Tema 4: Gestión del DOM

¿Qué aprenderás?

- Reconocer el modelo de objetos de un documento web.
- Modificar la página web añadiendo estilos a los elementos HTML.
- Crear y eliminar elementos del código HTML.
- Navegar a través de los elementos del código HTML.

¿Sabías que…?

- Mozilla Firefox retorna los colores CSS obtenidos desde JavaScript en RGB en vez de hexadecimal.
- En el navegador Internet Explorer definir una variable o función global con un nombre igual al id de un elemento HTML provocaba un conflicto.
- Cada tipo de elemento HTML tiene sus propios métodos y propiedades.
- Apache Cordova permite crear aplicaciones móviles con JavaScript y HTML.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

4. Manipulación del DOM

En el desarrollo de aplicaciones web, una de las acciones más comunes que querremos haces es manipular la estructura HTML de alguna forma. Ya sea creando nuevas marcas, modificando sus estilos o eliminando marcas para que el usuario las deje de ver. Todas éstas acciones las realizaremos a través del Document Object Model , es decir, el DOM.

4.1. El modelo de objetos del documento

El DOM es una representación del documento HTML en el que se relacionan todos los nodos que forman el documento según su descendencia en una estructura de árbol en la que el elemento inicial es la marca HTML. Éste árbol es utilizado por el propio navegador para renderizar el contenido y aplicar estilos y puede ser utilizado por los lenguajes de programación para interpretar el documento HTML.

Los nodos que forman un documento HTML se pueden dividir en 5 categorías principales:

- **Document** : es el nodo raíz y contiene todos los nodos.
- **Los nodos de tipo Comment o comentarios**: representan los comentarios del HTML.
- **Los nodos de tipo Element o elementos**: representan las marcas que forman el HTML. Éstos son los nodos con los que trabajaremos principalmente.
- **Los nodos de tipo Text o textos**: representan los textos contenidos en las marcas.
- **Los nodos de tipo Attribute o atributos**: son los atributos de las marcas.

En el siguiente ejemplo podemos ver un documento HTML y su representación del DOM.

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo DOM</title>
</head>
<body>
    <!-- Cuerpo del HTML-->
    <h1 id="título"> El DOM</h1>
</body>
</html>
```

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Como podemos ver en el gráfico anterior cada nodo tiene como mínimo un padre (excepto el nodo raíz). A demás en el gráfico podemos observar que los nodos de tipo texto no pueden tener hijos ni hermanos y que un nodo solo puede tener un único padre.

El DOM además de estructurar y clasificar los nodos que forman el documento HTML nos ofrece una API (un conjunto de métodos y atributos) para poder interaccionar con los nodos.

Es importante destacar que el DOM está íntimamente relacionado con el código HTML de tal forma que si el código sufre un cambio también lo sufrirá el DOM, e igualmente si el DOM sufre algún cambió éste se verá reflejado en el HTML y por lo tanto en la representación de éste. Por ejemplo, si cambiamos un estilo en el DOM automáticamente se cambiará en la visualización del HTML.

Para poder interaccionar con un nodo lo primero que tenemos que conocer es la forma de acceder a él. A continuación vamos a ver distintas formas de acceder a uno o varios nodos.

4.2. Acceso a los nodos desde document

Como “document” representa el nodo raíz del DOM , será a partir de él que podremos acceder a los distintos nodos que lo forman. Cuando accedamos a un nodo lo que obtendremos será un objeto que representará ése nodo y que estará vinculado a ése nodo. Si realizamos alguna modificación a las propiedades del objeto, el nodo se actualizará automáticamente reflejando los cambios. E igualmente si el nodo sufre de una modificación, el objeto que le haga referencia también será modificado.

VERSIÓN IMPRIMIBLE ALUMNO LIKIAEP

El objeto global “document” contiene un conjunto de propiedades que hacen referencia al “body” , “head”, todos los links, imágenes o formularios. Accediendo a éstas propiedades es una de las formas más directas que ofrece “document” para acceder a sus nodos. A continuación se muestran las propiedades principales de “document”:

propiedad	Descripción
document.body;	Retorna un objeto del que hará referencia a nodo BODY
document.head;	Retorna un objeto que hará referencia al nodo HEAD
document.links;	Retorna un array que contendrá, por cada uno de los links de la página web , un objeto representativo.
document.images;	Retorna un array que contendrá, por cada una de las imágenes de la página web , un objeto representativo.
document.forms;	Retorna un array que contendrá, por cada uno de los formularios de la página web , un objeto representativo.

Éstas formas de acceder a los elementos son muy generales y en ocasiones no nos permita acceder al nodo que deseamos. Por suerte “document” también nos ofrece formas de acceso más específicas como por ejemplo acceder a un elemento a través de su atributo “id”.

4.2.1. Acceso a un nodo por id

Es muy importante recordar que en HTML el atributo “id” ha de ser único dentro de un mismo documento. Podemos acceder rápidamente a un nodo a través de su atributo “id” con el método *“getElementById()”* de “*document”* pasando por parámetro el *“id”* del *“element”* al que queremos acceder. Si sintaxis sería:

document.getElementById(“idElemento”);

VERSIÓN IMPRIMIBLE PARA USO LINKIA

Éste método nos retornará un objeto que será una referencia al elemento con el "id" indicado. En el siguiente código de ejemplo obtenemos un objeto de referencia al body, un array con referencia a los links y un objeto referenciado al elemento con id "titulo":

```html
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo DOM</title>
    <script src="dom.js" defer></script>
</head>
<body>
    <h1 id="titulo"> El DOM</h1>
    <a href="#">Link1</a>
    <a href="#">Link2</a>
</body>
</html>
```

dom.is

```javascript
//accedemos al body
let nodoBody = document.body;

//accedemos a los links
let nodosLink = document.links;

//accedemos al elemento con id
"titulo"
let nodoH1 =
document.getElementById("titulo"
);
```

ATENCION! Es muy importante destacar en éste código el uso del atributo ***defer*** que vimos al inicio del módulo. Recordar que éste atributo obliga a ejecutar el JavaScript después de haber cargado todo el documento HTML. Si no lo ponemos no podrá acceder a ningún elemento del HTML, ya que el JavaScript está situado antes del Body.

VERSIÓN IMPRIMIBLE AL

4.2.2. Acceso a múltiples nodos

Dentro del objeto “document” podemos encontrar también otros métodos para acceder a múltiples nodos del tipo “element” según marca, clase o el “name”. Al seleccionar un conjunto de nodos obtendremos un objeto del tipo “NodeList” o “HTMLCollection”. Ambos tipos de datos los podemos tratar como arrays formados por el conjunto de nodos que seleccionemos pudiéndolo recorrer con un FOR o con un forEach. Los principales métodos para obtener un conjunto de elementos según la marca, clase o “*name*” son:

método	Descripción
document.getElementsByTagName("a");	Retorna un NodeList formado por todos los “element” que sean del tipo indicado por parámetro. En éste caso todas las marcas <a>.
document.getElementsByName("email");	Retorna un NodeList formado por todos los “element” que sean con el atributo “name” pasado por parámetro. En éste caso todos los “element” con un atributo “name” igual a “email”.
document.getElementsByClassName("rojo");	Retorna un NodeList formado por todos los “element” que sean de la clase indicada por parámetro. En éste caso todos los “element” donde una de sus clases sea “rojo”.

A continuación vemos un ejemplo de utilización de los métodos vistos en la tabla anterior:

```
<head>
   <title>Ejemplo DOM</title>

   <script src="dom.js" defer></script>
</head>
<body>
   <intpu name="email"   class="rojo"/>
   <a href="#" class="rojo"> Link1  </a>
   <a href="#">Link2</a>
</body>
```

```
let links = document.getElementsByTagName("a");
let inputs = document.getElementsByName("email");
let rojos = document.getElementsByClassName("rojo");

console.log(links.length); // muestra un 2
let link1 = links[0]; //accedemos al 1r link
let link2=links[1]; //accedemos al 2o link

console.log(inputs.length); // muestra un 1
let email =inputs[0]; //accedemos al input email

console.log(rojos.length);// muestra un 2
email=rojos[0]; //accedemos al input email
link1=rojos[1]; //accedemos al 1r link
```

4.2.3. Acceso mediante selectores

También podemos acceder a un elemento utilizando los selectores CSS gracias al método de document "*querySelector*()". Éste método retorna el primer elemento que satisface el selector CSS empezando desde el nodo raíz.

Asimismo todos los nodos "element" tienen también éste método. Al ejecutarse retornaran el primer elemento que satisfazca el selector CSS buscando dentro del nodo actual.

Si en vez de obtener un solo nodo como respuesta queremos obtener todos los nodos que satisfazcan el selector CSS, entonces utilizaremos el método "querySelectorAll()" , tanto desde document como desde un objeto referenciado a un nodo "element". Éste método nos retornará una lista de nodos, pero cuidado porque no se actualizará si se modifica el DOM.

En el siguiente ejemplo se muestra el acceso a nodos mediante selectores. Para ilustarlo mejor se ha añadid el HTML antes y después de ejecutarse el JavaScript para mostrar claramente los efectos de JavaScript sobre el HTML.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

HTML antes de ejecutar el JS

```
<body>
  <h1 id="titulo"> El DOM </h1>
  Contenido
  <a href="#" >Link1</a>
  <a href="#" >Link2</a>
</body>
```

BODY después de ejecutar el JS

```
<body
    style="background-color: orange;">
  <h1 id="titulo"> El DOM </h1>
  Contenido
  <a href="#"
        style="text-decoration:none;
          color: green;">
        Link1</a>
  <a href="#" style="color: green;">
    Link2</a>
</body>
```

Documento JavaScript dom.js

```
//seleccionamos la primera marca de nombre body
let nodoBody = document.querySelector("body");
//cambiamos su color de fondo
nodoBody.style.backgroundColor="orange";

//seleccionamos el elemento con id "titulo"
let h1 = document.querySelector("#titulo");
//cambiamos su contenido
h1.style.innerHTML="Ejemplo selectores";

//seleccionamos el segundo Link hijo de body
let segundoLink =
      document.querySelector("body>a:nth-child(2)");
//le quitamos el subrayado
segundoLink.style.textDecoration="none";

//seleccionamos todos los links dentro de body
let links=nodoBody.querySelectorAll("a");
// recorremos los links y los ponemos en verde
links.forEach(function (unLink, key) {
  unLink.style.color="green";
});
```

VERSIÓN IMPRIMIBLE

4.2.4. Propiedades de un nodo

Cada vez que obtenemos una referencia a un nodo podemos interactuar con sus propiedades para modificar sus estilos.

general, cualquier objeto que haga referencia a un nodo tendrá como mínimo una propiedad que se corresponda con todos los atributos naturales que puede tener el nodo como elemento HTML. Es decir, si accedemos a un nodo “A”, podremos acceder y modificar su atributo “href”, si accedemos a un nodo “INPUT” , podremos acceder y modificar su atributo “value”.

A continuación veremos las principales propiedades comunes a las que podemos acceder y/o modificar cuando accedamos a un nodo (exceptuando las de parentesco que veremos en el siguiente punto) :

propiedad	Modificable	Descripción

nodeName	no	Si el nodo es un elemento retorna el nombre de la marca. Si el nodo es un atributo retorna el nombre del atributo. Para otros casos retorna distintos nombres según el tipo.
nodeType	no	Si el nodo es un elemento retorna un 1 Si el nodo es un atributo retorna un 2 Si el nodo es un texto retorna un 3 Si el nodo es un comentario retorna un 8
tagName	No	Solo disponible para nodos de tipo Element, retorna el nombre de la marca.
innerHTML	Si	Retorna el contenido HTML del nodo. Establece el contenido HTML del nodo y es interpretado de nuevo por el navegador.
innerText	Si	Retorna el contenido del nodo indicado (sin mostrar los < >) y todos sus descendientes excepto los ocultos, <script> y <style>. Si se modifica, se sustituirá todo el contenido del nodo por el nuevo valor indicado sin interpretarlo como contenido HTML.
textContent	Si	Retorna el contenido del nodo indicado (sin mostrar los < >) y todos sus descendientes . Si se modifica, se sustituirá todo el contenido del nodo por el nuevo valor indicado sin interpretarlo como contenido HTML.
id	Si	Establece o retorna el id que tiene el nodo.
classList	Si	Establece o retorna la lista de clases que contiene el nodo tipo element.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

className	Si	Establece o retorna una string con el contenido del atributo “class”
title	si	Establece o retorna el título de un nodo tipo element.
style	Si	Establece o retorna las propiedades CSS aplicadas a un nodo tipo element. No permite modificar la hoja de estilos, sinó que las modificaciones se realizan siempre sobre el atributo *style* de la marca.

4.2.4.1. Estilos de un nodo

A través de la propiedad Style podremos acceder y modificar los estilos de una marca. En general si queremos acceder a un estilo solo deberemos escribir:

.style.nombrePropiedadCSS

Teniendo presente que todas aquellas propiedades CSS que tengan guiones, en nomenclatura JavaScript se elimina el guion y se pone en mayúsculas la segunda palabra. Así por ejemplo la propiedad CSS *backgroud-color* pasa a ser *backgroundColor*.

A continuación se muestra un ejemplo de acceso y modificación de las distintas propiedades de un nodo y sus estilos.

HTML antes de ejecutar el JS

