que nos permiten crear nuevo contenido. El método `createElement()` permite crear un nuevo objeto `Element`, mientras que `createTextNode()` que un `Node` de texto.

- https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement
- https://developer.mozilla.org/en-US/docs/Web/API/Document/createTextNode

En el siguiente ejemplo, vamos a crear un `li` para añadirlo al listado de servicios. La estructura HTML es la siguiente:

```
<li><a href="#"><i class="fas fa-angle-right"></i> Servicio 1</a></li>
```

Tenemos que ir creando elemento a elemento para añadir a su padre. El método `append()` añadirá en la zona de árbol que nos interese.

https://developer.mozilla.org/en-US/docs/Web/API/Element/append

https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild

En el siguiente código seleccionamos el elemento `ul`, creamos un elemento `li` y lo añadimos al elemento `ul`.

```
let ul = document.querySelector('#footer div.row div:nth-child(3) ul');
let li = document.createElement("li");
ul.append(li);
```

Ahora creamos el enlace con el elemento a. Una vez creado podemos añadir los atributos que necesitemos como el atributo `href`. Lo añadimos al `li` creado en el paso anterior.

```
let anchor = document.createElement("a");
anchor.href = "#";
li.append(anchor);
```

Para el elemento `i` le añadimos las clases para generar el icono.

```
let i = document.createElement("i");
i.classList.add('bi');
i.classList.add('bi-chevron-right');
anchor.append(i);
```

Por último, creamos el nodo de texto para finalizar la estructura del elemento.

```
let text = document.createTextNode(" Servicio " + ul.children.length);
anchor.append(text);
```

5.2. Insertar

Además de `append()`, tenemos otras opciones para insertar una lista arbitraria de nodos o de string en un elemento.

- `append(...nodos o strings)`: agrega nodos o strings al final del elemento.
- `prepend(...nodos o strings)`: insert nodos o strings al principio del elemento.
- `before(...nodos o strings)`: inserta nodos o strings antes del elemento.
- `after(...nodos o strings)`: inserta nodos o strings después del elemento.

Vamos a crear una serie de nodos y los vamos a insertar alrededor de las categorías.

```
const dom1 = document.createElement('div');
dom1.append(document.createTextNode('DOM Append'));
dom1.style.border = '1px solid blue';
const dom2 = document.createElement('div');
dom2.append(document.createTextNode('DOM Prepend'));
dom2.style.border = '1px solid blue';
const dom3 = document.createElement('div');
dom3.append(document.createTextNode('DOM After'));
dom3.style.border = '1px solid blue';
const dom4 = document.createElement('div');
dom4.append(document.createTextNode('DOM Before'));
dom4.style.border = '1px solid blue';

const categories = document.getElementById('categories');
categories.style.border = '5px solid green';
categories.append(dom1, 'Texto en append');
categories.prepend(dom2, 'Texto en prepend');
categories.after(dom3, 'Texto en after');
categories.before(dom4, 'Texto en before');
```

También poder reemplazar un elemento con uno o con una lista de elementos o texto mediante el método `replaceWith(...nodos o strings)`.

Sustituimos el banner de la página por un texto.

```
const banner = document.querySelector('.article-banner');
banner.replaceWith('Aquí estaba el banner');
```

Por último, podemos insertar texto para que sea parseado y se añada el contenido como nodos dentro del árbol DOM. El método `insertAdjacentHTML()` toma como primer argumento la ubicación de donde insertar el texto parseado pasado como segundo argumento. Las ubicaciones pueden ser:

- `"beforebegin"`: inserta `html` inmediatamente antes del elemento.
- `"afterbegin"` – inserta `html` , al principio del elemento.
- `"beforeend"` – inserta `html`, al final del elemento.
- `"afterend"` – inserta `html` inmediatamente después del elemento.

El siguiente ejemplo muestra como insertar html en las cuatro ubicaciones alrededor de las categorías.

```
const div1 = '<div style="border: 1px solid red">beforebegin</div>';
const div2 = '<div style="border: 1px solid red">afterbegin</div>';
const div3 = '<div style="border: 1px solid red">beforeend</div>';
const div4 = '<div style="border: 1px solid red">afterend</div>';
categories.insertAdjacentHTML('beforebegin', div1);
categories.insertAdjacentHTML('afterbegin', div2);
categories.insertAdjacentHTML('beforeend', div3);
categories.insertAdjacentHTML('afterend', div4);
```

5.3. Clonar elementos

Como hemos visto, crear la estructura de elementos puede ser bastante tedioso. El método `cloneNode()` nos permite clonar un nodo exactamente igual a la referencia tomada. Opcionalmente el método recibe el argumento para indicar si queremos copiar la estructura completa, o solamente el nodo concreto.

https://developer.mozilla.org/en-US/docs/Web/API/Node/cloneNode

Debemos tener en cuenta que el nodo clonado no forma parte del árbol. Tendremos añadir el nodo posteriormente en el lugar que nos interese.

Vamos a repetir el ejercicio anterior pero utilizando `cloneNode()`. Primero seleccionamos el elemento que queremos clonar, y asignamos a una variable el elemento incluido la estructura de nodos que contiene. Sabemos que los textos son tipo `Node`, por lo que para seleccionar el texto utilizamos la propiedad `childNodes` que es una colección con los nodos hijos de un elemento. En nuestra estructura tenemos dos nodos, el elemento `i` y el texto. Utilizamos la propiedad `textContent` para cambiar el contenido del texto. Por último, añadimos el elemento en la posición deseada.

```
const li = document.querySelector('#footer div.row div:nth-child(2) ul
li:nth-child(2)');
const new_li = li.cloneNode(true);
new_li.firstElementChild.childNodes[1].textContent = ' Nuevo enlace';
li.parentElement.append(new_li);
```

En este ejemplo particular hemos hecho una excepción al trabajar a nivel de nodo, pero solamente porque tenemos como hijos del elemento `a` un elemento y un nodo de texto. Si el nodo de texto hubiese sido único como hijo del elemento `a`, podríamos haber utilizado la propiedad `innerText`, evitándonos trabajar con nodos.

5.4. Eliminar elementos

Por último, el método más fácil de utilizar es `remove()` que nos permite eliminar un elemento del árbol DOM, y por tanto de la página. Lo invocamos sobre el elemento a eliminar.

https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/remove

El siguiente código eliminar un elemento de la lista de enlaces de interés.

```
let li = document.querySelector('#footer div.row div:nth-
child(2) ul li:nth-child(2)');
const ul = li.parentElement;
li.remove();
```

Tenemos que tener en cuenta que lo que hemos eliminado es el elemento del árbol DOM, el objeto sigue existiendo y lo podríamos utilizar para añadirlo en otra ubicación.

```
ul.append(li);
```

5.5. DocumentFragment

Se trata de un nodo DOM especial que podemos utilizar de contenedor para mover otros nodos a diferentes puntos de nuestro árbol DOM. Al insertar un `DocumentFragment` se insertará el contenido no el objeto en si.