```html
<html><head>
    <title>Ejemplo DOM</title>
    <script src="dom.js" defer>
    </script>
</head><body>
  <h1 id="titulo"> El DOM</h1>
    Contenido
  <a href="#" id="link">Lin</a>
</body></html>
```

BODY después de ejecutar el JS

```html
<body
style="background-color:orange;
text-align: center;">
```

Document JavaScript dom.js

```javascript
let nodoBody = document.body;
//accedemos al elemento con id "titulo"
let nodoH1=document.getElementById("titulo");
//accedemos al elemento con id "link"
let link = document.getElementById("link");

//ocultamo el link
link.style.visibility="hidden";
//el color de fondo de body a naranja
nodoBody.style.backgroundColor="orange";
//centramos el texto en body
nodoBody.style.textAlign="center";
//el texto de H1 en negrita
nodoH1.style.fontWeight="bold";
//la altura del a 100 pixeles
nodoH1.style.height="100px";
```

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

4.2.5. Recorrer los elementos del DOM por parentesco

A partir de una referencia a un nodo podemos , además de modificar sus estilos y atributos, navegar por el árbol del DOM utilizando sus propiedades. Veremos a continuación las propiedades que nos permiten obtener una referencia al resto de nodos que tengan un parentesco directo, es decir: padre, hijos y hermanos.

En el esquema del DOM del siguiente código HTML se muestran las principales propiedades que tiene cada nodo para recorrer el DOM:

Si nos fijamos en el gráfico veremos algunas particularidades. La diferencia entre “firstChild” y “firstElementChild” reside en que con la segunda forma estamos especificando que queremos acceder a un elemento que sea del tipo “element”, es decir, que sea una marca. Esto es muy importante porque si nos fijamos en el código HTML, estrictamente el primer hijo que tiene Body es el salto de línea que hay entre las marcas <body> y <h1>, y en tal caso el firstChild de bodu será ése salto de línea. En el gráfico hemos obviado ésta posibilidad por claridad, pero no hay que olvidarla.

La propiedad “childNodes” retorna un array con todos los hijos, pero en el caso de “children” retorna un array solo con los hijos que sean “element”.

También es importante remarcar que los atributos no son considerados hijos de un nodo, sino que todos los nodos de tipo “element” tienen un atributo “atributtes” que contiene un mapa con todos los atributos. No es realmente una propiedad que vayamos a utilizar mucho porque veremos formas más simples de acceder a un atributo.

A continuación se especifican las propiedades vistas en el gráfico anterior. El ejemplo de uso se basará en el HTML anterior:

propiedad	Uso	Descripción

VERSIÓN IMPRIMIBLE ALUMNO LINKAEP

firstElementChild	let b = document.body; let h1 = b.**firstElementChild**;	Retorna el primer hijo del tipo "element".
lastElementChild	let b = document.body; let link = b.**lastElementChild**;	Retorna el último hijo del tipo "element".

firstChild	let b = document.body;
	let h1 = b.firstElementChild;
	let elDom = h1.**firstChild**;
		Retorna el primer hijo.

nextSibling	let b = document.body;
	let h1 = b.firstElementChild;
	let contenido= h1.**nextSibling**;
		Retorna su siguiente hermano.

previousSibling	let b = document.body;
	let link = b.lastElementChild;
	
	let contenido = link.**previousSibling**;
		Retorna su anterior hermano.

nextElementSibling	let b = document.body;
	let h1 = b.firstElementChild;
	let link = h1.**nextElementSibling**;
		Retorna su siguiente hermano del tipo "element".

parentNode	let b = document.body;
	let h1 = b.firstElementChild;
	b=h1.**parentNode**;
		Retorna su padre.

childNodes	let b = document.body; let nodosHijos = b.childNodes;	Retorna un array con todos los nodos hijos.
children	let b = document.body; let marcasHijas = b.children;	Retorna un array con todos los nodos hijos que sean "element".

VERSIÓN IMPRIMIBLE UMNC LINK AFP

En el siguiente video podemos ver cómo navegar a través del DOM utilizando el parentesco y modificar los estilos y propiedades de los elementos HTML.

4_2_5 Navegar por el DOM y modificar estilos

VERSIÓN IMPRIMIBLE ALUMNO

4.1. Crear y añadir nodos

Utilizando el API del DOM podemos crear nuevos nodos y vincularlos con otros nodos.

4.1.1. Crear un nodo de tipo elemento

Para crear un nuevo nodo de tipo "element" utilizaremos el método "createElement()" que nos ofrece "document". La sintaxis será la siguiente:

document.createElement("TipoDeElemento");

Ésta sentencia generará un nuevo "element" del tipo pasado por parámetro (DIV, SPAN, H1, etc..) y retornará una referencia a él. Pero no añadirá el "element" creado dentro de nuestro HTML. Esto lo tendremos que hacer a continuación.

Para añadir un "element" dentro de otro, primero tenemos que acceder al padre. A partir del padre utilizar su método "*apendChild()*" pasando por parámetro el objeto con la referencia al "element" que queremos sea su hijo. La sintaxis será la siguiente:

elementPadre.appendChild(elementHijo);

En el siguiente ejemplo creamos un H1 y P, modificamos los estilos y los añadimos:

HTML antes de ejecutar el JS

```
<head>
  <script src="dom.js" defer></script>
</head>
<body>     <div id="nuevos"></div>
</body>
```

BODY después de ejecutar el JS

```
<body>
  <div id="nuevos">
    <h1 style="color: blue;">Nuevo H1</h1>
    <p> texto del párrafo</p>
</div>
</body>
```

Documento JavaScript dom.js

```
  //creamos el nuevo elemento
let elemH1 = document.createElement("H1");
  //accedemos al futuro padre
let nuevos= document.getElementById("nuevos");
  //le añadimos el nuevo elemento
nuevos.appendChild(elemH1);
  //seguimos modificando sus estilos
elemH1.innerHTML="Nuevo H1";
elemH1.style.color="blue";

  //creamos otro elemento
let elemP =document.createElement("P");
elemP.innerHTML=" texto del párrafo";
  //y lo añadimos
nuevos.appendChild(elemP);
```

A partir del ejemplo anterior podemos observar cómo podemos modificar indistintamente las propiedades de un objeto antes o después de añadirlo al HTML, aunque la recomendación sería establecer todas sus propiedades ates para evitar comportamientos extraños en su visualización.

También podemos observar como al añadir un elemento con "*appendChild()*" éste nuevo elemento es añadido como último elemento dentro de su padre. Lo podemos ver

VERSIÓN NO LINKIAFP

claramente si observamos como el nodo P se ha añadido después de H1. Si quisiéramos añadir un elemento ántes que otro deberíamos utilizar la función *"insertBefore(x,y)"* . Para utilizar ésta función necesitamos saber 3 cosas: el nodo padre, el nuevo nodo y el nodo existente delante del que queramos insertar el nuevo nodo. Su sintaxis es:

nodoPadre.insertBefore(nuevoNodo, nodoExistente);

Cabe destacar que si "*nuevoNodo*" ya está situado en algún sitio del DOM será movido hasta su nueva posición delante de "*nodoExistente*".

En el siguiente ejemplo creamos un LI y lo añadimos entre los dos existentes:

HTML antes de ejecutar el JS

```
<head>
   <script src="dom.js" defer></script>
</head>
<body>
   <ol id="lista">
      <li>Item1</li>
      <li id="item">Item2</li>
   </ol>
</body>
```

Documento JavaScript dom.js

```
//creamos el nuevo elemento
let newLI = document.createElement("LI");
newLI.innerHTML="Nuevo";

//accedemos al futuro padre
let ol = document.getElementById("lista");
//accedemos al futuro hermano
let li = document.getElementById("item");

//añadimos el nuevoitem delante su hermano
ol.insertBefore(newLI,li);
```

BODY después de ejecutar el JS

```
<body>
   <ol id="lista">
      <li>Item1</li>
      <li>Nuevo</li>
      <li id="item">Item2</li>
   </ol>
</body>
```

VERSIÓN IMPRESA LINKIAFP

Como resumen los métodos principales para crear e insertar un nodo son:

método	Uso	Descripción
createElement	let n = document.createElement("H1");	Crea un nuevo "element" del tipo indicado y lo retorna. No lo añade al HTML.
appendChild	nodoPadre.appendChild(nodoNuevo);	Añade como hijo del nodo padre el nuevo nodo. Si el nuevo nodo ya existe entonces lo mueve.
insertBefore	padre.insertBefore(nuevo, hermano);	Inserta el nuevo nodo dentro del nodo padre y delante del nodo hermano. Si el nuevo nodo ya existe lo mueve de sitio.

4.2. Eliminar nodos

Cuando queramos eliminar un nodo siempre necesitaremos tener una referencia del elemento que queremos eliminar y de su padre. A partir del padre llamaremos a su método *removeChild()* pasándole como parámetro la referencia al nodo que queramos eliminar. Su sintaxis es:

nodoPadre.removeChild(nodoHijoAEliminar);

También podemos eliminar un elemento sustituyéndolo por otro. En éste caso necesitaremos también la referencia al nuevo nodo que va a sustituir el antiguo y a partir del elemento padre llamaremos a su método *replaceChild()* pasando por parámetro el nuevo nodo y el nodo que queremos eliminar para ser reemplazado por el nuevo. Su sintaxis es:

nodoPadre.replaceChild(nuevoNodo, nodoHijoAEliminar);

Si el *nuevoNodo* está situado en otro punto del HTML, entonces será movido.

VERSIÓN IMPRIMIBLE ALUMNO

En el siguiente ejemplo eliminamos el Item2 y substituimos el Item4 por otro LI con el texto “Nuevo Item”:

HTML antes de ejecutar el JS

```
<head>
   <script src="dom.js" defer></script>
</head>
<body>
   <ol >
      <li>Item1</li>
      <li id="item2">Item2</li>
      <li>Item3</li>
      <li id="item4">Item4</li>
   </ol>
</body>
```

Documento JavaScript dom.js

```
//accedemos al elemento a eliminar
let li2 = document.getElementById("item2");

//Accedemos a su padre
let padre = li2.parentNode;

//borramos el elemento del padre
padre.removeChild(li2);

//accedemos al elemento a eliminar
let li4 = document.getElementById("item4");
let newLi = document.createElement("LI");
newLi.innerHTML="Nuevo Item";

//sustituimos li4 por newLi
padre.replaceChild( newLi , li4 );
```

BODY después de ejecutar el JS