En el siguiente ejemplo hemos creado una función que recoge los enlaces de interés y los guarda en un `DocumentFragment`, para posteriormente insertarlos junto con los enlaces de servicios.

```
function getDocumentFragment(links) {
  const df = new DocumentFragment();
  for (const child of links.children) {
    df.append(child.cloneNode(true));
  }
  return df;
}

const linksOfInteres = document.querySelector('#footer div.row div:nth-
child(2) ul');
const services = document.querySelector('#footer div.row div:nth-child(3)
ul');
const dfLinks = getDocumentFragment(linksOfInteres);
services.append(dfLinks);
```

Esta acción se puede realizar a partir de un array y utilizando el operador de propagación en el momento de insertar. En el siguiente ejemplo clonamos los links de servicios en un array, para añadirlos en el menú de enlaces de interés.

```
function getArrayFromNode(links) {
  const array = [];
  for (const child of links.children) {
    array.push(child.cloneNode(true));
  }
  return array;
}
const arrayServices = getArrayFromNode(services);
linksOfInteres.append(...arrayServices);
```

6. Manejadores de eventos

Los manejadores de eventos, en inglés *event listener*, son las funciones que son ejecutadas en respuesta de un evento determinado. En este punto vamos a ver la forma de asociar manejadores de eventos a un elemento.

Como ejemplo de evento vamos a utilizar el más típico, el evento *click*, y lo vamos a asociar al enlace con forma de botón de la página etiquetado con el identificador *button*.

Para que un objeto pueda recibir eventos y le podamos asociar un manejador de eventos debe implementar el interfaz DOM `EventTarget`. Los objetos más comunes que implementan este interfaz son `Element`, `Document` y `Window`.

https://developer.mozilla.org/en-US/docs/Web/API/EventTarget

6.1. Manejadores como atributos

Si trabajamos con los eventos a nivel de atributos, tenemos que utilizar una preposición antes del nombre del evento. La preposición más habitual es `on`, aunque podemos encontrar otras.

El siguiente código genera una manejador de eventos para el botón de nuestra página, el cual ejecuta una función `alert()`.

```
let button = document.getElementById("button");
button.onclick = function(){
  alert("Ejecución del manejador");
}
```

Asociar manejadores a atributos es muy sencillo, pero tiene la limitación de que solamente podemos asociar un único manejador por evento.

Nota: Al igual que intentamos no mezclar CSS con el contenido HTML de la página, lo que facilita el mantenimiento, lo ideal es no definir los manejadores de eventos directamente en el HTML de la página. Si nuestra aplicación genera el HTML dinámicamente desde el servidor, no como hacemos en nuestro ejemplo, estaremos reduciendo la cantidad de código que tiene que generar, nuestras páginas pesarán menos, y el código JavaScript suele cachearse por el navegador.

6.2. Referencia `this` en manejador

Podemos utilizar `this` en una manejador de eventos para hacer referencia al elemento sobre el que hemos declarado el manejador. En este ejemplo vemos como utilizando la referencia podemos ver el identificador del elemento sobre el que se ha disparado el evento.

```
let button = document.getElementById("button");
button.onclick = function(){
  alert("Identificador: " + this.id);
}
```

6.3. Método `addEventListener()`

Podemos utilizar este método para asignar un manejador de eventos a un elemento. Como argumentos utiliza el tipo de evento sin preposición, y la función como manejador que queremos ejecutar cuando se dispare el evento. Podemos invocar el método varias veces sobre el mismo elemento para asignar varios manejadores.

https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

En el siguiente código añadimos al botón un manejador para transformar el texto en minúsculas o restaurarlo a su formato normal. Utilizamos para ello la clase *text-lowercase* de Bootstrap.

```
let button = document.getElementById("button");
button.addEventListener("click", function(){
  console.log("Evento minúsculas.")
  this.classList.toggle("text-lowercase");
})
```

Vamos a añadir un segundo manejador para que el texto esté en negrita.

```
button.addEventListener("click", function(){
  console.log("Evento negrita.")
  this.classList.toggle("font-weight-bold");
})
```

Podemos revisar como el orden de ejecución de los eventos sigue el orden de su declaración en la consola.

Algunos eventos solo pueden ser capturar mediante este sistema, por ejemplo, el evento `DOMContentLoaded`.

6.3.1. Opciones del manejador

Podemos configurar las opciones de ejecución del manejador de eventos. A partir del tercer argumento, podemos pasar un objeto literal con el nombre de la opción y si está activada o no mediante un booleano. Pasamos un objeto por opción. Tenemos las siguientes:

- `once`: si es `true`, el manejador se eliminará automáticamente tras la primera ejecución.
- `capture`: configura cómo se captura el evento al propagarse en la página. Lo veremos más adelante en detalle.
- `passive`: si se activa, el evento no cancelará su comportamiento por defecto.
- `signal`: el manejados se eliminará se aborte la ejecución mediante un método `abort()`. Lo estudiaremos con la programación asíncrona.

El siguiente ejemplo configura un manejador de eventos para la imagen del perfil del usuario para que muestre su nombre al ser cliqueado. La acción solo se puede ejecutar una única vez.

```
const profile = document.querySelector('.image');
profile.addEventListener(
  'click',
  () => {
    alert('John Doe');
  },
  {
    once: true,
  },
);
```

6.4. Ejercicio 1: Nuevo enlace de interés

Añade un elemento `input` y un `button` en los enlaces de interés. El botón debe recoger el texto del input, validar que tenga al menos cinco caracteres y añadirlo como enlace al listado.

Solución

Creamos el formulario. Al input le añadimos los atributos `type` e id. Ambos elementos los añadimos al padre de la lista.

```
let ul = document.querySelector('#footer div.row div:nth-child(2) ul');
let input = document.createElement("input");
input.type = "text";
input.id = "new-link";
input.placeholder = "Nuevo enlace";
ul.parentElement.append(input);
let button = document.createElement("button");
button.innerText = "Añadir";
ul.parentElement.append(button);
```

Al botón le añadimos un manejador de eventos para añadir el enlace. Chequeamos la longitud del texto en el `input`, el cual es accesible a través del atributo `value`, y creamos el enlace como en los puntos anteriores.

```
button.addEventListener("click", function(){
  let input = document.getElementById("new-link");
  if (input.value.length > 4){
    let ul = document.querySelector('#footer div.row div:nth-
child(2) ul');
    let li = document.createElement("li");
    ul.append(li);
    let anchor = document.createElement("a");
    anchor.href = "#";
    li.append(anchor);
    let i = document.createElement("i");
    i.classList.add("bi");
    i.classList.add("bi-chevron-right");
    anchor.append(i);
    let text = document.createTextNode(" " + input.value);
    anchor.append(text);
  } else {
    alert ("La longitud debe ser de al menos 5 caracteres.");
  }
  input.value = "";
})
```

6.5. Ejercicio 2: Imágenes a tamaño completo

Añade un manejador de eventos para cada una de la sección de categorías, para que al cliquear en ellas se abra la imagen a tamaño de pantalla completo. Podemos utilizar el método `requestFullscreen()` para hacerlo.

Solución

Seleccionamos los elementos con los que vamos a trabajar.

```
let categories = document.getElementById("categories");
let images = categories.getElementsByTagName("img");
```

Creamos una función como manejador. En este caso, al querer añadir la misma función a varios elementos es mejor no utilizar una función anónima, ya que de esa forma se estaría creando funciones diferentes por cada imagen. Con una función nominal podemos asociar el mismo manejador a todas las imágenes.

```
function imageFullScreen(){
  if (this.requestFullscreen) {
    this.requestFullscreen();
  }
}
```

Por último, iteramos para añadir los manejadores.

```
for (let image of images){
  image.addEventListener("click", imageFullScreen);
}
```

6.6. Método `removeEventListener()`

Si queremos eliminar un manejador de eventos de un elemento debemos pasar como argumentos el tipo de evento y el manejador al método `removeEventListener()`.

https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/removeEventListener

En este código eliminamos el manejador de eventos a la primera imagen de las categorías del ejercicio anterior.

```
let categories = document.getElementById("categories");
let images = categories.getElementsByTagName("img");
images[0].removeEventListener("click", imageFullScreen);
```

7. Objeto `Event`

Tenemos una larga lista de eventos que podemos capturar en nuestra página. Las categorías más comunes de eventos son:

- Carga de recursos
- Red
- Foco
- WebSocket
- Animaciones CSS
- Transiciones CSS
- Formulario
- Impresión
- Composición de texto
- Vista
- Cortapapeles
- Teclado
- Ratón
- Drag & Drop
- Media
- Progreso
- Desplazamiento

Es conveniente que revisemos todos los eventos disponibles. En el siguiente enlace podemos consular los eventos de cada una de las categorías.

https://developer.mozilla.org/en-US/docs/Web/Events

Cada una de las categorías de eventos implementa el interfaz `Event`, por lo que tenemos una interfaz común en cuanto propiedades y métodos. El segundo interfaz que implementan los eventos son `UIEvent` para representan eventos de interfaz de usuario simples.

- https://developer.mozilla.org/en-US/docs/Web/API/Event
- https://developer.mozilla.org/en-US/docs/Web/API/UIEvent

7.1. Recuperar el evento

Al dispararse un evento, el navegador crea un objeto `Event` con toda su información. El manejador de eventos puede declarar un parámetro para recoger este objeto y acceder a sus propiedades. Es el navegador el que invoca en manejador y el que pasa el objeto con el evento.

En el siguiente código recogemos el argumento con el evento en el parámetro *event*, y accedemos a algunos de sus propiedades como `target`, que hace referencia al objeto sobre el que se ha lanzado el evento, y `type`, con el tipo específico de evento.

```
let button = document.getElementById("button");
button.onclick = function(event){
  alert("Identificador: " + event.target.id + " Tipo: " + event.type);
}
```

7.2. Relación entre `this` y `target`

Como hemos comentado, `this` hace referencia al elemento donde hemos definido el manejador de eventos, es decir, es el punto donde se captura este evento. Por el contrario, `target` es el evento que dispara el evento. Vamos a declarar un manejador de eventos en una estructura compleja, por ejemplo, a la capa *categories* de nuestra página, ya que tiene una profundidad de varios niveles.

Si cliqueamos en un elemento interior, también estaremos cliqueando en el elemento que lo contiene debido a la estructura de árbol. La propiedad `target` es la referencia del elemento interior que hemos cliqueado. Lógicamente, se puede dar el caso que ambas referencias son el mismo objeto.

En ocasiones podemos perder la referencia `this`, ya sea por utilizar una función flecha como manejadora de eventos, la cual no tiene vinculada esta referencia, o que tengamos vinculado `this` a otro objeto. El evento posee la propiedad `currentTarget` que hace referencia a el objeto donde el evento es capturado y coincide con `this`.

En el código hacemos la comparación entre ambas referencias para comprobar que se tratan de objetos diferentes.

```
const categories = document.getElementById('categories');
categories.addEventListener('click', function (event) {
  alert(`this: ${this.id} target: ${event.target.nodeName}, ${this ===
event.currentTarget}`);
});
```

7.3. Objetos handlers

Además de funciones, podemos utilizar como manejadores de eventos objetos. Estos objetos deben tener implementado el método `handleEvent(event)` que recibe como argumento el evento que estamos capturando. Será el navegador el que invoque dicho evento del objeto. La ventaja es clara, podemos utilizar el estado del objeto para utilizarlo en respuesta de un evento.

En el siguiente ejemplo vamos a una clase con la funcionalidad de un contador, el estado del objeto lo utilizaremos en el método `handleEvent(event)` para modificar el contenido del elemento que captura el evento, en este caso el botón de ver artículo. A continuación, vemos la clase del manejador.

```
class CounterHandler {
  constructor(init) {
    this.value = init;
  }

  increment() {
    this.value += 1;
  }

  handleEvent(event) {
    this.increment();
    event.currentTarget.innerHTML = this.value;
  }
}
```

Para poder asignarlo como manejador, primero lo instanciamos, lo hacemos con un valor inicial, y lo asignamos como manejador al botón que capturará el evento.

```
const button = document.getElementById('button');
const handler = new CounterHandler (3);
button.addEventListener('click', handler);
```

El objeto handler puede ser utilizado para manejar diferentes eventos. Vamos a reescribir el código anterior para que al cliquear en la imagen de perfil o el botón se incremente el contador, el resultado lo mostramos en el mensaje de descuento. La clase toma un argumento con el objetivo a la hora de mostrar el resultado en el argumento. El método lo tiene `handleEvent(event)` en cuenta.

```
class CounterHandler {
  constructor(init, target) {
    this.value = init;
    this.target = target;
  }

  increment() {
    this.value += 1;
  }

  handleEvent(event) {
    this.increment();
    (this.target)
      ? this.target.innerHTML = this.value
      : event.currentTarget.innerHTML = this.value;
  }
}
```

Asignamos el objeto manejador instanciado como manejador de dos eventos en la imagen de perfil y en el botón. Ambos comparten el estado del objeto.

```
const button = document.getElementById('button');
const message = document.getElementById('message');
const profile = document.querySelector('.image');
const handler = new CounterHandler(3, message);
button.addEventListener('click', handler);
profile.addEventListener('click', handler);
```

7.4. Eventos de ratón

La información más importante que aporta este tipo de eventos es la posición exacta donde se ha disparado el evento. Las propiedades para dar información están en base a coordenadas X e Y utilizando varios puntos de referencia.

- `offsetX` y `offsetY`: Fija el punto de referencia en la esquina superior izquierda del elemento **donde se disparó el evento**, es decir, el objeto referenciado por `target`. No incluimos el borde del elemento.
- `clientX` y `clientY`: El punto de referencia es la esquina superior izquierda del área del usuario.
- `pageX` y `pageY`: El documento es el punto de referencia, por lo que tenemos también en cuenta el posible scroll que hayamos generado en la página.
- `screenX` y `screenY`: El último punto de referencia es la esquina superior izquierda de la pantalla.

https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent

7.4.1. Evento `click`

En el siguiente código añadimos un elemento `DIV` al *banner* de la página, y le asociamos un manejador de eventos para el evento `click` que nos indique las coordenadas para los cuatro tipos de puntos de referencia.

```
let banner = document.querySelector(".banner");
let div = document.createElement("div");
div.id = "references";
div.style.border = '5px solid red';
banner.after(div);
banner.addEventListener("click", function(event){
  let str = "";
  str = "offsetX: " + event.offsetX + " offsetY: " + event.offsetY + "<br
>";
  str += "clientX: " + event.clientX + " clientY: " + event.clientY + "<b
r>";
  str += "pageX: " + event.pageX + " pageY: " + event.pageY + "<br>";
  str += "screenX: " + event.screenX + " screenY: " + event.screenY + "<b
r>";
  div.innerHTML = str;
});
```

Si queremos forzar el evento `click` en un elemento concreto, tenemos el método `click()` sobre el elemento que lo fuerza. El siguiente código fuerza el click del banner al cliquear en el botón de ver artículo.

```
const button = document.getElementById('button');
button.addEventListener('click', () => {
  banner.click();
});
```

Los eventos del ratón pueden comprobar si tenemos algunas teclas pulsadas en el momento en el que se dispara el evento. Las siguientes propiedades indican si la respectiva tecla está pulsada:

- `shiftKey`: Shift.
- `altKey`: Alt u Opt para Mac.
- `ctrlKey`: Ctrl
- `metaKey`: Cmd para Mac.