```
<body>
 <ol >
    <li>Item1</li>
    <li>Item3</li>
    <li >Nuevo Item</li>
  </ol>
</body>
```

Los métodos principales para eliminar un nodo son:

método	Uso	Descripción
removeChild	padre.removeChild(nodoAEliminar);	Elimina el *nodoAEliminar* del nodo *padre*.
replaceChild	padre. replaceChild (nuevo, antiguo);	Elimina el nodo *antiguo* del nodo *padre* y lo sustituye por el nodo *nuevo.*

VERSIÓN IMPRIMIBLE

4.3. Gestionar atributos propios

Todos los nodos HTML pueden tener atributos. Muchos de ellos son atributos “nativos”, es decir, ya especificados en el estándar HTML. Éstos atributos no los deberemos crear ni eliminar nosotros. Incluso si en el código HTML hay un elemento que no tiene un atributo “nativo”, el DOM sí que lo tendrá especificado y podremos acceder a él a través de su propiedad relacionada. Por ejemplo, si queremos modificar el atributo id de un elemento (aunque inicialmente éste no lo tenga) solo deberemos asignar a la propiedad id del objeto que referencie el elemento un nuevo valor.

Pero en muchas ocasiones nos puede resultar útil crear atributos própios que no existan en la especificación de HTML. Será solo en ésta ocasión cuando utilizaremos los siguientes métodos para crear , modificar o eliminar nuestro atributo.

El DOM nos ofrece muchas formas para leer los atributos própios de un nodo “element”. Y también muchas formas para modificar sus valores y crear nuevos atributos. A continuación veremos las formas más prácticas para tales efectos.

Cuando queramos leer o modificar un atributo siempre necesitaremos tener una referencia del elemento que lo tenga o lo vaya a tener.

Para leer el valor de un atributo llamaremos al método *getAttribute()* del nodo que lo contiene pasándole como parámetro el nombre del atributo que queremos leer utilizando la sintaxis

nodoElement.getAttribute(“nombreAtributo”);

Tambien podemos utilizar el método *setAttribute()* para modificar el contenido de un atributo o crear un nuevo atributo si éste ya no existía. La sintaxis utilizada es:

nodoElement.setAttribute(“nombreAtributo”,”valor del atributo”);

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Hemos de tener presente que se han de cumplir las reglas HTML al establecer el nombre de un atributo. En el siguiente ejemplo accedemos a un atributo “contar”, mostramos y alteamos su valor y finalmente creamos un nuevo atributo “restar” con valor “2”:

HTML antes de ejecutar el JS

```
<head>
   <script src="dom.js" defer></script>
</head>
<body>
   <p id="p" contar="4">
      el atributo contar no existe en la
especificación HTML  </p>
</body>
```

BODY después de ejecutar el JS

```
<body>
    <p id="p" contar="5" restar="2">       el
atributo contar no existe en la especificación
HTML   </p>
</body>
```

Documento JavaScript dom.js

```
let li2 = document.getElementById("p");

//obtenemos el valor del atributo contar
let valorContar = li2.getAttribute("contar");
console.log(valorContar); //4

//modifiamos el valor de contar
li2.setAttribute("contar",5);
valorContar = li2.getAttribute("contar");
console.log(valorContar); //5

//añadimos un nuevo atributo restar="2"
li2.setAttribute("restar",2);
valorContar = li2.getAttribute("restar");
console.log(valorContar); //2
```

Los métodos principales para gestionar los atributos son:

método	Uso	Descripción
getAttribute	nodo.getAttribute (nombreAtributo);	Retorna el valor del atributo *nombreAtributo* contenido en el *nodo*
setAttribute	nodo.setAttribute (nombreAt, valor);	Crea un atributo de nombre *nombreAt* dentro del *nodo*. Si ya existe modifica su valor.
removeAttribute	nodo.removeAttribute (nombreAt);	Elimina un atributo de nombre *nombreAt* del *nodo*.

VERSIÓN IMPRIMIBLE

Test de autoevaluación

¿Qué tipo de nodos pueden tener hijos?

a) Todos
b) Element
c) Ninguno
d) Text

¿Cuál es la sentencia recomendada para modificar el atributo “id” de un elemento con “id” “pelota”?.

e) document.getElementByd(“pelota”).id=”nuevo valor”;
f) document.getElementByd(“pelota”).setAttribute (“id”, “nuevo valor”);
g) document.getElementsByTagName(“pelota”).id=”nuevo valor”;
h) document.getElementsByTagName(“pelota”).setAttribute (“id”, “nuevo valor”);

¿Cómo accedemos al elemento anterior del elemento con “id” “pelota”?

i) document.getElementByd(“pelota”).previousElementSibling();
j) document.getElementByd(“pelota”).previousSibling();
k) document.getElementByd(“pelota”).previousSibling;
l) document.getElementByd(“pelota”).previousElementSibling;

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Recursos y enlaces

- DOM Enlightenment: versión online y gratuita de un libro centrado en el estudio del DOM. Escrito por Cody Lindley http://domenlightenment.com

- Documentación sobre el DOM en MDN: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

- X3Dom : librería JavaScript para insertar contenido 3D utilizando HTM, CSS y JS. https://www.x3dom.org

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Conceptos clave

- **DOM** (Document Object Model): es el resultado de estructurar en objetos el contenido de un documento HTML.
- Modificando el DOM con JavaScript podemos modificar el documento HTML.
- **Nodo HTML:** un nodo HTML es cualquier texto, marca, atributo o comentario que podamos encontrar en el código HTML.
- **Elemento HTML**: es el conjunto de una etiqueta inicial, su contenido y la etiqueta final.

Ponlo en práctica

Actividad 1

Crea un documento HTML con una lista ordenada de 7 *items*. Programa que:

1. al clicar en el primero se cambie su color e texto a verde
2. al clicar en el segundo se cambie su color de fondo a amarillo
3. al clicar en el tercero cambie el último elemento LI por el primero
4. al clicar en el cuarto se cambie el color de fondo de todos a azul
5. al clicar en el quinto, se borre a él.
6. al clicar en el sexto, se borre el *element* hermano anterior
7. al clicar en el séptimo, se añada otro LI con texto "NUEVO LI" y color de fondo rojo

Los solucionarios están disponibles en el aula virtual.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Tema 5: Gestión de eventos y formularios

¿Qué aprenderás?

- Diferenciar los distintos tipos de eventos que se pueden producir en un documento web.
- Capturar los eventos que el usuario realice sobre la web.
- Diferenciar los distintos tipos de validaciones que se pueden realizar en un formulario web.
- Controlar el envío de los formularios con JavaScript.
- Validar formularios web con eventos y expresiones regulares.
- Crear, modificar y borrar cookies.

¿Sabías que…?

- La sentencia document.forms nos retorna a todos los formularios del documento.
- La función eval() permite ejecutar cualquier texto como si fuera un código JavaScript (incluso un texto escrito por el usuario!).
- JavaScript fue diseñado en 10días.
- JavaScript es uno de los lenguajes de programación más demandados hoy en día según Stack Overflow.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

5. Gestión de eventos y formularios

5.1. Gestión de eventos

JavaScript nos ofrece la posibilidad de capturar aquellos momentos en los que se produce una interacción o cambio en el estado del DOM u otros objetos JavaScript. Éstas interacciones y cambios de estado es lo que entenderemos como evento.

Una vez capturemos un evento tendremos acceso a un objeto con información sobre ése tipo de evento. Si por ejemplo capturamos un evento de teclado podremos saber qué letra se ha escrito, o si capturamos un evento de ratón podremos conocer donde está situado el ratón.

5.1.1. Capturar un evento

A continuación vamos a ver cómo podemos capturar un evento para que al ejecutarse se lance una función personalizada. JavaScript nos ofrece tres formas de vincular un evento a un elemento.

Todos los elementos HTML tienen un atributo que va asociado a un tipo de evento y que puede contener un código JavaScript que se ejecutará cuando se ejecute dicho evento sobre el elemento. Aprovechando ésta característica la primera forma de vincular un evento a un elemento consiste en modificar ése atributo indicándole el código que queramos ejecutar.

En el siguiente ejemplo modificamos el atributo “onclick” que va vinculado al evento que se lanza cuando se clica sobre un elemento. De ésta forma cuando se clique encima el div con el texto PRESS se realizará la operación matemática y mostrará el resultado por consola.

```
<div onclick="let resul=5+6; console.log(resul);"> PRESS </div>
```

Como no es una buena idea escribir el código directamente en el HTML podemos crear una función en JavaScript y modificar el atributo para que al clicar encima de PRESS se ejecuta dicha función dos veces. A continuación vemos un ejemplo:

```
<head>  <script>
            function sumar(){
                let resul = 5+6;
                console.log(resul);
            }
  </script> </head>
<body>
   <div onclick="sumar();sumar();"> PRESS </div>
</body>
```

VER LIBRE ALUMNO LINKIAFP

Ésta manera es la menos recomendables porque mezcla la programación con la estructuración del código HTML y además es completamente estático ya que no lo generamos desde JavaScript.

La segunda manera consiste en acceder mediante el DOM al elemento al que queramos añadir el evento y modificar la propiedad asociada al atributo HTML asignándole la función que queramos ejecutar.

En el siguiente ejemplo accedemos al DIV a través de su atributo "id" y modificamos su atributo "onclick" para que ejecute la función "sumar" mostrando por consola el valor 11 cada vez que se clique sobre PRESS.

Documento HTML

```
<html>
<head>
   <!-- Imporante usar defer!-->
   <script src="dom.js" defer></script>
</head>
<body>
   <div id="suma"> PRESS </div>
</body>
</html>
```

Documento JavaScript dom.js

```
let elemDiv= document.getElementById("suma");
//modificamos el atributo onclick asignandole la
funcion a ejecutar
elemDiv.onclick=sumar;

//definimos la función a ejecutar
function sumar(){
   let resul = 5+6;
   console.log(resul);
}
```

Éste método conlleva el problema de que no se pueden asignar dos funciones a un mismo evento. Si le asignamos otra función estaremos sobrescribiendo el valor anterior.

La última manera de asignar eventos es la más recomendable y utilizada ya que nos permite añadir eventos a través de JavaScript. Consiste en acceder mediante el DOM al elemento sobre el que queramos añadir el evento y utilizar el método "addEventListener()" que tienen todos los objetos que hacen referencia a un elemento del DOM.

La sintaxis del "addEventListener()" es la siguiente:

nodoElement.addEventListener("nombreEventoSinOn",funcionAsociada,false);

Vamos a ver a continuación las distintas partes que forman la sintaxis anterior:

- *nodoElement:* se va a corresponder con el objeto que haga referencia al element al que queramos capturar el evento.
- *addEventListener:* es el nombre de la función. (siempre será éste)
- "nombreEventoSinOn": es el nombre del evento que queremos capturar quitándole el prefijo "on-". Por ejemplo, para capturar un evento "onclick" éste nombre será "click".
- *funcionAsociada*: es la función que queremos que se ejecute cuando se lanza el evento "nombreEventoSinOn" sobre el elemento "nodoElement".

VERSIÓN IMPRIMIBLE NKIAFP

- *false*: éste valor indica si queremos que se lance el evento en la fase de "captura" o por lo contrario en la fase de "bubbling". Esto lo veremos más adelante en el apartado "Propagación del Evento", pero normalmente se suele utilizar el valor *false* (fase de "bubbling");

En el siguiente ejemplo utilizamos el "addEventListener" para vincular una función sumar y otra función sumar al evento onclick del div con id "suma".

Documento HTML

```
<html>
<head>
   <!-- Imporante usar defer!-->
   <script src="dom.js" defer></script>
</head>
<body>
   <div id="suma"> PRESS </div>
</body>
</html>
```

Documento JavaScript dom.js