Reescribimos el ejemplo anterior para forzar un click en el banner si tenemos la tecla *control* pulsada.

```
button.addEventListener('click', (event) => {
  if (event.ctrlKey) banner.click();
});
```

7.4.2. Eventos `mousedown` y `mouseup`

Estos eventos se disparan en el momento de pulsar un botón del ratón y al liberarlo respectivamente. Se utilizan para cualquier botón del ratón, por lo que nos puede interesar saber cuál es el botón pulsado. La propiedad `button` del evento nos indica cuál es el botón pulsado. Los valores de esta propiedad son:

- Botón izquierdo (primario): 0
- Botón central (auxiliar): 1
- Botón derecho (secundario): 2
- Botón atrás: 3
- Botón adelante: 4

El siguiente código muestra como detectar estos eventos al pulsar en el banner de la página con cualquier botón.

```
const banner = document.querySelector('.banner');
const message = document.getElementById('message');
banner.addEventListener('mousedown', (event) => {
  banner.style.border = '10px solid red';
  message.innerHTML = event.button;
});
banner.addEventListener('mouseup', (event) => {
  banner.style.border = 'none';
});
```

7.4.3. Eventos `mouseenter`, `mouseleave` y `mousemove`

Vamos a revisar otros eventos de ratón que pueden ser interesantes.

- `Mouseenter`: se produce en el momento de entrar a un elemento.
- `Mouseleave`: se produce al salir del elemento.
- `Mousemove`: se produce al movernos por un elemento.

Para probar los eventos, vamos a mostrar una capa flotante al situarnos en las imágenes de las categorías de la página. En esta capa mostraremos las dimensiones de alto y ancho que ocupan las imágenes exactamente en la página.

El primer paso es seleccionar los elementos donde vamos a añadir los manejadores de eventos. En lugar de utilizar directamente las imágenes, vamos a seleccionar los `DIV` que las contienen, ya que una imagen es un elemento vacío, y nosotros tenemos que añadir la capa con las dimensiones, no se puede hacer dentro de un elemento vacío.

```
let categories = document.getElementById("categories");
let divImages = categories.querySelectorAll(".cat-list-image");
```

Como vamos a asociar el manejador a varios elementos, definimos una función. La función crea el un elemento DIV para pegarlo al elemento con que hemos asociado el manejador. En este nuevo elemento personalizamos los estilos, y como contenido mostramos las propiedades `offsetWidth` y `offsetHeight` con las dimensiones en la página del propio elemento. El API ofrece más propiedades que nos permiten trabajar con las dimensiones del elemento que deberíamos revisar.

```
function showDimensions(event){
  let dimensionsDiv = document.createElement("div");
  dimensionsDiv.classList.add("border");
  dimensionsDiv.classList.add("border-primary");
  dimensionsDiv.classList.add("p-2");
  dimensionsDiv.style.background = "#f5f5f5";
  dimensionsDiv.style.width = "150px";
  let str = "offsetWidth: " + this.offsetWidth + "<br>" + "offsetHeight:
" + this.offsetHeight;
  dimensionsDiv.innerHTML = str;
  this.append(dimensionsDiv);
}
```

La segunda función la utilizaremos para eliminar el elemento.

```
function hideDimensions(event){
  this.children[1].remove();
}
```

Para cada elemento de la colección debemos asociar estos manejadores a los eventos `mouseenter` y `mouserleave`.

```
for (let div of divImages){
  div.addEventListener("mouseenter", showDimensions);
  div.addEventListener("mouseleave", hideDimensions);
}
```

Por la configuración del código HTML, la capa se crea, pero no podemos visualizarla. Vamos a hacer que la capa sea flotante y se mueva con el ratón mientras que estemos situados en la imagen. En la función *showDimensions()* añadimos los siguientes estilos a la capa creada.

```
dimensionsDiv.style.position = "absolute";
dimensionsDiv.style.top = event.offsetY + "px";
dimensionsDiv.style.left = event.offsetX + "px";
```

Le hemos asignado una posición absoluta, por lo que la capa no es renderizada respecto a su elemento padre. En estas capas flotantes debemos indicar la posición respecto a un punto de referencia, la esquina superior izquierda de la página, lo hacemos con las propiedades `left` y

`top` en pixeles, los cuales recogemos de la posición del ratón en el elemento recogidos del evento. Tenemos que observar que CSS hay que indicar la unidad de medida `px` tanto en posiciones como en dimensiones, por esa razón concatenamos el texto a los valores recogidos de `event.offsetY` y `event.offsetX`.

Creamos una función que modifique la posición de la capa en función de la situación del ratón. La capa es el segundo hijo del elemento con el manejador.

```
function moveDimensions(event) {
  this.children[1].style.top = (event.offsetY + 5) + "px";
  this.children[1].style.left = (event.offsetX + 5) + "px";
}
```

Por último, debemos indicar que el elemento con el manejador tiene una posición relativa, esto nos permite que los elementos absolutos estén ubicados respecto a la esquina superior izquierda del elemento contenedor, en lugar de utilizar la página, lo que facilita los cálculos. Además, añadimos el nuevo manejador para el evento `mousemove`. El código queda de la siguiente forma.

```
for (let div of divImages){
  div.style.position = "relative";
  div.addEventListener("mouseenter", showDimensions);
  div.addEventListener("mousemove", moveDimensions);
  div.addEventListener("mouseleave", hideDimensions);
}
```

7.4.4. Eventos `mouseover` y `mouseout`

Tenemos otros dos eventos típicos de ratón. Los eventos `mouserover` y `mouseout` son similares a `mouseenter` y `mouserleave`, pero también se disparan cuando cambiamos entre los elementos interiores donde hemos definido el manejado, lo que puede provocar resultados no esperados cuando la estructura interna tiene varios niveles. Con `mouseenter` y `mouseleave` nos estamos asegurando que los eventos se disparan una única vez.

Para todos los elementos de la página, queremos mostrar un borde cuando el ratón pase por encima de ellos, y lo elimine cuando salga. Añadimos los eventos `mouseover` y `mouseout` al elemento `BODY`, y añadimos las clases con los bordes al `target` del evento. El objeto `document` ya dispone algunas búsquedas como es la selección del elemento `BODY`.

```
document.body.addEventListener("mouseover", function (event){
  event.target.classList.add("border");
  event.target.classList.add("border-danger");
});
document.body.addEventListener("mouseout", function (event){
  event.target.classList.remove("border");
  event.target.classList.remove("border-danger");
});
```

La propiedad `relatedTarget` del evento indica un segundo objetivo con el que el elemento está relacionado. Tanto `mouseover`, `mouseout`, `mouseenter` y `mouseleave` disponen de la propiedad referenciando de qué elemento proviene el ratón, o de qué elemento hemos accedido en función de cada caso. Podemos reescribir el código anterior con un solo evento utilizando esta propiedad.

```
document.body.addEventListener('mouseover', (event) => {
  event.target.classList.add('border');
  event.target.classList.add('border-danger');
  event.relatedTarget.classList.remove('border');
  event.relatedTarget.classList.remove('border-danger');
});
```

7.5. Eventos de foco

Estos eventos son disparados al recibir el foco, evento `focus`, o al perderlo, evento `blur`.

https://developer.mozilla.org/en-US/docs/Web/API/FocusEvent

Tenemos que tener la precaución en estos eventos ya que la referencia `this` será el objeto `window` que representa el navegador, aunque `event.target` será el elemento sobre el que ponemos el foco o lo perdemos respectivamente.

En el formulario de envío de correo electrónico para la suscripción, vamos a mostrar un mensaje de texto al situarnos en el `INPUT` que recoge el email, y eliminar el mensaje al cambiar de elemento por perder el foco.

Seleccionamos primero el `INPUT` y añadimos los dos eventos. Tan solo destacar la forma de seleccionar el `DIV` creado con el texto para borrarlo al perder el foco, que lo hacemos a través de los elementos hermanos del `INPUT`.

```
const input = document.querySelector("input[name = 'email']");
input.addEventListener('focus', (event) => {
  const div = document.createElement('div');
  event.target.parentElement.append(div);
  div.innerText = 'Introduce un correo electrónico.';
});
input.addEventListener('blur', (event) => {
  event.target.nextElementSibling.nextElementSibling.remove();
});
```

Los elementos tienen dos métodos para forzar el pase del foco o perderlo:

- `focus()`: Hacemos que el elemento tenga el foco.
- `blur()`: Hacemos que el elemento pierda el foco.

Trabajaremos más a fondo estos evento con el uso de formularios.

7.6. Eventos de teclado

Los tipos de evento de teclado se producen al pulsar una tecla `keydown` y al liberarla `keyup`.

Las dos propiedades que vamos a utilizar son `key` con la tecla que ha sido pulsada, y `code` con el código asociado a la tecla, el cual depende de la ubicación del teclado.

https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent

Los eventos de teclado solo deberían ser usados cuando queremos manejar acciones de teclado. Por ejemplo, para reaccionar a las teclas de flecha Up y Down, atajos de teclado o detectar combinaciones de teclas.

7.6.1. Evento keydown

Vamos a reproducir cada carácter y la tecla pulsada al escribir en el elemento `input` para enviar el correo de suscripción. Seleccionamos el elemento, y creamos un `DIV` para mostrar el resultado.

Vamos a pasar el foco directamente a este elemento `INPUT` seleccionado, lo hacemos con el método `focus()`.

```
let input = document.querySelector("input[name = 'email']");
let div = document.createElement("div");
input.parentElement.append(div);
input.focus();
```

Ahora debemos asociar el evento `keydown` al `INPUT` para que muestre las propiedades `key` y `code`. Para evitar hacer múltiples concatenaciones de string, vamos a crear un nodo de texto y lo añadimos como hijo en el `DIV`. El método `normalize()` permite fusionar en uno todos los nodos de texto de un elemento, y borrar los nodos de texto sin contenido.

```
input.addEventListener("keydown", function(event){
  this.parentElement.append(
    document.createTextNode(`${event.key}(${event.code}) `)
  );
  this.normalize();
});
```

¿Qué sucede cuando dejamos una tecla presionada? La pulsación se autorepite y el evento se dispara una y otra vez. Cuando se suelta se dispararía el evento `keyup`. Para saber si estamos dejando pulsada una tecla disponemos de la propiedad `repeat` que establecerá un valor de `true`.

7.6.2. Evento keyup

Vamos a mostrar capas con las dimensiones de las imágenes de las categorías y a ocultarlas. En función de la tecla numérica pulsada, y en combinación con la tecla ALT, mostraremos las dimensiones en base a su posición.

Seleccionamos las imágenes con las que vamos a trabajar como hicimos en el ejercicio anterior. Lo hacemos de manera global porque esta selección la reutilizaremos muchas veces en la ejecución de la página, y siempre serán las mismas

```
let categories = document.getElementById("categories");
let divImages = categories.querySelectorAll(".cat-list-image");
```

Asociamos el evento `keydown` para mostrar las dimensiones en el objeto `document`, ya que se tiene que capturar en toda la página.

```
document.addEventListener("keydown", function(event){
});
```

Tenemos que hacer varias comprobaciones antes de mostrar la capa. La primera es saber si está pulsada la tecla ALT. La propiedad `altKey` del evento nos dice si la tecla está pulsada en un booleano. Existen otras propiedades similares para las teclas CTRL o SHIFT.

```
if (event.altKey){
}
```

A través de la propiedad `code`, verificamos que se ha pulsado una tecla numérica. Si comienza con `Digit` es un digito, y si comienza con `Numpad` es del teclado numérico.

```
if (event.code.indexOf("Numpad") > -1 ||
event.code.indexOf("Digit") > -1){ }
```

Una vez verificado, recogemos el número pulsado del con el substring.

```
let number = (event.code.length === 7)?
  event.code.substring(6):
  event.code.substring(5);
```

La última comprobación es que el número recogido sea menor que el número de imágenes disponibles. Transformamos el número en un entero ya que es un string. Además, chequeamos que la capa no exista todavía. Si dejamos las teclas pulsadas el evento se ejecuta constantemente, por lo que evitamos crear N capas iguales.

```
number = +number;
if (number < divImages.length && divImages[number].children.length < 2){
}
```

Generamos la capa.

```
let dimensionsDiv = document.createElement("div");
dimensionsDiv.classList.add("border");
dimensionsDiv.classList.add("border-primary");
dimensionsDiv.classList.add("p-2");
dimensionsDiv.style.background = "#f5f5f5";
dimensionsDiv.style.width = "150px";
dimensionsDiv.style.position = "absolute";
dimensionsDiv.style.top = 0 + "px";
dimensionsDiv.style.left = 0 + "px";
let str = "offsetWidth: " + divImages[number].offsetWidth + "<br>" + "off
setHeight: " + divImages[number].offsetHeight;
dimensionsDiv.innerHTML = str;
```

Y la añadimos a la imagen en función de su posición.

```
divImages[number].style.position = "relative";
divImages[number].append(dimensionsDiv);
```

El segundo evento es eliminar la capa cuando dejemos de pulsar las teclas. Las comprobaciones son las mismas, que hemos hecho hasta ahora, y tan solo eliminamos la capa seleccionada.

```
document.addEventListener("keyup", function (event) {
  if (event.altKey) {
    if (event.code.indexOf("Numpad") > -1 ||
      event.code.indexOf("Digit") > -1) {
      let number = (event.code.length === 7) ?
        event.code.substring(6) :
        event.code.substring(5);
      number = +number;
      if (number < divImages.length) {
        divImages[number].children[1].remove();
      }
    }
  }
});
```

8. Propagación de los eventos

Como ya hemos comentado, los eventos se propagan a lo largo de los objetos del árbol DOM de la página. La propagación es el proceso por el cual invocamos los manejadores asociados a los elementos de árbol DOM, con un objeto `evento` con la información más relevante del evento. Los manejadores son invocados en el orden que sigue la propagación.