```
let elemDiv= document.getElementById("suma");
//vinculamos la función sumar al evento click
elemDiv.addEventListener("click",sumar,false);
//añadimos la función restar al evento click
elemDiv.addEventListener("click",restar,false);
//definimos la función suma
function sumar(){
   let resul = 5+6;
   console.log(resul);
}
//definimos otra función resta
function restar(){
   let resul = 5-6;
   console.log(resul);
}
```

5.1.1.1. Objeto Event

Cuando un evento lanza una función, ésa función recibirá siempre como parámetro un objeto del tipo Event y más concretamente del evento que se haya lanzado. Ése objeto contendrá en su propiedades toda la información relativa al evento.

En el siguiente ejemplo se utiliza ésta característica para conocer qué botón ha apretado el usuario cada vez que clica sobre el BODY.

Documento HTML

```
<html>
<head>
   <!-- Imporante usar defer!-->
   <script src="dom.js" defer></script>
</head>
<body>
   <div id="suma"> PRESS </div>
</body>
</html>
```

Documento JavaScript dom.js

```
let body = document.body;
body.addEventListener("mousedown",capturaRaton,false);
function capturaRaton(evt){
   //si evt es un objeto del tipo MouseEvent podemos
conocer el boton clicado con la propiedad .button
   let botonClicado = evt.button;
   //obtendremos un 0 para el boton izquierdo, un 1 para
el boton central y un 2 para el boton derecho
   console.log("boton clicado:"+botonClicado);
}
```

En éste ejemplo hemos utilizado la propiedad *button* porque sabemos que el evento es del tipo MouseEvent. Más a delante veremos un listado de los principales tipos de evento disponibles y sus propiedades más destacadas.

5.1.2. Propagación del evento

Cuando se produce un evento sobre un element en muchas ocasiones se produce también sobre sus padres o hijos. Por ejemplo, si clicamos sobre un DIV también estaremos clicando sobre el BODY que irremediablemente contiene el DIV. Y si el DIV y el BODY tuvieran ambos asociados un evento onclick, se ejecutarían ambos? Y en caso afirmativo, cuál se ejecutaría antes? Todo éste comportamiento es lo que entenderemos como "la propagación del evento".

En JavaScript cuando se produce un evento éste evento se propaga dos veces por todos los elementos a los que afecte a través del árbol del DOM. Inicialmente, en la llamada "fase de captura" se propaga desde el nodo raíz "document" hasta el nodo más alejado al que afecte. Y posteriormente en la llamada "fase de bubbling" se propaga desde el nodo más alejado afectado por el evento hasta document.

Cuando nosotros vinculamos un evento con "addEventListener" podemos indicarle 3 parámetros, el 3º contendrá un valor "true" para ejecutar la función en la "fase de captura" y un valor "fase" para ejecutar la función en la "fase de bubbling".

En el siguiente ejemplo hemos añadido tres eventos "onclick" que van a ser lanzados cada vez que el usuario clique sobre el H1. Al ser añadidos con el parámetro "true" los hemos vinculado en la "fase de captura". Por ello el orden de ejecución será: BODY, DIV y H1 mostrando por consola los valores:

click Body Capture!

click Div Capture!

click H1 Capture!

Documento HTML

```html
<html>
<head>
  <!-- Imporante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <div id="elDiv">
    Propagacion
    <h1 id="elH1">Event</h1>
  </div>
</body>
</html>
```

Documento JavaScript dom.js

```javascript
let body = document.body;
let div=document.getElementById("elDiv");
let h1=document.getElementById("elH1");
//vinculamos los eventos en la "fase de captura"
body.addEventListener("click",clickBody,true);
div.addEventListener("click",clickDiv,true);
h1.addEventListener("click",clickH1,true);

function clickBody(){
  console.log("click Body Capture!");
}
function clickDiv(){
  console.log("click Div Capture!");
}
function clickH1(){
  console.log("click H1 Capture!");
}
```

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

En el siguiente ejemplo hemos añadido tres eventos “onclick” que van a ser lanzados cada vez que el usuario clique sobre el H1 pero ésta vez son añadidos con el parámetro “false” y por lo tanto los vincularemos en la “fase de bubbling”. Por ello el orden de ejecución será: H1, DIV, BODY mostrando por consola los valores:

click H1 Capture!

click Div Capture!

click Body Capture!

Documento HTML

```
<html>
<head>
  <!-- Imporante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <div id="elDiv">
    Propagacion
    <h1 id="elH1">Event</h1>
  </div>
</body>
</html>
```

Documento JavaScript dom.js

```
let body = document.body;
let div=document.getElementById("elDiv");
let h1=document.getElementById("elH1");

//vinculamos los eventos en la "fase de captura"
body.addEventListener("click",clickBody, false);
div.addEventListener("click",clickDiv, false);
h1.addEventListener("click",clickH1, false);

function clickBody(){
  console.log("click Body Capture!");
}
function clickDiv(){
  console.log("click Div Capture!");
}
function clickH1(){
  console.log("click H1 Capture!");
}
```

Como normalmente se suele dar preferencia al elemento más alejado del “document”, lo más común es capturar los eventos en la “fase de bubbling”.

5.1.2.1. Detener el evento

Cuando se ejecuta una función lanzada por un evento, podemos utilizar el objeto “Event” que recibe como parámetro para detener la propagación del evento. De ésta forma evitaremos que se ejecuten el resto de funciones vinculadas con el evento que aún no se hayan ejecutado.

Para hacerlo utilizaremos el método “stopPropagation();” implementado en el mismo objeto “Event”.

VERSIÓN IMPRIMIBLE AFP

En el siguiente ejemplo si clicamos encima del DIV o del H1 no se ejecutará la función "clickBody()" ya que hemos detenido la propagación del evento y las funciones se han vinculado en la fase de Bubbling.

Documento HTML

```html
<html>
<head>
  <!-- Imporante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
  <div id="elDiv">
    Propagacion
    <h1 id="elH1">Event</h1>
  </div>
</body>
</html>
```

Documento JavaScript dom.js

```javascript
let body = document.body;
let div=document.getElementById("elDiv");
let h1=document.getElementById("elH1");

//vinculamos los eventos en la "fase de captura"
body.addEventListener("click",clickBody, false);
div.addEventListener("click",clickDiv, false);
h1.addEventListener("click",clickH1, false);

function clickBody(){   console.log("click Body Capture!");
}
function clickDiv(){
  //detenemos la propagación del evento
  evt.stopPropagation();
  console.log("click Div Capture!");
}
function clickH1(){   console.log("click H1 Capture!");
}
```

En muchas ocasiones hay eventos que tienen acciones predeterminadas en el navegador. La más típica es el click derecho del ratón que nos abre un menú contextual. Desde JavaScript también podemos evitar las acciones por defecto utilizando el método "preventDefault()" implementado en el mismo "Event".

A continuación se muestra un ejemplo en el que evitamos que se muestre el menú contextual con la función "preventDefault()":

Documento HTML

```html
<html>
<head>
  <!-- Imporante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
    <p>Evita el menú
        contextual</p>
</body>
</html>
```

Documento JavaScript dom.js

```javascript
let body = document.body;
//contextmenu es el evento que se lanza al mostrar el
menu contextual
body.addEventListener("contextmenu",evitaMenu,true);