La ruta que recorre los elementos en el proceso es estática, y es establecida al inicializar el evento. La ruta comienza en el objeto `window`, aunque este objeto no forma parte de DOM, enlaza con la rama de objetos DOM desde `document` hasta el evento donde se ha desencadenado el evento, `Event.target`.

```
window -> document -> HTML -> BODY -> … -> Event.target
```

La propagación se divide en tres fases principales:

1. **Fase de Captura (Capturing Phase)**: El evento comienza su propagación desde el elemento superior del árbol DOM hacia el elemento específico que originó el evento. Durante esta fase, los eventos se capturan en los elementos padre antes de llegar al elemento de destino.
2. **Fase Objetivo (Target Phase)**: Una vez que el evento alcanza el elemento de destino (el elemento específico donde ocurrió el evento), se produce la fase objetivo. Aquí, el evento está en su punto más específico.
3. **Fase de Burbujeo (Bubbling Phase)**: Después de la fase objetivo, el evento comienza a propagarse hacia arriba desde el elemento de destino hacia el elemento superior del árbol DOM. Durante esta fase, los eventos se propagan a través de los elementos padre en el orden inverso al de la fase de captura.

La propagación del evento puede ser bidireccional, se hace tanto de bajada como de subida. Cuando definimos un manejador decidimos en qué fase queremos que se invoque. A través de las opciones del método `addEventListener` elegimos en qué fase capturar el evento.

- **Bubbling**: `{capture: false}`.
- **Capturing**: `{capture: true}`.

8.1. Ejemplo Bubbling

Es el proceso de propagación por defecto. Vamos a declarar manejadores de eventos en los siguientes elementos:

- Imágenes en categorías.
- Elementos `DIV` individuales de cada categoría.
- Contenedor `DIV` de categorías.
- `BODY`
- `HTML`
- `Document`
- `window`

En el siguiente código añadimos un manejador a los elementos anteriores para añadir un borde cuando sean cliqueados, además debemos mostrar elemento cliqueado en la consola.

```
let categories = document.getElementById("categories");
let divImages = categories.querySelectorAll(".cat-list-image");

function getElement(event){
  this.style.border = "2px solid red";
  console.log("Target: " + event.target.nodeName + " Element: " + this.no
deName)
}

for (let div of divImages){
  div.firstElementChild.addEventListener("click", getElement);
  div.addEventListener("click", getElement);
}
categories.addEventListener("click", getElement);
document.body.addEventListener("click", getElement);
document.documentElement.addEventListener("click", getElement);
document.addEventListener("click", () => {console.log("document")});
window.addEventListener("click", () => {console.log("window")});
```

Al lanzarse el evento `click` desde una de las imágenes, los manejadores se ejecutan desde la imagen como elemento target, hasta el objeto `window`.

8.2. Detener la propagación

Podemos detener la propagación del evento con el método `stopPropagation()` del evento. Modificamos el ejemplo anterior para detener la propagación en el elemento `BODY`.

```
document.body.addEventListener("click", (event) => {
  event.stopPropagation();
});
```

El método `stopImmediatePropagation()` se utilizaría en el caso de tener más manejadores de eventos asociados al mismo tipo de evento, por ejemplo, tener dos manejadores diferentes para el evento *click*, en este caso el método detendría la propagación y también el segundo manejador asociado al *click*.

8.3. Ejemplo Capturing

Si repetimos el código anterior configurando los manejadores como *capturing* el resultado es el mismo, pero el orden de invocación comienza en `window` y termina en la imagen.

```
for (const div of divImages) {
  div.firstElementChild.addEventListener('click', getElement, {capture:
true});
  div.addEventListener('click', getElement, {capture: true});
}
categories.addEventListener('click', getElement, {capture: true});
document.body.addEventListener('click', getElement, {capture: true});
document.documentElement.addEventListener('click', getElement, {capture:
true});
document.addEventListener('click', () => { console.log('document'); },
{capture: true});
```

```
window.addEventListener('click', () => { console.log('window'); },
{capture: true});
```

8.4. Cancelar acciones por defecto

Cada evento tiene asociadas una serie de acciones por defecto que, aunque paremos la propagación, estas acciones se siguen produciendo. Algunos ejemplos de estas acciones podrían ser:

- Detener el envío de datos de un formulario en el evento *submit*.
- Chequear un botón de tipo radio, o de lista de chequeo.
- Ir a la ubicación asociada al elemento anchor al cliquear en él.

Vamos a cancelar esta tercer acción utilizando el método `preventDefault()` del evento.

El siguiente código añade un manejador a cada uno de los enlaces de servicios para generar un `alert()`. La acción por defecto del evento en el elemento `a` es ir a la URL indicada en el enlace.

```
let links = document.querySelectorAll('#footer div.row div:nth-child(3)
ul li a');
for (let link of links){
  link.addEventListener("click", function(event){
    alert(this.innerText);
  })
}
```

Añadimos el siguiente código al manejador para cancelar las acciones por defecto.

```
event.preventDefault();
```

En ocasiones la captura de un evento debería implicar la cancelación de la acción por defecto de otro evento, veamos un ejemplo. Si vinculamos una manejador al evento *dblclick*, al realizar la acción el texto quedará también seleccionado. Vamos añadir un manejador sobre el elemento *message* de la página.

```
const message = document.getElementById('message');
message.addEventListener('dblclick', function () {
  alert(`Este es mi identificador: ${this.id}`);
});
```

Para evitar este efecto, debemos añadir un segundo manejador sobre el evento *mousedown* que cancele la acción por defecto.

```
message.addEventListener('mousedown', (event) => {
  event.preventDefault();
});
```

Otro ejemplo es si tuviéramos que trabajar como menús contextuales. Tenemos que prevenir la acción por defecto para evitar mostrar el menú contextual del documento.

```
message.addEventListener('contextmenu', function (event) {
  alert(`Este es mi identificador: ${this.id}`);
  event.preventDefault();
});
```

Si queremos saber si se ha cancelado la acción por defecto de un evento tenemos la propiedad `defaultPrevented` del evento. La propiedad la podemos utilizar en una propagación. En el siguiente ejemplo añadimos un menú contextual en el documento, para evitar que se muestren ambos menús

```
document.addEventListener('contextmenu', (event) => {
  if (!event.defaultPrevented) {
    alert('contextmenu: document');
    event.preventDefault();
  }
});
```

Una de las opciones de configuración de un manejador de eventos en `addEventListener` es el **control pasivo** con el objeto `{passive:true}`. Esta opción indica al navegador que no se invocará al método `preventDefault()`.

La razón de esta configuración es para eventos como `touchmove` para dispositivos móviles cuando el usuario mueve el dedo por la pantalla, lo que provoca una acción por defecto de desplazamiento. Para evitarlo necesitamos invocar `preventDefault()`.

En este caso, el navegador debe procesar todos los manejadores, y luego, si no se ha invocado `preventDefault()` en ningún punto del código, continuar con el desplazamiento, lo que provoca retrasos y saltos repentinos en el interfaz.

El control pasivo indica que no se cancelará la acción por defecto de desplazamiento, con lo que puede realizar el desplazamiento de inmediato sin tener que esperar a computar el resto de manejadores, mejorando la experiencia del usuario. Algunos navegadores como Firefox o Chrome configuran esta opción por defecto para los eventos `touchstart` y `touchmove`.

8.5. Delegación de eventos

Qué ocurre cuando tenemos muchos elementos idénticos, es decir, con la misma estructura y en el mismo contenedor, que dan respuesta a un tipo de evento en concreto. Primeo deberíamos seleccionar todos esos elementos, para posteriormente recorrerlos añadiendo un manejador de eventos cada uno de ellos.

La **delegación de eventos** es una técnica que implica asignar un controlador de eventos a un elemento contenedor en lugar de a los elementos individuales dentro de ese contenedor. En lugar de manejar eventos en cada elemento específico, se aprovecha la propagación de eventos para capturar y gestionar los eventos en un nivel superior del árbol DOM. Tenemos una estructura con estas características en la capa con identificador *tiles*, la cual contiene una serie de tarjetas organizadas en filas. La estructura es la siguiente:

```
<div id="tiles" class="container" style="display: none;">
    <div class="row py-2">
        <div class="col-sm-4">
            <div class="card">
                <div class="card-body">
                    <h5 class="card-title">Noroeste</h5>
                    <p class="card-text">Descripción.</p>
                    <a href="#" class="btn btn-primary">Acción</a>
                </div>
            </div>
        </div>
        ...
    </div>
    ...
</div>
```

Queremos modificar el color de fondo cada vez que cliqueamos en una tarjeta, pero, vamos a capturar el evento en el elemento contenedor. Utilizamos la propiedad `target` del evento para saber dónde se ha disparado el evento, y a través del método `closest('.card')` detectamos la capa donde se inicia la tarjeta. Construimos un objeto manejador para recordar cuál es la tarjeta activa y poder desactivarla al cliquear en una nueva tarjeta.

```
class TilesHandler {
  constructor() {
    this.selectedTile = null;
  }

  handleEvent(event) {
    const target = event.target.closest('.card');
    if (event.currentTarget.contains(target)) {
      if (this.selectedTile) {
        this.selectedTile.classList.remove('bg-danger');
        this.selectedTile.classList.remove('text-white');
      }
      this.selectedTile = target;
      this.selectedTile.classList.add('bg-danger');
      this.selectedTile.classList.add('text-white');
    }
  }
}
```

Asignamos un objeto manejador como manejador de eventos para el contenedor.

```
const tiles = document.getElementById('tiles');
tiles.style.display = 'block';
const th = new TilesHandler();
tiles.addEventListener('click', th);
```

8.6. Eventos personalizados

Podemos crear nuestros propios eventos para por ejemplo implementar pruebas automatizadas. Tenemos la clase nativa `Event` que nos permite instanciar nuevos eventos. Los argumentos que admite son:

```
let event = new Event(type[, options]);
```

- *type*: indica el tipo de evento, pudiendo ser “click” o crear uno propio. “mi-evento”.
- *options:* Admite dos propiedades para indicar si el evento se va a propagar, o si lo podemos cancelar la acción por defecto. Por defecto los valores son `{bubbles: false, cancelable: false}` para que el evento no se propague y que no se pueda cancelar la acción por defecto.

Para lanzarlo lo haremos a partir de un elemento y utilizando el método `elem.dispatchEvent(event)`. El API nos permite saber si un evento se ha creado por un usuario real o si se ha hecho en un script. La propiedad `event.isTrusted` tiene valor `true` si lo ha creado un usuario real.

Vamos a crear un manejador de eventos que muestre la información del elemento que dispara el evento, el elemento donde se captura y si el evento es confiable o no.

```
function getElement(event) {
  console.log(`Target: ${event.target.nodeName} Element:
${this.nodeName}`);
  console.log(event.isTrusted);
}
```

Vamos a configurar manejadores de eventos para el evento *click* en desde las imágenes de las categorías y cada uno de sus elementos padre hasta el documento.

```
const categories = document.getElementById('categories');
const divImages = categories.querySelectorAll('.cat-list-image');

for (const div of divImages) {
  div.firstElementChild.addEventListener('click', getElement);
  div.addEventListener('click', getElement);
}
categories.addEventListener('click', getElement);
document.body.addEventListener('click', getElement);
document.documentElement.addEventListener('click', getElement);
document.addEventListener('click', getElement);
```

Creamos un evento del tipo *click* y que se pueda propagar, lo disparamos a partir de una de las categorías.

```
const clickEvent = new Event('click', {
  bubbles: true,
});
divImages[0].dispatchEvent(clickEvent);
```

Al ejecutarse veremos que los eventos no son confiables al ser creados en código. Si ahora cliqueamos en una de las categorías veríamos que si son confiables.

El evento creado es genérico. Si quisiéramos añadir propiedades especificas en función del tipo de evento, debemos instanciar el evento con la debida subclase y en las opciones añadir las propiedades que consideremos. Modificamos el ejemplo anterior para instanciar un objeto `MouseEvent` y configuramos las propiedades `clientX` y `clientY`. El manejador mostrará las propiedades.

Añadimos la siguiente línea al manejador.

```
console.log(`clientX: ${event.clientX} clientY: ${event.clientY}`);
```

Y modificamos el evento.

```
const clickEvent = new MouseEvent('click', {
  bubbles: true,
  clientX: 100,
  clientY: 200,
});
```

La clase `CustomEvent` permite que creemos nuestros propios eventos. Se trata de una subclase de `Event` que añade una propiedad `detail` para que introduzcamos cualquier dato que queramos hacer llegar al manejador. Vamos a repetir el ejemplo anterior, pero asignando nuestro propio evento, el cual configuraremos la propiedad `detail`.

Definimos el un nuevo manejador para que muestre los datos de la propiedad `detail`.

```
function getElementInCustomEvent(event) {
  console.log(`Target: ${event.target.nodeName} Element:
${this.nodeName}`);
  console.log(`prop1: ${event.detail.prop1} prop2:
${event.detail.prop2}`);
}
```

Asignamos el manejador.

```
for (const div of divImages) {
  div.firstElementChild.addEventListener('myevent',
getElementInCustomEvent);
  div.addEventListener('myevent', getElementInCustomEvent);
}
categories.addEventListener('myevent', getElementInCustomEvent);
document.body.addEventListener('myevent', getElementInCustomEvent);
document.documentElement.addEventListener('myevent',
getElementInCustomEvent);
document.addEventListener('myevent', getElementInCustomEvent);
```

Creamos el evento y lo lanzamos.

```
const myevent = new CustomEvent('myevent', {
  bubbles: true,
  detail: {
    prop1: 'value1',
    prop2: 'value2',
  },
});
divImages[0].firstElementChild.dispatchEvent(myevent);
```

Los **eventos producidos en una página siempre son asíncronos**, cada vez que se produce un evento se encolará y si atenderá cuando llegue su turno. Esta afirmación tiene su excepción, **cuando creamos y lanzamos un evento dentro del manejador de eventos se gestionará de forma síncrona**. El siguiente ejemplo muestra como la ejecución de un manejador es pausada al lanzar un evento interno, para posteriormente seguir con su ejecución.

```
const button = document.getElementById('button');
button.onclick = function (event) {
  alert('Primera parte del evento');

  button.dispatchEvent(new CustomEvent('myevent', {
    bubbles: true,
  }));

  alert('Segunda parte del evento');
};
document.addEventListener('myevent', () => alert('Evento anidado'));
```

9. Proyecto

Vamos a generar un área en una página HTML con un cubo dentro del área. Mediante el teclado queremos registrar diferentes acciones:

- Cursor izquierdo: Mueve el cubo a la izquierda.
- Cursor derecho: Mueve el cubo a la derecha.
- Cursor arriba: Mueve el cubo hacía arriba.
- Cursor abajo: Mueve el cubo hacía abajo.
- Tecla "C": cambia el cubo de color de forma aleatoria.