function evitaMenu(evt){
  //preventDefault evita las acciones por defecto del
navegador
  evt.preventDefault();
}
```

VERSIÓN IMPRIMIBLE

5.1.3. Tipos de eventos

Como ya hemos visto, cuando capturamos un evento recibimos por parámetro información sobre ése event. Cada tipo de evento contiene unas propiedades con distinta información sobre el evento. Todos los evento heredan de "Event" y por lo tanto tienen como mínimo sus mismas propiedades y métodos, A continuación se muestran los tipos principales de eventos en JS y sus propiedades más destacadas:

Tipos Eventos	Descripción	Propiedades destacadas
Event	Es el tipo de evento más general. Cuando un evento no pertenezca a ningún otro tipo, pertenecerá a Event. Todos los eventos son del tipo Event y heredan las propiedades de Event.	target: retorna el elemento que ha lanzado el evento. currentTarget: retorna el elemento al que se ha vinculado la función. type: retorna el tipo de evento
UiEvent	Gestionan la información derivada de cambios en la interfaz del usuario, como puede ser el redimensionar la pantalla. Hereda de Event.	view: retorna una referencia a al objeto window que ha lanzado el evento.
TouchEvent	Si el dispositivo lo acepta, gestionan la información cuando el usuario toca la pantalla. Hereda de UiEvent.	altKey , shiftKey, ctrlKey : retorna si la tecla "ALT", "SHIFT" o "CONTROL" estaba pulsada. touches: retorna cuantos dedos están tocando la pantalla.

VERSIÓN IMPRIMIBLE ALUMNO CIFP

FocusEvent	Contienen información sobre los cambios del foco entre los distintos elementos de la web. Hereda de UiEvent.	relatedTarget: retorna el elemento que ha perdido o adquirido el foco.
InputEvent	Contienen información sobre los cambios e interacciones con los inputs del HTML. Hereda de UiEvent.	data: retorna los caracteres insertados. inputType: información sobre el tipo de operación realizada (insertado texto, borrado, etc.)
KeyboardEvent	Contienen información sobre la interacción del usuario con el teclado. Hereda de UiEvent.	altKey , shiftKey, ctrlKey : retorna si la tecla “ALT”, “SHIFT” o “CONTROL” estaba pulsada. charCode: retorna el carácter pulsado. keyCode, code, key: retornan un código identificativo de la tecla pulsada. repeat: retorna si se está manteniendo la tecla pulsada.
StorageEvent	Gestiona los cambios en el “window storage”	key: retorna la clave del ítem que se ha modificado. newValue, oldValue: retorna el antiguo o nuevo valor del ítem modificado

VERSIÓN IMPRIMIBLE ALUMNADO IES

MouseEvent	Gestionan las acciones del ratón y su movimiento. Hereda de UiEvent.	altKey , shiftKey, ctrlKey : retorna si la tecla “ALT”, “SHIFT” o “CONTROL” estaba pulsada. button , buttons : retorna un número identificativo de qué botón o qué botones se han pulsado (0 izquierdo, 1 medio, 2 derecho) clientX, clientY: retorna la coordenada horizontal y vertical del ratón respecto al navegador. offsetX, offsetY: retorna la coordenada horizontal y vertical del ratón respecto elemento que lanza el evento. pageX, pageY: retorna la coordenada horizontal y vertical del ratón respecto al documento. movementX, movementY: retorna el desplazamiento horizontal y vertical del raton.
DragEvent	Gestionan las acciones de arrastrar contenido por la web. Hereda de MouseEvent.	dataTransfer: contiene la información que se está moviendo
ClipboardEvent	Gestionan la información cuando el usuario corta/copia/pega información.	clipboardData: contiene la información copiada.

VERSIÓN IMPRIMIBLE ALUMNO - IKAEP

PopStateEvent	Gestionan la información sobre los cambios en el historial.	state: retorna un objeto con una copia del historial
ProgressEvent	Gestionan la información del progreso de carga de recursos externos. Hereda de UiEvent.	loaded: retorna la cantidad de contenido cargado. total: retorna la cantidad de contenido que se ha de cargar
AnimationEvent	Contienen información sobre los cambios de estado en las animaciones CSS.	animationName: retorna el nombre de la animación elapsedTime: retorna los segundos que lleva ejecutándose la animación.
TransitionEvent	Gestionan la información cuando se producen cambios de estado en las transiciones CSS.	animationName: retorna el nombre de la transición. elapsedTime: retorna los segundos que lleva ejecutándose la transición.
HashChangeEvent	Contienen información sobre los cambios que se produzcan en el hash de la URL.	newURL: retorna la URL nueva oldURL: retorna la URL antes de ser cambiada
WheelEvent	Gestionan la información cuando el usuario mueve la ruedecita del ratón. Hereda de UiEvent y MouseEvent	deltaX, deltaY, deltaZ retorna la cantidad de scroll que se ha producido en el eje de las X, Y y Z.

Cada tipo de evento clasifica a varios eventos. A continuación vamos a ver los principales eventos que utilizaremos para desarrollar aplicaciones web agrupadas según su funcionalidad.

VERSIÓN IMPRIMIBLE ALUMNADO

5.1.4. Eventos generales

Eventos que generan un objeto del tipo Event

evento	Descripción
load	se lanza cuando el objeto es cargado completamente. Normalmente se usa con el window para ejecutar un código una vez se ha cargado toda la página: window.addEventListener("load", myScript, false);
contextmenu	se lanza cuando el click derecho del ratón provoca que se habrá un menú contextual.
copy/cut	Se lanza cuando se copia o pega contenido. Generan un objeto del tipo ClipboardEvent.
scroll	Se lanza cuando se produce un scroll sobre el elemento vinculado. Genera un objeto del tipo UiEvent.
resize	Éste evento se debe vincular con window, y se lanza cuando la ventana es redimensionada. Genera un objeto del tipo UiEvent. Por ejemplo: window.addEventListener("resize",myScript,false);

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

5.1.5. Eventos de ratón

A continuación se muestran el listado de eventos que generan un objeto MouseEvent:

evento	Descripción
click	Evento que se ejecuta cuando se clica sobre un elemento del DOM.
dblclick	Evento que se ejecuta cuando se clica dos veces sobre un elemento del DOM.
mousedown	Evento que se ejecuta cuando se presiona un botón del ratón sobre un elemento del DOM.
mousemove	Evento que se ejecuta cuando se mueve el ratón por encima un elemento del DOM.
mouseover	Evento que se ejecuta cuando el ratón pasa a estar sobre un elemento del DOM o uno de sus hijos.
mouseout	Evento que se ejecuta cuando el ratón sale de un elemento del DOM o uno de sus hijos.
mouseup	Evento que se ejecuta cuando se deja de presionar un botón del ratón sobre un elemento del DOM.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

5.1.6. Eventos del teclado

Los siguientes eventos capturan acciones sobre el teclado y generan un objeto del tipo KeyboardEvent

evento	Descripción
keydown	Evento que se ejecuta cuando se está pulsando una tecla.
keypress	Evento que se ejecuta mientras se está pulsando una tecla excepto Shift, Fn, CapsLock.
keyup	Evento que se ejecuta cuando se deja de pulsar una tecla.

5.1.7. Eventos de formularios

A continuación se muestran los eventos que típicamente utilizaremos al tratar con formularios:

evento	Descripción
submit	Evento que se ha de vincular a un formulario y se ejecuta cuando se va a enviar el formulario.
reset	Evento que se ejecuta antes de realizar un *reset* en el formulario para borrar todos sus datos.
change	Evento que se ejecuta cuando el usuario cambia la selección de un elemento SELECT, CHECKBOX, RADIO
input	Evento que se ejecuta justo antes de que el usuario cambie el valor de un input.
select	Se ejecuta después de que el usuario seleccione texto en un input o un textarea.
blur	Se ejecuta después de que elemento pierda el foco.

VERSIÓN IMPRESA ALUMNO LINKIAFP

focus	Se ejecuta después de que elemento gane el foco.
invalid	Evento que se ejecuta cuando un elemento del formulario es inválido.

5.1.1. Eventos copy/paste

Eventos que generan un objeto del tipo ClipboardEvent

evento	Descripción
copy/cut	Se lanza cuando se copia o pega contenido.

5.2. Gestión de formularios

JavaScript ofrece una gran flexibilidad para la validación de formularios con el objetivo de hacer más agradable la interacción del usuario con la aplicación web. JavaScript ha de ser el encargado de que vayamos obteniendo un feed-back sobre lo correctos que son nuestros datos antes de enviar el formulario. Pero en ningún caso JavaScript debe utilizarse como una medida de seguridad ya que es posible enviar datos a nuestro servidor sin utilizar ninguna aplicación web. Por ello, hay que destacar que todas la medidas de seguridad deben de estar en el servidor.

Para poder gestionar los formularios a continuación veremos cómo acceder a los elementos de un formulario a través del DOM , como gestionar los principales eventos relacionados con los formularios y como realizar la validación de sus datos.

5.2.1. Acceso al DOM del formulario

JavaScript siempre nos ofrece múltiples caminos para realizar una misma tarea. En ésta ocasión veremos la forma más recomendable y específica de acceder a los inputs de un formulario, aunque hay que destacar que no es la única.

Como ya vimos, el objeto global “document” dispone de un atributo “forms” que nos retorna una “HTMLCollection” formada por todos los formularios de nuestra página web.

Una “HTMLCollection” la podemos entender como un array en donde todos los elementos están guardados en una posición numérica según el orden en el que aparecen en el DOM y

VERSIÓN IMPRIMIBLE ALUMNO EUKIAFP

que además pueden estar guardados según su atributo “id” o su “name”. Podremos recorrer una “HTMLCollection” con un bucle FOR o un bule ForEach.

Utilizando el objeto “document” y su propiedad “forms” accederemos a un formulario con la sintaxis:

```
document.forms[posicionFormulario];
document.forms["nameFormulario"];
document.forms["idFormulario"];
```

A continuación vamos a ver cómo podemos acceder a los distintos elementos que forman un formulario a partir de “document.forms[]”

5.2.1.1. Inputs

Acceder a través de “document.forms[]” nos retornará un objeto del tipo FormObject que a su vez será una colección HTML de los elementos que contiene el formulario. Así que si queremos acceder a través del DOM a un input en concreto podremos utilizar las sintaxis:

```
document.forms[posicionFormulario][posicionInput];
document.forms["nameFormulario"]["nombreInput"];
document.forms["idFormulario"]["idInput"];
```

Pudiendo combinar entre ellos los accesos al formulario mediante posición, nombre e id con los accesos a los *inputs* mediante posición, nombre e id. Por ejemplo, podríamos acceder al *input* de nombre “nombreInput” del primer formulario escribiendo:

```
document.forms[0]["nombreInput"];
```

O acceder al primer input del formulario con id “idFormulario” escribiendo:

```
document.forms["idFormulario"][0];
```

Una vez obtenemos del DOM un objeto que referencia un *input* podemos acceder fácilmente a sus atributos como una propiedad del objeto. De ésta forma vamos a poder leer o modificar todos los atributos de los elementos que forman un formulario.

Por ejemplo, podemos acceder el valor que contiene un input de nombre “apellido” de un formulario con id “altaUsuario” con la sintáxis:

```
document.forms["altaUsuario"]["apellido"].value
```

La propiedad *value* contendrá el valor actual del input que el usuario puede haber modificado escribiendo en él.

Si vamos a querer acceder a muchos *inputs* del formulario podemos optar por guardar la referencia al formulario en un objeto y acceder al resto de inputs a través de ése objeto. En

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

éste ejemplo guardamos la referencia al objeto en una variable “miForm” y a través de ella accedemos al valor del primer input y del input con nombre “apellido”

```
let miForm = document.forms["altaUsuario"];
miForm[0].value
miForm["apellido"].value
```

En el siguiente ejemplo vemos como acceder al valor de un input de tres formas distintas. En la primera accedemos directamente y guardamos el valor en una variable *txt*. En la segunda primero obtenemos una referencia al formulario que guardamos en *form* y luego accedemos al valor a partir de ésa referencia. En la tercera utilizamos la variable *form* para obtener una referencia al input *nombre* que guardamos en la variable *input* y a partir de ésta variable obtenemos el valor.

Documento HTML

```html
<html>
<head>
  <!-- Imporante usar defer!-->
  <script src="dom.js" defer></script>
</head>
<body>
 <form id="form0" name="miForm">
  <input name="nombre" value="Nil"/>
  <input name="edad" value="56"/>
 </form>
</body>
</html>
```

Documento JavaScript dom.js

```javascript
//acceso a un valor directamente
let txt = document.forms["miForm"]["nombre"].value;
console.log(txt);  //Nil

//acceso al valor desde un objeto DOM del formulario
let form = document.forms["miForm"];
txt = form["nombre"].value;
console.log(txt);//Nil

//acceso al valor desde un objeto DOM del input
let input = form["nombre"];
txt = input.value;
console.log(txt);//Nil
```

Los inputs son los elementos más simples que pueden componer nuestro formulario y el acceso mediante el nombre es directo. Pero, qué ocurre con los checkbox que tienen el mismo nombre? A continuación lo vamos a ver.

VERSIÓN IMPRIMIBLE PARA USO LINKIAFP

5.2.1.2. Checkbox

Los *checkbox* son un tipo de marca HTML que nos permiten marcar más de una opción en un formulario. En el siguiente código vemos un ejemplo de tres *checkbox* utilizados para saber qué quiere comer el usuario, y el usuario va a poder escoger de 0 a 3 comidas distintas.

```
<form id="form0" name="miForm">
  Todo lo que  quieras comer! <br />
  <input type="checkbox" name="comer" value="boata">Bocata <br />
  <input type="checkbox" name="comer" value="car" checked>Wok<br />
  <input type="checkbox" name="comer" value="sushi">Sushi<br />
```

Qué quieres comer?
- [ ] Bocata
- [x] Wok
- [ ] Sushi

Las principales características que nos interesan como programadores son:

1. Los *checkbox* marcados tienen el atributo *checked*. Cuando accedamos a un *checkbox* sabremos si el usuario lo ha marcado comprobando el atributo *checked*.
2. El *value* no lo escribe el usuario sino que está prefijado por el desarrollador HTML.
3. Todos los *checkbox* tienen el mismo nombre. Cuando accedamos a un *checkbox* por su *name* nos va a retornar un *HTMLCollection* con los *checkbox* seleccionados.

Conociendo éstas características vamos a ver un ejemplo en el que inicialmente contemos cuantos *checkbox* están marcados en un formulario.

Documento HTML

```
<form id="form0" name="miForm">
  Todo lo que  quieras comer! <br />
  <input type="checkbox"
     name="comer" value="bocata">Bocata

  <input type="checkbox"
    name="comer" value="wok" checked>Wok

<input type="checkbox"
    name="comer" value="sushi">Sushi

  <input type="checkbox"
 name="comer" value="crepe" checked>Crepe

</form>
```

Documento JavaScript

```
let totalCheckbox=0;
let totalMarcados=0;
let form = document.forms["miForm"];
//obtenemos los checkbox de nombre comer
let checks= form["comer"];
// usar un for para recorrer el HTMLCollection
for(let k=0; k < checks.length;k++){
  totalCheckbox++;
  if(checks[k].checked==true){
    totalMarcados++;
  }
}
console.log("Total Checkboxs: "
                +totalCheckbox); //4
console.log("Total marcados: "
                +totalMarcados); //2
```

VER ALUMNO LINKIA

En éste ejemplo hemos utilizado un bucle *FOR* para recorrer el array. Vamos a ver cómo podríamos recorrer el array con un bucle FOREACH, pero en éste caso en vez de contar mostraremos por consola aquellos valores marcados.

Documento HTML

```
<form id="form0" name="miForm">
   Todo lo que  quieras comer! <br />
   <input type="checkbox"
      name="comer" value="bocata">Bocata

   <input type="checkbox"
     name="comer" value="wok" checked>Wok

<input type="checkbox"
     name="comer" value="sushi">Sushi

   <input type="checkbox"
  name="comer" value="crepe" checked>Crepe
</form>
```

Documento JavaScript

```
let totalCheckbox=0;
let totalMarcados=0;
let form = document.forms["miForm"];
//obtenemos los checkbox de nombre comer
let checks= form["comer"];
//recorrer el HTMLCollectioncon forEach
totalCheckbox=0;
checks.forEach(function(inputComida,key){
// inputComida referencía a cada input
   console.log(
inputComida.value+"-"+inputComida.checked);
});
```

Como podemos observar en el ejemplo anterior cada vez que iteramos sobre el array obtenemos una referencia a un input. A partir de ésta referencia podemos acceder a entre otras a su propiedad “value” para obtener su valor o “checked” para saber si está marcado. En ésta caso nos mostraría por consola el mensaje:

bocata-false
wok -true
sushi-false
tortilla-true

En éste caso no es recomendable utilizar el bucle FOR/LET, porque nos retornaría todas las propiedades y todos los elementos, ya que el bucle FOR/LET recorre tanto las posiciones numéricas como las no numéricas.

5.2.1.3. Radio

Los *radio* son un tipo de marca HTML que nos permiten marcar una única opción en un formulario. En el siguiente código vemos un ejemplo de cuatro *radio* utilizados para saber qué quiere comer el usuario pero en éste caso el usuario solo va a poder escoger una comida:

```
<form id="form0" name="miForm">
   Una única opción para comer <br />
   <input type="radio" name="comer" value="bocata">Bocata <br />
   <input type="radio" name="comer" value="wok" checked>Wok<br />
   <input type="radio" name="comer" value="sushi">Sushi<br />
   <input type="radio" name="comer" value="tortilla">Tortilla<br />
```

Qué quieres comer?
- Bocata
- Wok
- Sushi
- Tortilla

VERSIÓN IMPRIMIBLE ALUM

Las principales características que nos interesan como programadores son:

1. El *radio* seleccionado tienen el atributo *checked*. Cuando accedamos a un *radio* sabremos si el usuario lo ha seleccionado comprobando el atributo *checked*.
2. El *value* no lo escribe el usuario sino que está prefijado por el desarrollador HTML.
3. Todos los *radio* tienen el mismo nombre. Cuando accedamos a un *radio* por su *name* nos va a retornar un *HTMLCollection* con los *radio* seleccionados.

Conociendo éstas características vamos a ver un ejemplo en el que obtengamos qué valor ha marcado el usuario o un mensaje de alerta si no hay nada marcado nada. Cuando ejecutemos éste código nos mostrará el mensaje:

Para comer: Wok

Si quitáramos el atributo *checked* nos mostraría el mensaje:

No hay nada preseleccionado

Documento HTML

```
<form id="form0" name="miForm">
   Qué quieres comer? <br />
   <input type="radio"
name="comer" value="bocata">Bocata <br />
   <input type="radio"
name="comer" value="wok" checked>Wok<br />
   <input type="radio"
name="comer" value="sushi">Sushi<br />
   <input type="radio"
 name="comer" value="tortilla">Tortilla<br />
</form>
```

Documento JavaScript

```
let algunRadioChecked=false;
let form = document.forms["miForm"];
//obtenemos todos los radio de nombre comer
let radios= form["comer"];
radios.forEach(function(comida,key){
   if(comida.checked){
   console.log("para comer:"+comida.value);
      algunRadioChecked=true;
   }
 });
if(!algunRadioChecked){
   console.log("no hay nada preseleccionado");
}
```

En vez de un FOREACH también podríamos utilizar un FOR, pero no tendría mucho sentido utilizar un FOR/Let por la misma razón que la vista con los checkbox

VERSIÓN INMNO LINKIAFP

5.2.1.4. Select

El *select* es un tipo de marca HTML que muestra una lista desplegable de opciones que el usuario puede escoger. En el siguiente código vemos un ejemplo de un select con cuatro *options* utilizados para saber qué quiere comer el usuario:

```
<form id="form0" name="miForm">
  <select name="comer" value="nada" multiple>
    <option value="bocata">Bocata</option>
    <option value="wok" selected>Wok</option>
    <option value="sushi">Sushi</option>
    <option value="tortilla">Tortilla</option>
  </select>
</form>
```

El *select* es un tipo de marca HTML con un comportamiento bastante distinto a los *checkbox* o *radio* (aunque no lo parezca a simple vista). Para empezar, un *select* puede o no permitir selecciones múltiples dependiendo de si tiene o no el atributo múltiple.

En el caso que un *select* **no sea múltiple** bastará con acceder a su atributo "value" para conocer qué valor se ha marcado. En el siguiente ejemplo accedemos al *select* con su nombre y miramos su valor que mostrará el valor preseleccionado "wok".

Documento HTML

```
<form id="form0" name="miForm">
  <select name="comer" >
    <option value="bocata">Bocata</option>
    <option value="wok" selected>Wok</option>
    <option value="sushi">Sushi</option>
    <option value="tortilla">Tortilla</option>
</select>
</form>
```

Documento JavaScript

```
function checkSelect(){

let form = document.forms["miForm"];
let select= form["comer"];
let options = select.options;

for(let k=0;k<options.length;k++){
  console.log(
    options[k].value+"-"+options[k].selected
  );
}
```

Si el *select* permite selección múltiple deberemos obtener todas las opciones que tiene mediante su propiedad "options" y recorrerlas comprobando cuales están seleccionadas.

La propiedad "options" de un *select* nos retorna un HTMLOptionCollection con todos los "options" que forman el *select*. Deberemos entonces iterar por ése array preguntando cuales tienen su propiedad "selected" a true (cuales están seleccionados). Pero hay que tener presente que un HTMLOptionCollection no tiene el método *forEach*, así que deberemos utilizar un bucle FOR para recorrer las options.

VERSIÓN ASIBLE ALUMNO LINUX-P

En el siguiente ejemplo se ha añadido un SPAN que al clicarlo muestra por consola el valor de todas las opciones y si están seleccionadas o no. Mostrará por consola los mensajes:

valor:bocata seleccionado:false
valor:wok seleccionado:true
valor:sushi seleccionado:true
valor:tortilla seleccionado:false

Documento HTML

```
<form id="form0" name="miForm">
  <span onclick="checkSelect()">
      Qué hay seleccionado
  </span> <br/>
  <select name="comer" multiple>
    <option value="bocata">Bocata</option>
    <option value="wok" >Wok</option>
    <option value="sushi">Sushi</option>
    <option value="tortilla">Tortilla</option>
  </select>
</form>
```

Documento JavaScript

```
function checkSelect(){
let form = document.forms["miForm"];
let select= form["comer"];
let options = select.options;
//recorremos todas las opciones
for(let k=0;k<options.length;k++){
  console.log(//accedemos a su valor
    `Valor:`+options[k].value
    //miramos si está seleccioando
    +` Seleccionado:`+options[k].selected
 );
}
```

5.2.2. Validación del formulario

Cuando validemos formularios con JavaScript siempre tenemos que tener presente que la máxima prioridad es facilitar la web al usuario. A nadie le gusta rellenar un formulario extenso y que después de dar-le a enviar se muestre un mensaje indicando que uno de sus campos era incorrecto y que debe volver a poner todos los campos desde 0.

También es importante remarcar que los mensajes de error que se han de mostrar al usuario deberían de no bloquear la página, ser explicativos (ha de quedar claro cómo solucionar el error) y no ser demasiado intrusivos.

La validación de formularios se puede realizar en tres momentos distintos durante el proceso de rellenar y enviar el formulario. Dependerá de nuestro sentido común decidir cuándo tiene sentido hacerlo y cuándo no. Los tres momentos claves para realizar la validación son:

1. Al cambiar el contenido de un elemento del formulario. Es decir, cada vez que el usuario escribe una nueva letra en un intput texto, o cambia la opción de un select, etc.. Son las validaciones más agresivas ya que se realizan constantemente. No es buena idea lanzar una avalancha de mensajes de alerta al usuario cuando solo ha escrito una letra.

VERSIÓN IMPRIMIBLE A FP

2. Al terminar de escribir en un input y **cambiar el foco** a otro input. Ésta es una validación continua pero menos agresiva que la anterior.
3. Al terminar el formulario y clicar en "SUBMIT". Es muy importante realizar siempre ésta validación y controlar que todos los campos están correctamente escritos y en caso contrario detener el envío del formulario y mostrar un mensaje al usuario.

Para controlar cada uno de los momentos anteriores utilizaremos un conjunto de eventos específicos.

5.2.2.1. Cambio de valor

Los eventos que más utilizaremos para controlar el momento en el que se produce un cambio valor en un elemento del formulario son: *input, keydown, keypress, keyup, change* . Los cambios de valores se suelen producir al clicar sobre ellos o al escribir en ellos. Según cuando se produzca llamaremos a un evento u otro.

Los eventos *input, keydown, keypress, keyup* se lanzan al modificar el valor de un elemento del formulario al modificar su valor escribiendo en él.

El evento *change* se lanza cuando el usuario modifica el valor de un elemento del formulario interactuando con él con el ratón, típicamente con un SELECT, RADIO o CHECKBOX.

En el siguiente ejemplo utilizaremos eventos *input, keydown, keypress, keyup* para mostrar un mensaje al usuario si el nombre que va escribiendo en un input tiene menos de 4 caracteres. Para mostrar el mensaje de error hemos accedido gracias a *previousElementSibling* al tag B creado delante de cada input y modificado su contenido con *innnerHTML*. El crear el tag B delante es solo una sugerencia para facilitar el mostrar mensajes.

La validación se realiza en una función a parte y a modo de ejemplo se ejecuta en los 4 eventos: *input, keydown, keypress, keyup* para mostrar claramente cuándo se ejecuta cada uno. En éste caso si ejecutamos y escribimos la letra "a" en el *input* se mostrará por consola el mensaje:

```
evento:keydown valor:
evento:keypress valor:
evento:input valor:a
evento:keyup valor:a
```

Y dentro del tag B se mostrará el texto "mínimo 4 caracteres".

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Documento HTML

```
<head>
   <!-- Imporante usar defer!-->
   <script src="dom.js" defer></script>
</head>
<body>
<form name="formPedido">
   <b></b>Nombre:
     <input type="text" name="nombre" />
   <input type="submit" />
</form>
</body>
```

Documento JavaScript

```
let form = document.forms["formPedido"];
let inName = form["nombre"]
inName.addEventListener("input",checkName,false);
inName.addEventListener("keydown", checkName,false);
inName.addEventListener("keyup", checkName,false);
inName.addEventListener("keypress", checkName,false);

function checkName (evt){
   let inputNombre =evt.target;
   let valor =inputNombre.value;
   //información sobre el evento y el valor
   console.log("evento:"+evt.type+" valor:"+valor);

   if(valor.length<4){
      let tagB =  inputNombre.previousElementSibling;
      tagB.innerHTML="mínimo 4 caracteres";
   }else{
      let tagB =  inputNombre.previousElementSibling;
      tagB.innerHTML="";
   }
}
```

A continuación capturaremos el evento *change* para validar inputs de tipo *checkbox* . Como hemos visto que el acceso a los *checkbox* a través del nombre nos retorna una HTMLCollection , tendremos que recorrerla para añadirle los eventos . En el ejemplo comprobaremos si hay algún *checkbox* seleccionado cada vez que se cambie el valor de un *checkbox.* Para controlar el cambio de valor utilizaremos el evento “change”. En caso de que ninguno esté seleccionado mostraremos un mensaje de error en “infoCheckBox”.

VERSIÓN IMPRIMIBLE

Documento HTML

```
<head>
   <!-- Imporante usar defer!-->
   <script src="dom.js" defer></script>
</head>
<body>
<form name="formPedido">
<span id="infoCheckBox"></span>

<input type="checkbox" name="moneda"
value="euro">Euro

<input type="checkbox" name="moneda"
value="dollar">Dollar

<input type="checkbox" name="moneda"
value="yen">Yen

   <input type="submit" />
</form>
</body>
```

Documento JavaScript

```
let form = document.forms["formPedido"];
let radios = form["moneda"];
//añadimos un evento blur a cada radio
radios.forEach(function (moneda, key) {
  moneda.addEventListener("change", checkRadio, false);
});