Los movimientos no pueden sobrepasar los límites del área.

Por cada pulsación de tecla la tenemos que registrar, para que cuando pulsemos ENTER se ejecuten todas las acciones registradas de forma secuencial, y en el orden introducido. Las acciones deben mostrarse en el área mediante etiquetas `SPAN`. Cliqueando en uno de los `SPAN` implica que la acción sea borrada de la secuencia.

9.1. Paso 1: Registro de área y cubo

Utilizamos dos variables globales para ambos elementos, *area* y *cube*. El área tiene posición relativa, y el cubo como absoluta para que lo podamos mover tomando como referencia el área.

```
// Variables globales
let area;
let cube;

// Área para el proyecto
let main = document.getElementsByTagName("main")[0];
area = document.createElement("div");
area.classList.add("container");
area.style.border = "2px solid red";
area.style.height = "400px";
area.style.position = "relative";
main.before(area);

// Pieza que queremos mover
cube = document.createElement("div");
cube.style.background = "red";
cube.style.width = "50px";
cube.style.height = "50px";
cube.style.position = "absolute";
cube.style.top = "100px";
cube.style.left = "150px";
area.append(cube);
```

9.2. Paso 2: Manejador de captura de teclas

Creamos un evento `keydown` para capturar las teclas. Mediante un `switch` chequeamos la tecla para ejecutar una acción. Cancelamos las acciones por defecto tras pulsar la tecla.

```
document.addEventListener("keydown", function (event) {
  switch (event.code){
    case "ArrowUp":
      moveUp(cube);
      break;
    case "ArrowDown":
      moveDown(cube);
      break;
    case "ArrowLeft":
      moveLeft(cube);
      break;
    case "ArrowRight":
      moveRight(cube);
      break;
    case "KeyC":
      randomColor(cube);
      break;
    default:
      break;
  }
  event.preventDefault();
});
```

Creamos las funciones para que ejecuten cada una de las acciones.

```
function moveUp(cube){
  let top = cube.offsetTop;
  top -= 10;
  top = (top < 0)? 0 : top;
  cube.style.top = top + "px";
}

function moveDown(cube){
  let top = cube.offsetTop;
  top += 10;
  top = (top > area.offsetHeight - cube.offsetHeight)? area.offsetHeight
- cube.offsetHeight : top;
  cube.style.top = top + "px";
}

function moveLeft(cube){
  let left = cube.offsetLeft;
  left -= 10;
  left = (left < 0)? 0 : left;
  cube.style.left = left + "px";
}
```

```
function moveRight(cube){
  let left = cube.offsetLeft;
  left += 10;
  left = (left > area.offsetWidth - cube.offsetWidth)? area.offsetWidth -
 cube.offsetWidth : left;
  cube.style.left = left + "px";
}

function randomColor(cube){
  let r = Math.floor((Math.random() * 256));
  let g = Math.floor((Math.random() * 256));
  let b = Math.floor((Math.random() * 256));
  cube.style.background = `rgb(${r}, ${g}, ${b})`;
}
```

El método `getBoundingClientRect()` facilita acceso a las dimensiones, al contener todos los valores en un objeto literal.

9.3. Paso 3: Registro de acciones

Definimos una variable global para contener un array con el registro de acciones.

```
let acctions = [];
```

Creamos una función para registrar cada una de las acciones. En el array añadimos un objeto literal con el texto de la acción y el `SPAN` que hemos creado, el cual utilizaremos más adelante.

```
function addAction(action){
  let span = document.createElement("span");
  acctions.push({
    action: action,
    span: span
  });
  span.textContent = action;
  span.style.padding = "10px";
  span.style.border = "1px solid #ddd";
  span.style.display = "block";
  span.style.float = "left";
  span.style.margin = "2px";
  span.style.cursor = "pointer";
  span.addEventListener("mouseenter", function () {
      this.style.backgroundColor = "red";
      this.style.color = "white";
  })
  span.addEventListener("mouseleave", function () {
      this.style.backgroundColor = "white";
      this.style.color = "black";
  })
  area.append(span);
}
```

Tenemos que corregir el evento `keydown` para invocar el registro de las acciones.

```
document.addEventListener("keydown", function (event) {
  console.log(event.code);
  switch (event.code){
    case "ArrowUp":
      addAction("up");
      break;
    case "ArrowDown":
      addAction("down");
      break;
    case "ArrowLeft":
      addAction("left");
      break;
    case "ArrowRight":
      addAction("right");
      break;
    case "KeyC":
      addAction("color");
      break;
    default:
      break;
  }
  event.preventDefault();
});
```

9.4. Paso 4: Ejecutar las acciones

En una función recogemos el primer movimiento realizado, lo ejecutamos en base a un `switch`, y borramos el `SPAN`. Para darle dinamismo, utilizamos una función `setTimeOut()` que ejecuta una función en base a un periodo de tiempo en milisegundos. Estaremos ejecutando una función recursiva, siendo el caso base para finalizar la ejecución el caso en el que el array es vacío.

```
function executeAcctions(){
  if (acctions.length > 0){
    let action = acctions.shift();
    switch(action.action){
      case "up":
        moveUp(cube);
        break;
      case "down":
        moveDown(cube);
        break;
      case "left":
        moveLeft(cube);
        break;
      case "right":
        moveRight(cube);
        break;
      case "color":
```

```
        randomColor(cube);
        break;
      default:
        break;
    }
    action.span.remove();
    setTimeout(executeAcctions, 50);
  }
}
```

Añadimos una opción para capturar la tecla ENTER en el evento `mousedown`.

```
    case "Enter":
      executeAcctions();
      break;
```

9.5. Paso 5: Borrar acciones

Al cliquear el elemento `SPAN` debemos borrar la acción de la lista y del array. Utilizamos la propiedad *span* que hemos añadido en el objeto literal del array para hacer la comparación con el objeto cliqueado. Añadimos el evento `click` a cada `SPAN` creado para que se compare con los que hemos guardado en el array de acciones.

```
span.addEventListener("click", function () {
  let index = acctions.findIndex((action) => {
    return action.span === this;
  })
  acctions.splice(index,1);
  this.remove();
})
```

10. Conclusiones

El API DOM es la solución nativa para manipular e interactuar con el documento HTML que hemos cargado en el navegador. DOM es intuitiva, pero es muy extensa. En este documento tan solo hemos visto una parte significativa de qué podemos hacer con el API, pero no hemos visto todo lo que ofrece. Es conveniente revisar los objetos que tiene el API, sus propiedades, los eventos disponibles, para saber todo lo que nos puede ofrecer. No es necesario aprenderlos de memoria, eso lo haremos conforme vayamos usando DOM, pero si que nos puede ayudar para no reinventar la rueda. Muchas veces un problema tiene una fácil solución tan solo revisando las alternativas del API.