//comprueba si hay algún check sin marcar
function checkRadio(evt) {
   let totalMarcados=0;
   // usar un for para recorrer el HTMLCollection
   for (let k = 0; k < radios.length; k++) {
      if (radios[k].checked == true) {
         totalMarcados++;
      }
   }
   let info=document.getElementById("infoCheckBox");
   if(totalMarcados==0){
      info.innerHTML="Marca mínimo un checkbox";
   }else{
      info.innerHTML="OK";
   }
}
```

VERSIÓN IMPRIMIBLE

5.2.2.2. Cambio de foco

El cambio en el foco se produce en el momento en el que dejamos de estar interactuando con un elemento y pasamos a interactuar con otro. Éste cambio se suele producir o bien por clicar sobre un nuevo elemento o bien por seleccionar el nuevo elemento con la tecla de "tabulación".

Los eventos para controlar cambio en el foco son principalmente: *focus y blur.*

El evento *focus* se ejecuta cuando un elemento gana el foco. En ése momento el usuario a dejar de interactuar con otro elemento y ha empezado a interactuar con el elemento que ha ganado el foco.

El evento *blur* se ejecuta cuando un elemento pierde el foco. El usuario ha dejado de interactuar con él y es un momento muy recomendable de validar si contiene un valor correcto.

A continuación utilizaremos *focus* y blur para comprobar cada vez que el usuario cambia el foco sobre un *select* que el usuario no ha seleccionado más de 2 platos que comer.

Como hemos visto que el acceso a los *option* a través del *select* nos retorna una HTMLOptionCollection , tendremos que recorrerla para contar cuantos platos se han seleccionado . En caso de que se hayan seleccionado más de 2 platos mostraremos un mensaje de error en el tag B previo al SELECT. El crear el tag B delante es solo una sugerencia para facilitar el mostrar mensajes con *previousElementSibling*.

Documento HTML

```
<head>
<!--Con load en JS nos ahorramos defer-->
  <script src="dom.js"></script>
</head>
<body>
<form name="formPedido">
  <b></b><select name="comer" multiple>
    <option value="wok">Wok</option>
    <option value="sushi">Sushi</option>
    <option value="papas">Papas</option>
    <option value="pasta">Pasta</option>
  </select>

  <input type="submit" />
</form>
</body>
```

Documento JavaScript

```
//ejecutar ini una vez cargada la web
window.addEventListener("load",ini,false);
function ini(){
  let form = document.forms["formPedido"];
  let comer=form["comer"];
  comer.addEventListener("blur", checkComer, false);
  comer.addEventListener("focus", checkComer, false);
}

function checkComer(evt){
  let totalSelected =0;
  let select = evt.target; //el select
  let options = select.options;//sus opciones

  //recorremos todas las opciones
  for(let k=0;k<options.length;k++){
      //miramos si está seleccioando
      if(options[k].selected  ){
        totalSelected++;
      }
  }
  //si hay más de 2 selected, mostramos el aviso
  if(totalSelected>2){
    let tagB = select.previousElementSibling;
    tagB.innerHTML = "máximo 2";
  } else {
    let tagB = select.previousElementSibling;
    tagB.innerHTML = "OK";
  }
}
```

VERSIÓN IMPRIMIBLE

En el ejemplo anterior en vez de obligar a ejecutar el JavaScript una vez cargada la web utilizando el atributo *defer* en el HTML, hemos optado por añadir un evento load a window para indicar que se ejecute la "función inicial " una vez se haya cargado la web. La "función inicial" que en el ejemplo hemos definido como *ini()* va a ser la encargada de vincular todos los eventos. Ésta técnica ayuda a encapsular mejor las variables, ya que ahora el ámbito de las variables utilizadas en vincular eventos se limita a la función *ini()*.

5.2.2.3. Envío del formulario

El evento que utilizaremos para controlar el momento en el que el usuario va a enviar el formulario será el evento *submit*. Éste evento se ejecuta inmediatamente después de que el usuario haya clicado en un input de tipo "SUBMIT" del formulario y justo antes de que el formulario se envíe.

Al ejecutarse el evento *submit* deberemos comprobar de nuevo todas las validaciones del formulario y solo permitir que se envíe el formulario con todas las validaciones correctas.

Podemos evitar el envío de un formulario de dos formas distintas según hayamos vinculado el evento *submit*.

Si hemos vinculado el evento *submit* mediante el atributo *onsubmit* evitaremos que se envíe el formulario añadiendo la sentencia *return* delante de *onsubmit* y retornando *false* en la función vinculado al evento.

En el siguiente ejemplo el formulario solo se envia si el contenido del input "name" es 22.

Documento HTML

```
<head>
  <script src="dom.js"></script>
</head>
<body>
<!--Muy importante el return validaFormm(); -->
  <form name="myForm"
        onsubmit="return validaForm();">
    Nombre:
        <input type="text" name="name">
    <input type="submit" />
  </form>
</body>
```

Documento JavaScript

```
function validaForm() {
  var formulario = document.forms["myForm"];
  var text = formulario["name"].value;
  //si el texto del input es 22 se enviará
  if (text == "22") {
    return true; //enviará el formulario
  }
  return false; //no envia el formulario
}
```

Si hemos vinculado el evento *submit* mediante el método *addEventListener()* deberemos de utilizar el método *preventDefault()* del *Event* para evitar el envío del formulario. Como en el ejemplo anterior en el siguiente código solo enviará el formulario si el input "name" contiene un valor de 22.

Documento HTML

```
<head>
  <script src="dom.js"></script>
</head>
<body>
<!--Muy importante el return validaForm(); -->
  <form name="myForm"
        onsubmit="return validaForm();">
    Nombre:
        <input type="text" name="name">
    <input type="submit" />
  </form>
</body>
```

Documento JavaScript

```
function validaForm() {
  var formulario = document.forms["myForm"];
  var text = formulario["name"].value;
  //si el texto del input es 22 se enviará
  if (text == "22") {
    return true; //enviará el formulario
  }
  return false; //no envia el formulario
}
```

VERSIÓN IMPRESA NO LINKIAFP

5.2.2.4. Expresiones regulares

Cuando tengamos que validar datos es muy común que tengamos que validar secuencias complejas. Por ejemplo validar un número de teléfono, un DNI o email implica comprobar no solo la longitud sino también la secuencia de números, letras o caracteres que forman los valores.

Para validar secuencias de datos complejos podemos utilizar las expresiones regulares. ´

En términos generales una expresión regular es una secuencia de caracteres que conforman un patrón de búsqueda sobre textos. Y en la validación de formularios con JavaScript podremos comprobar si un texto cumple o no con ésos patrones.

Los patrones pueden decir cosas como: "ha de empezar con una letra seguido de tres números de 1 al 7". En éste caso solo los textos que empiecen por letra y terminen con tres números del 1 al 7 serán correctos.

Para indicar éstos patrones se utiliza un amplio conjunto de operadores y signos con una sintaxis bastante compleja. A continuación veremos los principales elementos de las expresiones regulares y algunos ejemplos:

- Los corchetes "[ ]": agrupan a grupo de caracteres. Por ejemplo:
    - [A-z] valida si hay alguna letra mayúscula o minúscula de la A a la z.
    - [0-9] valida si hay haya algún número entre el 0 y el 9.
    - [A-z][0-9] valida si hay alguna letra mayúscula o minúscula de la A a la z seguido de algún número entre el 0 y el 9.
- Las llaves "{}": después de los corchetes indican el número mínimo y máximo de veces que deben aparecer los caracteres entre llaves. Por ejemplo:
    - [0-9] {5} : valida si el texto contiene cinco números de 0 al 9.
    - [0-9][A-z]{1,5} : valida si el texto contiene un número de 0 al 9 seguido de una a cinco letras.
    - [0-9][A-z]{3,} : valida si el texto contiene un número de 0 al 9 seguido de como mínimo tres letras.
- Carácter concreto: podemos especificar un carácter en concreto. Por ejemplo:
    - [0-9]@[A-z] : valida si el texto contiene un número seguido de una @ y seguido de un carácter.
- El más "*": indica repetición de cero a infinitas veces. Por ejemplo:
    - [0-9]@[A-z]* :valida si el texto en algún momento tiene un numero seguido @ y de 0 a infinitas letras
- El más "+": indica repetición de una a infinitas veces. Por ejemplo:
    - [0-9]+@[A-z] :valida si el texto en algún momento tiene de uno a infinitos números seguido de @ y de una letra.
- El dollar "$": indica el fin del texto. Por ejemplo:

VERSIÓN IMPRIMIBLE ALUMNO LINKIAEP

    - [0-9][A-z]{1,5}$: valida si termina con un numero seguido de como máximo 5 letras

- El acento circunflejo "^": indica el inicio del texto. Por ejemplo:
    - ^[0-9][A-z]{1,5}$: valida si solo contiene un número seguido de como máximo 5 letras y nada más.
- Los paréntesis "( )": permiten agrupar una parte de la expresión regular. Por ejemplo:
    - ^([0-9][A-z]){1,5}$: valida si solo contiene de una a cinco veces la combinación de un numero seguido una letra.
- El OR "|" : permite indicar que solo puede aparecer uno de los elementos agrupados por los paréntesis. Por ejemplo:
    - ^([0-9]|[A-z]){1,5}$: valida si solo tiene de 1 a 5 números y/o valores.
    - ^([0-9]|[A-z]){5, }$: valida si solo tiene un mínimo de 5 números y/o valores.

Para trabajar con expresiones regulares en JavaScript deberemos crear un objeto del tipo RegExp que contendrá el patrón a comparar y todos los métodos que nos van a permitir utilizarlos para comparar con un texto. Podemos inicializar la expresión regular o bien con el constructor "RegExpr" o bien como un literal con el carácter "/" :

1. let expresion1 = new RegExpr("patrón de la expresión regular");
2. let expresion2 = /Patrón de la expresión regular/;

Una vez hemos creado nuestra expresión regular la podemos utilizar para validar un texto o bien con su método "test() " o bien con el método "*match()*" del objeto String que queramos comparar siguiendo la siguiente sintaxis:

1. miExpresionRegular.test(miTexto);
2. miTexto.match(miExpresionRegular);

En el siguiente ejemplo podemos utilizamos ambas sintaxis combinadas con expresiones regulares creadas como literal o el constructor RgExp para que cada vez que se escribe en uno de los 4 inputs se compruebe respectivamente si tienen: solo letras, cualquier texto que incluya la palabra "linkia" , cualquier texto formado por un número y 5 letras, o un número de teléfono de España si espacios ni guiones.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Documento HTML

```
<form name="myForm" >
   <br />Solo Letras <b></b>
   <input type="text" name="soloLetras">

   <br />contiene palabra linkia<b></b>
   <input type="text" name="linkia">

   <br />Solo letra y 5 numeros<b></b>
   <input type="text" name="letraNums">

   <br />Telefono sin espacios<b></b>
   <input type="text" name="telefono">
</form>
```

Documento JavaScript

```
//Eventos oninput para validar al escribir en un input
window.addEventListener("load", ini, false);
function ini() {
   let info = document.forms["myForm"]["soloLetras"];
   info.addEventListener("input", checkSoloLetras, false);

   let telef = document.forms["myForm"]["telefono"];
   telef.addEventListener("input", checkTelefono, false);

   let linkia = document.forms["myForm"]["linkia"];
   linkia.addEventListener("input", checkLinkia, false);
```

Documento JavaScript

```
   let lN = document.forms["myForm"]["letraNums"];
   lN.addEventListener("input", checkLetraNums, false);
}
```

Documento JavaScript

```
/*entre el inicio y fin solo letras*/
function checkSoloLetras(evt){
   let input = evt.target;
   let texto = input.value;
   let tagB = input.previousElementSibling;
//expresión regular creada por constructor RegExp
   let expresion = new RegExp("^[A-z]*$");
   //miramos si la expresión valida el texto
   if (expresion.test(texto)) {
      tagB.innerHTML = "Valido, solo letras";
   } else {    tagB.innerHTML = "Hay no letras"; }
}
/*que contenga la palabra linkia*/
function checkLinkia(evt){
   let input = evt.target;
   let texto = input.value;
   let tagB = input.previousElementSibling;
//expresión regular creada por constructor RegExp
let expresion = new RegExp("linkia");
//miramos si el texto es válido según la expresión
   if(texto.match(expresion)){
      tagB.innerHTML = "Aparece linkia";
   } else {tagB.innerHTML = "No hay linkia";    }
}
```

```
/**formado por una letra y 5 números */
function checkLetraNums(evt){
   let input = evt.target;
   let texto = input.value;
   let tagB = input.previousElementSibling;
   //expresion regular creada como un literal
   let expresion = /^[A-z][0-9]{5}$/;
   //miramos si el texto es válido según la expresión
   if(texto.match(expresion)){
      tagB.innerHTML = "Una letra y 5 numeros";
   } else {tagB.innerHTML = "no valida";    }
}
/** número de teléfono sin espacios */
function checkTelefono(evt){
   let input = evt.target;
   let texto = input.value;
   let tagB = input.previousElementSibling;
//expresión regular creada como un literal
let expres=/^(\+34|0034|34)?[6|7|8|9][0-9]{8}$/;
//miramos si el texto es válido según la expresión
   if(texto.match(expres)){
      tagB.innerHTML = "Telefono correcto";
   } else { tagB.innerHTML="Telefono incorrecto";   }
}
```

En el siguiente video podemos ver cómo validar un formulario y depurar errores derivados en JavaScript.

5_2_2 Validación del formulario

5.3. Persistencia de datos

Como vimos al inicio, cada vez que el cliente pide una página web al servidor , el servidor retorna un HTML que el navegador ha de interpretar y realizar una petición por cada recurso contenido en ella (si la página tiene 3 fotos, se harán 4 peticiones contando el mismo HTML). Lo curioso es que el servidor por defecto no tiene ni idea de qué hemos pedido anteriormente. Así que si le pido una foto el servidor ni sabe ni le importa que se lo esté pidiendo porque anteriormente le he pedido la web con la foto.

Ésta característica hace que los servidores web de por sí no recuerden información de las interacciones que hemos hecho con ellos. Entonces, si entramos en una página web con un loggin cómo es posible que al navegar por ése sitio web nos recuerde quiénes somos?

Para conseguir que nos recuerde de alguna forma deberemos conseguir que algún dato sea “permanente". Y que además ésos datos sean independientes para cada navegador que hace consultas e incluso en cada navegador independientes de cada sitio web (no quiero que otro navegador o sitio web acceda a mis datos). Quiero en definitiva información vinculada a mi “sesión”, es decir, vinculada al navegador y al sitio web.

Cuando navegamos en modo “incognito” estaremos navegado como si fuéramos otro navegador y toda ésa información vinculada a la “sesión” será destruida al cerrar el navegador.

A continuación veremos la técnicas más ampliamente soportada para la persistencia de datos: las “cookies”.

VERSIÓN IMPRIMIBLE ALUMNO

5.3.1. Cookies

Una de las primeras técnicas más aceptadas y ampliamente utilizadas para mantener información de una "sesión". Las cookies las podemos crear desde JavaScript o desde el mismo lenguaje de servidor, ya que son valores que se pueden ir pasando con cada petición y respuesta del cliente con el servidor.

Utilizar cookies con JavaScript es bastante tedioso y ya vimos cómo hacerlo de forma nativa en la UF1 el apartado "Introducción a las Cookies". Por ello para trabajar con cookies se suelen definir y utilizar funciones propias. A continuación veremos dos funciones propuestas por "W3Schools" para gestionar las cookies:

- setCookie(nombre, valor, días) : ésta función nos permite crear, modificar y borrar una cookie. Para crear o modificar una cookie solo hemos de indicar el nombre de la cookie, el valor de la cookie y el número de días que queremos que el navegador mantenga la cookie (a no ser que el cliente las borre).
  Para borrar una cookie solo hemos de indicar el nombre de la cookie a borrar , un valor cualquiera y un número de días negativo. Al poner un número de días negativo la fecha de la cookie será anterior a la actual y la cookie se borrará.
- getCookie(nombreCookie): retorna el valor de la cookie con el nombre indicado por parámetro. Si no existe la cookie retorna una cadena vacía "".

En el siguiente ejemplo definiremos las dos funciones según están en "W3Schools " y las utilizaremos para crear una cookie de nombre "cuenta" con un valor inicial a 0. Cada vez que carguemos la página leeremos el valor de ésta cookie y le sumaremos 1 (como las cookies solo almacenan texto deberemos *parsear* el valor a entero). Finalmente mostraremos el valor por consola y en caso que el valor sea 4 la borraremos.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

```
//Crear,Modificar y Borrar una cookie
function setCookie(cname, cvalue, exdays) {
   var d = new Date();
   d.setTime(d.getTime() + (exdays * 24 * 60 * 60 * 1000));
   var expires = "expires=" + d.toUTCString();
   document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";
}
//obtener el valor de una cookie
function getCookie(cname) {
   var name = cname + "=";
   var ca = document.cookie.split(';');
   for (var i = 0; i < ca.length; i++) {        var c = ca[i];
      while (c.charAt(0) == ' ') {          c = c.substring(1);       }
      if (c.indexOf(name) == 0) { return c.substring(name.length, c.length);       }
   }   return "";
}
//obtengo el valor de la cookie "cuenta"
let cuenta = getCookie("cuenta");
if(cuenta==""){//cookie inexistente
   cuenta=0;
}else{//si existe le sumo 1
   cuenta=parseInt(cuenta)+1;
}
//creo o modifico el valor de la cookie.
setCookie("cuenta",cuenta,5);
console.log("cuenta:"+cuenta); //muestro cuenta
//si tiene un valor de 4, la borro
if(cuenta ==4){
setCookie("cuenta","",-1);
   console.log("cookie borrada");
}
```

VERSIÓN IN AFP

Test de autoevaluación

¿Con qué objetivo se deben validar los datos en el servidor?

a) Evitar tener que validar los datos en el servidor.

b) Evitar tener que realizar consultas a la base de datos para comprobar que los datos son correctos.

c) Mejorar la experiencia del usuario y de paso evitar demasiadas peticiones al servidor

d) Aumentar la seguridad de la aplicación web.

Las expresiones regulares...

a) Son los métodos más utilizados en JavaScript.

b) Permiten comprobar si un patrón aparece en un texto.

c) Son una Collection de eventos de rápido acceso.

d) Permiten transformar los datos en un formato concreto.

¿Qué atributo nos indica si un *checkbox* está marcado?

a) selected

b) checked

c) marked

d) active

VERSIÓN IMPRIMIBLE ALUMNO LINKIA FP

Recursos y enlaces

- Interesante y sencilla web que nos muestra fácilmente las *keyCode* que se capturan al pulsar una tecla según el evento: http://asquare.net/javascript/tests/KeyCode.html
- Documentación sobre el objeto Event en W3Schools
- Documentación sobre el objeto Event en MDN
- Recurso sobre las cookies en W3Schools

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Conceptos clave

- **Evento:** son aquellas interacciones o cambios de estado que se pueden producir contra la página web u objetos JavaScript
- Al lanzarse un evento primero se ejecuta la **fase de captura** y luego la **fase de *bubbling***.
- Todas las funciones llamadas desde un evento reciben como parámetro un objeto del tipo ***Event*** con información sobre el evento.
- Las ***cookies*** no permiten almacenar datos que persisten entre distintas peticiones dentro de un mismo sitio web

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Ponlo en práctica

Actividad 1

Crea una aplicación web con un evento a la ventana *window* para que se ejecute una función inicial una vez se haya cargado el contenido de la web. Programa en la función inicial que cada vez que se pulse un bóton se cree un nuevo div con texto introducido por el usuario. El clicar el nuevo div añade aleatoriamente uno de los siguientes 3 eventos:

1. Al mover el raton encima el div, cambia su color a naranja
2. Al quitar el raton de encima el div cambia su texto por "nuevo Texto
3. Al pulsar el div elimina su elemento anterior

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Crea un formulario con dos inputs tipo texto que muestren un mensaje a su lado

1. Cuando el ratón sale del primer input su contenido ha de estar compuesto por cualquier combinación de números o letras, pero mínimo longitud 3
2. Cuando el usuario escribe en el segundo input su contenido ha de estar compuesto por 3 números y 2 letras.
3. A demás no se debe enviar el formulario si no se cumple alguna regla

Los solucionarios están disponibles en el aula virtual.

VERSIÓN IMPRESA DE ALUMNO LINKIA

SOLUCIONARIOS

Test de autoevaluación tema 4

¿Qué tipo de nodos pueden tener hijos?

m) Todos

n) Element

o) Ninguno

p) Text

¿Cuál es la sentencia recomendada para modificar el atributo “id” de un elemento con “id” “pelota”?.

q) document.getElementByd(“pelota”).id=”nuevo valor”;

r) document.getElementByd(“pelota”).setAttribute (“id”, “nuevo valor”);

s) document.getElementsByTagName(“pelota”).id=”nuevo valor”;

t) document.getElementsByTagName(“pelota”).setAttribute (“id”, “nuevo valor”);

¿Cómo accedemos al elemento anterior del elemento con “id” “pelota”?

u) document.getElementByd(“pelota”).previousElementSibling();

v) document.getElementByd(“pelota”).previousSibling();

w) document.getElementByd(“pelota”).previousSibling;

x) **document.getElementByd(“pelota”).previousElementSibling;**

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Ponlo en práctica tema 4

Actividad 1

Crea un documento HTML con una lista ordenada de 7 *items*. Programa que:

1. al clicar en el primero se cambie su color e texto a verde
2. al clicar en el segundo se cambie su color de fondo a amarillo
3. al clicar en el tercero cambie el último elemento LI por el primero
4. al clicar en el cuarto se cambie el color de fondo de todos a azul
5. al clicar en el quinto, se borre a él.
6. al clicar en el sexto, se borre el *element* hermano anterior
7. al clicar en el séptimo, se añada otro LI con texto "NUEVO LI" y color de fondo rojo

Los solucionarios están disponibles en el aula virtual.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Test de autoevaluación tema 5

¿Con qué objetivo se deben validar los datos en el servidor?

e) Evitar tener que validar los datos en el servidor.

f) Evitar tener que realizar consultas a la base de datos para comprobar que los datos son correctos.

g) Mejorar la experiencia del usuario y de paso evitar demasiadas peticiones al servidor

h) Aumentar la seguridad de la aplicación web.

Las expresiones regulares...

e) Son los métodos más utilizados en JavaScript.

f) Permiten comprobar si un patrón aparece en un texto.

g) Son una Collection de eventos de rápido acceso.

h) Permiten transformar los datos en un formato concreto.

¿Qué atributo nos indica si un *checkbox* está marcado?

e) selected

f) checked

g) marked

h) active

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP

Ponlo en práctica tema 5

Actividad 1

Crea una aplicación web con un evento a la ventana *window* para que se ejecute una función inicial una vez se haya cargado el contenido de la web. Programa en la función inicial que cada vez que se pulse un bóton se cree un nuevo div con texto introducido por el usuario. El clicar el nuevo div añade aleatoriamente uno de los siguientes 3 eventos:

1. Al mover el raton encima el div, cambia su color a naranja
2. Al quitar el raton de encima el div cambia su texto por "nuevo Texto
3. Al pulsar el div elimina su elemento anterior

Los solucionarios están disponibles en el aula virtual.

Actividad 2

Crea un formulario con dos inputs tipo texto que muestren un mensaje a su lado

1. Cuando el ratón sale del primer input su contenido ha de estar compuesto por cualquier combinación de números o letras, pero mínimo longitud 3
2. Cuando el usuario escribe en el segundo input su contenido ha de estar compuesto por 3 números y 2 letras.
3. A demás no se debe enviar el formulario si no se cumple alguna regla

Los solucionarios están disponibles en el aula virtual.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP