

Resumen tema 6: Gestión de eventos y formularios en JavaScript

Desarrollo Web en Entorno Cliente

18 de enero de 2023

CIFP Carlos III - Cartagena

Santiago Francisco San Pablo Raposo

2º curso DAW

Contenido

# Índice de ilustraciones.

# Índice de tablas.

Resumen del tema 6.

# Introducción.

* Se explica la forma de acceso a los formularios, a sus propiedades y métodos.
* Se explica la forma de pasar a una función la referencia a un objeto.
* Se hace una introducción a lo que son los eventos.
* Se explica cómo llevar a cabo la validación de un formulario y cómo utilizar expresiones regulares para validar el contenido de los campos.
* Se hace una introducción a lo que son las cookies y cómo gestionarlas con JavaScript.

# 1.- El objeto Form.

La mayor parte de interactividad entre una página web y el usuario tiene lugar a través de un formulario.

En este apartado verás cómo identificar un formulario y sus objetos, cómo modificarlos, cómo examinar las entradas de usuario, enviar un formulario, validar datos, etc.

Los formularios y sus controles, son objetos del DOM. Por ejemplo, un formulario tiene una propiedad **action**, que le indica al navegador donde tiene que enviar las entradas del usuario, cuando se envía el formulario.

JavaScript añade a los formularios dos características muy interesantes:

* JavaScript permite examinar y validar las entradas de usuario directamente, en el lado del cliente.
* JavaScript permite dar mensajes instantáneos, con información de la entrada del usuario.

**El objeto de JavaScript Form, es una propiedad del objeto document**. Se corresponderá con la etiqueta <form> del HTML. Un formulario podrá ser enviado llamando al método submit de JavaScript, o bien haciendo click en el botón submit del formulario.

**Reflexiona**:

¿Si, por ejemplo, usamos JavaScript para validar un formulario, será necesario también validar esos datos en el lado del servidor?

¿Qué pasaría con nuestras validaciones si por ejemplo desactivamos JavaScript en el navegador?

¿Y que pasaría si alguien programa una copia de nuestro formulario en otro servidor web para enviar datos a nuestro servidor sin validarlos previamente?

## 1.1.- Formas de selección del objeto Form.

Partiendo del siguiente ejemplo:

<div id="menulateral"> <form id="contactar" name="contactar" action="...">...</form></div>

Tendremos 3 métodos de selección del objeto Form en el documento:

* **Método 1**: a través del método getElementById del DOM:

let formulario = document.getElementById("contactar");

* **Método 2**: A través del método getElementsByTagName() del DOM

let formularios = document.getElementsByTagName("form");let primerFormulario = formularios[0]; // primer formulario del documento

O también todo en una única línea:

let primerFormulario = document.getElementsByTagName("form")[0] ;

Otra posibilidad interesante que te permite el método anterior, es la de buscar objetos con un padre determinado, por ejemplo:

let menu = document.getElementById("menulateral");

let formularios = menu.getElementsByTagName("form"); // formularios contenidos en el menu lateral

let primerFormulario = formularios[0]; // primer formulario en el menú lateral

* **Método 3**: a través de la colección **forms[]** del **objeto document**.

let formularios = document.forms; // la referencia a todos los formularios del documento

let miFormulario = formularios[0]; // primer formulario del documento

o bien:

let miFormulario = document.forms[0]; // primer formulario del documento

o bien:

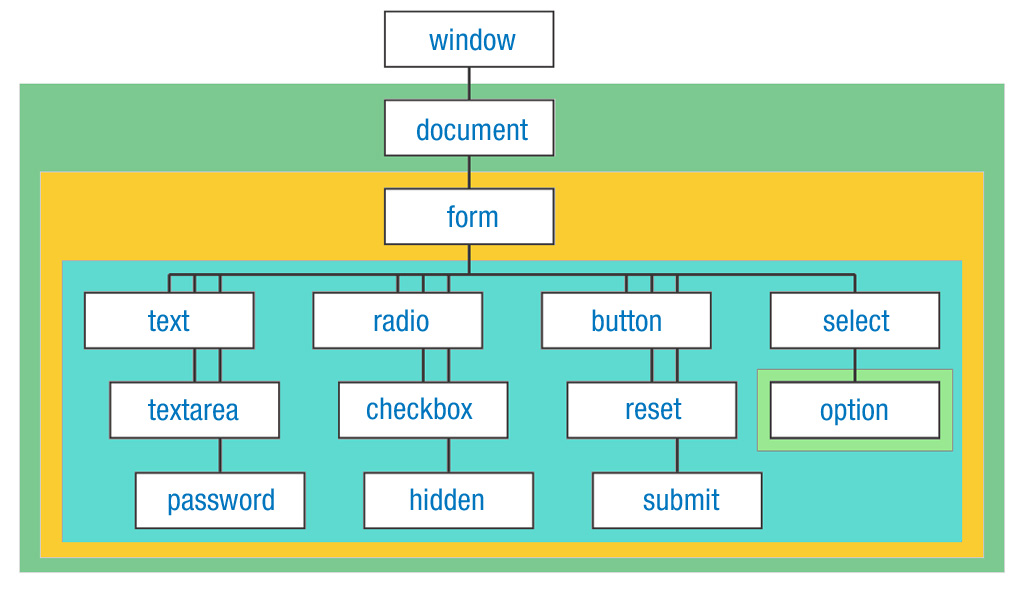
let miFormulario = formularios["contactar"]; // referenciamos al formulario con name "contactar"

## 1.2.- El formulario como objeto y contenedor.

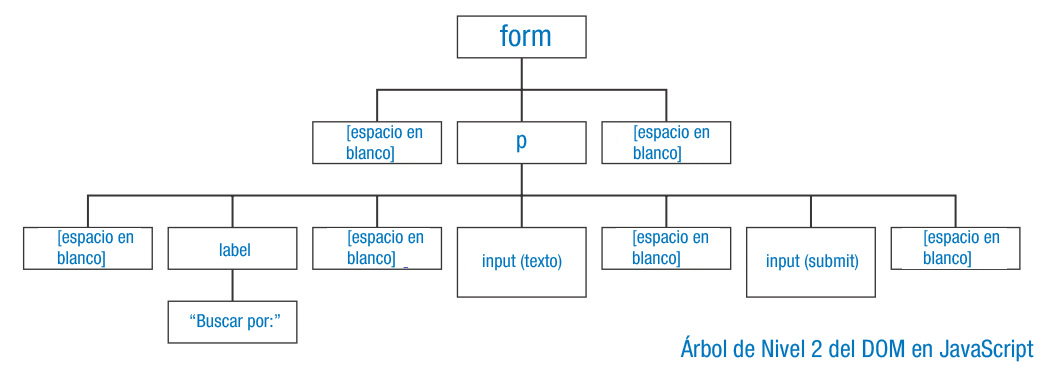
Con la evolución de JavaScript, nos encontramos con que el objeto Form está dentro de dos árboles al mismo tiempo.

En las nuevas definiciones del DOM, se especifica que Form es el padre de todos sus nodos hijos, incluidos objetos y textos, mientras que en las versiones antiguas Form sólo era padre de sus objetos (input, select, button y elementos textarea).

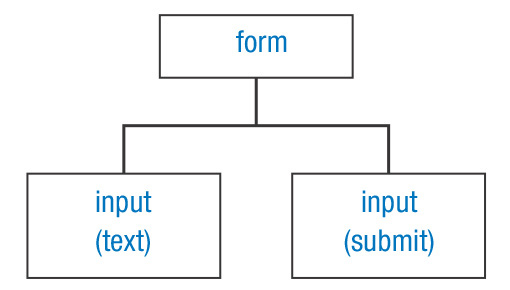
* **Jerarquía de nivel 0 del DOM para formularios y controles**:



* **Árbol de nivel 2 del DOM para un formulario típico**:



* **Árbol de nivel 0 para el mismo formulario**:



Los dos árboles que te mostré anteriormente pueden ser útiles para diferentes propósitos. El árbol del DOM de nivel 2, se puede utilizar para leer y escribir en todo el documento con un nivel muy fino de granularidad[[1]](#footnote-1). El árbol del DOM de nivel 0, hace muchísimo más fácil leer y escribir los controles del formulario.

## 1.3.- Acceso a propiedades y métodos del formulario.

Se pueden asignar atributos como **name**, **action**, **target** y **enctype**. Cada uno deestos atributos es una propiedad del objeto **Form**, a las que podemos acceder como en el ejemplo:

let paginaDestino = objetoFormulario.action;

**Para modificar una de estas propiedades** lo haremos mediante asignaciones, por ejemplo:

objetoFormulario.action = "http://www.educacion.gob.es/recepcion.php";

**Estas dos instrucciones las podemos recomponer usando referencias a objetos**:

let paginaDestino = document.getElementByid("id").action;document.forms[0].action = "http://www.educacion.gob.es/recepcion.php";

Propiedades del objeto Form

|  |  |  |
| --- | --- | --- |
| Propiedad | Descripción | W3C |
| acceptCharset | Ajusta o devuelve el valor del atributo **accept-charset** en un formulario. | **Sí** |
| action | Ajusta o devuelve el valor del atributo **action** en un formulario. | **Sí** |
| enctype | Ajusta o devuelve el valor del atributo **enctype** en un formulario. | **Sí** |
| length | Devuelve el número de elementos en un formulario. | **Sí** |
| method | Ajusta o devuelve el valor del atributo **method** en un formulario. | **Sí** |
| name | Ajusta o devuelve el valor del atributo **name** en un formulario. | **Sí** |
| target | Ajusta o devuelve el valor del atributo **target** en un formulario. | **Sí** |

Métodos del objeto Form

|  |  |  |
| --- | --- | --- |
| Método | Descripción | W3C |
| reset() | Resetea un formulario. | **Sí** |
| submit() | Envía un formulario. | **Sí** |

Propiedad form.elements[]

Esta propiedad es otro array, con todos los campos input en el orden en el cual aparecen en el código fuente del documento.

Generalmente, es mucho más eficaz y rápido referenciar a un elemento individual usando su ID, pero a veces, los scripts necesitan recorrer cada elemento del formulario.

let miFormulario = document.getElementById("contactar"); // guardamos la referencia del formulario en una variable.

if (! miFormulario) return false; // Si no existe ese formulario devuelve false.

for (let i=0; i< miFormulario.elements.length; i++) {

if (miFormulario.elements[i].type == "text") {

miFormulario.elements[i].value = "";

}

}

# 2.- Objetos relacionados con formularios.

**Caso práctico**: Cada uno de los elementos de un formulario, son objetos en JavaScript que tendrán propiedades y métodos, que nos permitirán realizar acciones sobre ellos. Gracias a esos métodos y propiedades, podremos realizar acciones como validar el contenido de un formulario,marcar o desmarcar una determinada opción, mostrar contenido de un campo u ocultarlo, etc.

Para poder trabajar con los objetos de un formulario, lo primero que necesitas saber es, cómo referenciar a ese objeto, por ejemplo, de estas formas:

document.getElementById("id-del-control");

o bien

document.nombreFormulario.name-del-control;

Por ejemplo, si consideramos un ejemplo sencillo de formulario:

<form id="formularioBusqueda" action="cgi-bin/buscar.pl">

<p>

<input type="text" id="entrada" name="cEntrada">

<input type="submit" id="enviar" name="enviar" value="Buscar...">

</p>

</form>

Las siguientes referencias al campo de texto entrada, serán todas válidas:

document.getElementById("entrada");

document.formularioBusqueda.cEntrada;

document.formularioBusqueda.elements[0];

document.forms["formularioBusqueda"].elements["cEntrada"];

document.forms["formularioBusqueda"].cEntrada;

**Aunque muchos de los controles de un formulario tienen propiedades en común, algunas propiedades son únicas a un control en particular**. Por ejemplo, en un objeto select (o checkbox, o botones de tipo radio) tienes propiedades que te permiten conocer la opción que está actualmente seleccionada.

## 2.1.- Objeto input de tipo texto.

Los 4 elementos de tipo texto de los formularios son: **text**, **password**, **hidden** y elementos **textarea**. Todos los elementos, excepto los tipos **hidden**, se mostrarán en la página.

**Recomendación**: asignar a cada objeto de tu formulario un atributo id único y que coincida con el name de ese objeto.

Cuando se envían los datos de un formulario a un programa en el lado del servidor, lo queen realidad se envía son los atributos name, junto con los valores (contenido del atributo value.).

La propiedad más utilizada (value) puede ser recuperada o modificada en cualquier momento mediante JavaScript.

**El contenido de un value es siempre una cadena de texto**, y quizás puedas necesitar realizar conversiones numéricas si quieres realizar operaciones matemáticas con esos textos.

**En este tipo de objetos, los gestores de eventos** (que verás más adelante) **se podrán disparar de múltiples formas**: por ejemplo, cuando pongamos el foco en un campo (situar el cursor dentro de ese campo) o modifiquemos el texto (al introducir el texto y salir del campo).

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">

<title>DWEC05 - Propiedad VALUE de un objeto de tipo texto</title>

<script type="text/javascript">

const convertirMayusculas = () => {

/\*

En este ejemplo accedemos a la propiedad value de un objeto con id nombre y le asignamos

su contenido actual pero convertido a mayúsculas con el método toUpperCase() del objeto String.

\*/

document.getElementById("nombre").value=document.getElementById("nombre").value.toUpperCase();

}

</script>

</head>

<body>

<h1>Propiedad VALUE de un objeto INPUT de tipo TEXT</h1>

<form id="formulario" action="pagina.php">

<p>

<label for="nombre">Nombre y Apellidos: </label>

<input type="text" id="nombre" name="nombre" value="" size="30" onblur="convertirMayusculas()">

</p>

<p>

Introduce tu Nombre y Apellidos y haz click fuera del campo.

</p>

</form>

</body>

</html>

Propiedades del objeto INPUT de tipo texto

|  |  |  |
| --- | --- | --- |
| Propiedad | Descripción | W3C |
| defaultValue | Ajusta o devuelve el valor por defecto de un campo de texto. | Sí |
| form | Devuelve la referencia al formulario que contiene ese campo de texto. | Sí |
| maxLength | Devuelve o ajusta la longitud máxima de caracteres permitidos en el campo de tipo texto | Sí |
| name | Ajusta o devuelve el valor del atributo **name** de un campo de texto. | Sí |
| readOnly | Ajusta o devuelve si un campo es de sólo lectura, o no. | Sí |
| size | Ajusta o devuelve el ancho de un campo de texto (en caracteres). | Sí |
| type | Devuelve el tipo de un campo de texto. | Sí |
| value | Ajusta o devuelve el contenido del atributo **value** de un campo de texto. | Sí |

Métodos del objeto INPUT de tipo texto

|  |  |  |
| --- | --- | --- |
| Metodo | Descripción | W3C |
| select() | Selecciona el contenido de un campo de texto. | Sí |

**Para saber más**: [sobre las propiedades y métodos de los objetos **input**](https://www.w3schools.com/jsref/dom_obj_all.asp) y encontrarás [más ejemplos](https://www.htmlquick.com/es/reference/tags/input.html).

## 2.2.- Objeto input de tipo checkbox.

En los botones de un formulario la propiedad **value** nos mostrará el texto del botón, pero en un **checkbox** la propiedad **value** es un texto que está asociado al objeto. Este texto no se mostrará en la página, y su finalidad es la de asociar un valor con la opción actualmente seleccionada. Dicho valor será el que se enviará, cuando enviemos el formulario.

**Por ejemplo**:

<label for="cantidad">Si desea recibir 20 Kg marque esta opción: </label>

<input type="checkbox" id="cantidad" name="cantidad" value="20 Kg">

Si chequeamos este **checkbox** y enviamos el formulario, el navegador enviará el par **name/value**"cantidad" y "20 Kg". Si el **checkbox** no está marcado, entonces este campo no será enviado en el formulario.

* El texto del **label** se mostrará en la pantalla pero las etiquetas **label** no seenvían al servidor
* Para saber si un campo de tipo **checkbox** está o no marcado, disponemosde la propiedad **checked**.

**Ejemplo sencillo en el que manejamos dicha propiedad**:

<!DOCTYPE html>

<html>

<head>

<script type="text/javascript">

const marcar = () => {

document.getElementById("verano").checked = true;

}

const desmarcar = () => {

document.getElementById("verano").checked = false;

}

</script>

</head>

<body>

<form action="" method="get">

<label for="verano">¿Te gusta el verano?</label>

<input type="checkbox" id="verano" name="verano" value="Si"/>

</form>

<button onclick="marcar()">Marcar Checkbox</button>

<button onclick="desmarcar()">Desmarcar Checkbox</button>

</body>

</html>

Propiedades del objeto INPUT de tipo checkbox

|  |  |  |
| --- | --- | --- |
| Propiedad | Descripción | W3C |
| checked | Ajusta o devuelve el estado **checked** de un **checkbox**. | Sí |
| defaultChecked | Devuelve el valor por defecto del atributo **checked**. | Sí |
| form | Devuelve la referencia al formulario que contiene ese campo **checkbox**. | Sí |
| name | Ajusta o devuelve el valor del atributo **name** de un **checkbox**. | Sí |
| type | Nos indica que tipo de elemento de formulario es un **checkbox**. | Sí |
| value | Ajusta o devuelve el valor del atributo **value** de un **checkbox**. | Sí |

## 2.3.- Objeto input de tipo radio.

El ajustar un grupo de objetos de tipo radio desde JavaScript requiere un poco más de trabajo. Para dejar que el navegador gestione un grupo de objetos de tipo radio, deberemos asignar el mismo atributo **name** a cada uno de los botones del grupo.

Cuando le asignamos el mismo **name** a varios elementos en un formulario, el navegador lo que hace es crear un array con la lista de esos objetos que tienen el mismo **name**. El contenido del atributo **name** será el nombre del array.

Podemos ver cuántos botones hay en un grupo radio, consultando la propiedad length de ese grupo:

objetoFormulario.nombregrupo.length;

Y si queremos acceder a la propiedad **checked** de un botón en particular, lo haremos accediendo a la posición del array, a su propiedad **checked**:

objetoFormulario.nombregrupo[0].checked; // Accedemos a la propiedad checked del primer botón del grupo

**Ejemplo**:

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<title>DWEC05 - Trabajando con objetos input de tipo radio</title>

<script type="text/javascript">

const mostrarDatos = () => {

for (let i=0;i<document.formulario.actores.length; i++) {

if (document.formulario.actores[i].checked)

alert(document.formulario.actores[i].value);

}

}

</script>

</head>

<body>

<h1>Trabajando con objetos input de tipo radio</h1>

<form name="formulario" action="stooges.php">

<fieldset>

<legend>Selecciona tu actor favorito:</legend>

<label for="actor1">Willis</label>

<input type="radio" name="actores" id="actor1"value="Walter Bruce Willis - 19 de Marzo de 1955" checked>

<label for="actor2">Carrey</label>

<input type="radio" name="actores" id="actor-2" value="James Eugene Jim Carrey - 17 de Enero de 1962">

<label for="actor3">Tosar</label>

<input type="radio" name="actores" id="actor-3" value="Luis Tosar - 13 de Octubre de 1971">

<input type="button" id="consultar" name="consultar" value="Consultar Más Datos" onclick="mostrarDatos()">

</fieldset>

</form>

</body>

</html>

**Para saber más**: [sobre las propiedades y métodos de este objeto](https://desarrolloweb.com/articulos/1448.php).

## 2.4.- Objeto select.

Un objeto **select** está compuesto realmente de un array de objetos **option**. El objeto **select** se suele mostrar como una lista desplegable en la que seleccionas una de las opciones, aunque también tienes la opción de selecciones múltiples, según definas el objeto en tu documento.

Algunas propiedades pertenecen al objeto select al completo, mientras que otras, solo se pueden aplicar a las opciones individuales dentro de ese objeto.

* **Si lo que quieres hacer es detectar la opción seleccionada por el usuario**, y quieres usar JavaScript,tendrás que utilizar propiedades tanto de **select**, como de **option**.

objetoFormulario.nombreCampoSelect.selectedIndex;

document.getElementById("objetoSelect").selectedIndex;

* Las opciones tienen dos propiedades accesibles que son **text** y **value**, y que **te permitirán acceder al texto visible en la selección y a su valor interno** para esa opción.

objetoFormulario.nombreCampoSelect.options[n].text;

objetoFormulario.nombreCampoSelect.options[n].value;

o también usando **document.getElementById("objetoSelect").options[n].text** ó **.value**

**Ejemplo**:

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">

<title>DWEC05 - Trabajando con un objeto Select</title>

<script type="text/javascript">

const consultar = () => {

let provincias = document.getElementById("provincias");

let texto = provincias.options[provincias.selectedIndex].text;

let valor = provincias.options[provincias.selectedIndex].value;

alert(`Datos de la opción seleccionada:\n\nTexto: ${texto}\nValor: ${valor}`);

}

</script>

</head>

<body>

<h1>Trabajando con un objeto Select</h1>

<form id="formulario">

<p>

<label for="provincias">Seleccione provincia: </label>

<select name="provincias" id="provincias">

<option value="AL">Almería</option>

<option value="JA">Jaen</option>

<option value="GR">Granada</option>

<option value="MA">Málaga</option>

<option value="SE">Sevilla</option>

<option value="CO">Córdoba</option>

<option value="CA">Cádiz</option>

<option value="HU">Huelva</option>

</select>

</p>

<p>

Selecciona una opción y pulsa el botón.

</p>

<input type="button" name="boton" value="Consultar información de la opción" onclick="consultar()"/>

</form>

</body>

</html>

**Debes conocer**: [propiedades y métodos del objeto Select](https://www.htmlquick.com/es/reference/tags/select.html).

## 2.5.- Pasando objetos a las funciones usando this.

***This*** hace referencia siempre al objeto que contiene el código de JavaScript en donde usamos dicha palabra reservada. **Por ejemplo**, si programamos una función para un botón, que al hacer click haga algo, si dentro de esa función usamos la palabra ***this***, entonces estaremos haciendo referencia al objeto en el cuál hemos hecho click, que en este caso será el botón.

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">

<title>Uso de la palabra reservada this</title>

<script type="text/javascript">

const identificar = (objeto) => {

let nombre = objeto.name;

let id = objeto.id;

let valor = objeto.value;

let tipo = objeto.type;

alert(`Datos del campo pulsado:\n\nNombre: ${nombre}\nID: ${id}\nValor: ${valor}\nTipo: ${tipo}`);

}

</script>

</head>

<body>

<h1>Trabajando con this</h1>

<form id="formulario">

<p>

<label for="nombre">Nombre: </label>

<input type="text" name="nombre" id="nombre" value="Bob" onclick="identificar(this)"/>

<label for="apellidos">Apellidos: </label>

<input type="text" name="apellidos" id="apellidos" value="Esponja" onclick="identificar(this)"/>

<label for="edad">Edad: </label>

<input type="number" name="edad" id="edad" value="18" onclick="identificar(this)"/>

<label for="pais">Pais: </label>

<label for="pais1">España: </label>

<input type="radio" name="pais" id="pais1" value="ES" onclick="identificar(this)"/>

<label for="pais2">Francia: </label>

<input type="radio" name="pais" id="pais2" value="FR" onclick="identificar(this)"/>

</p>

<p>

Haga click en cada uno de los campos para ver más información.

</p>

</form>

</body>

</html>

En este ejemplo, cada vez que hagamos click en alguno de los objetos, llamaremos a la función **identificar()** y a esa función le pasaremos como parámetro ***this***, que en este caso será la referencia al objeto en el cuál hemos hecho click.

En el siguiente apartado veremos los eventos y allí te mostraré otro uso de ***this*** por ejemplo dentro de la función **identificar()** sin tener que pasar ningún parámetro.

# 3.- Eventos.

Hay que tener en cuenta que, sin eventos prácticamente no hay scripts. La razón es muy simple, **JavaScript fue diseñado para añadir interactividad a las páginas: el usuario realiza algo y la página reacciona**.

Cuando el usuario hace algo se produce un evento. También habrá algunos eventos que no están relacionados directamente con acciones de usuario: por ejemplo el evento de carga(**load**) de un documento.

## 3.1.- Modelo de registro de eventos en línea.

**En el modelo de registro de eventos en línea**, el evento es añadido como un atributo más a la etiqueta HTML, como por ejemplo:

<a href="pagina.html" onClick="alert('Has pulsado en el enlace')">Pulsa aqui</a>

Cuando hacemos click en el enlace, se llama al gestor de eventos onClick (al hacer click) y se ejecuta el script; que contiene en este caso una alerta de JavaScript. También se podría realizar lo mismo pero llamando a una función:

const alertar = () => {

alert("Has pulsado en el enlace");

}

...

<a href="pagina.html" onClick="alertar()">Pulsa aqui</a>

No uses el modelo de registro de eventos en línea.

Este modelo no se recomienda. tiene el problema de que estamos mezclando la estructura de la página web con la programación de la misma, y lo que se intenta hoy en día es separar la programación en JavaScript, de la estructura HTML.

En el ejemplo anterior, cuando haces click en el enlace se mostrará la alerta y a continuación te conectará con la **pagina.html**.

Éste modo de funcionamiento ha sido un principio muy importante en la gestión de eventos. Si un evento genera la ejecución de un script y además también se genera la acción por defecto para ese objeto entonces:

1. El script se ejecutará primero.
2. La acción por defecto se ejecutará después.

Evitar la acción por defecto

Cuando programamos un gestor de eventos, ese gestor podrá devolver un valor booleano **true** o **false**. Eso tendremos que programarlo con la instrucción **return true|false**. False quiere decir "no ejecutes la acción por defecto". Por lo tanto nuestro ejemplo quedará del siguiente modo:

<a href="pagina.html" onClick="alertar(); return false">Pulsa aqui</a>

También se puede utilizar el método **preventDefault()** del evento para lo mismo.

## 3.2.- Modelo de registro de eventos tradicional.

Con la llegada de DHTML (Dynamic HTML), el modelo se extendió para ser más flexible. En este nuevo modelo el evento pasa a ser una propiedad del elemento, así que por ejemplo los navegadores modernos ya aceptan el siguiente código de JavaScript:

elemento.onclick = hacerAlgo; // cuando el usuario haga click en el objeto, se llamará a la función hacerAlgo()

**Esta forma de registro, no fue estandarizada por el W3C, pero debido a que fue ampliamente utilizada por Netscape y Microsoft, todavía es válida hoy en día**. La **ventaja** de este modelo es que podremos asignar un evento a un objeto desde JavaScript, con lo que ya **estamos separando el código de la estructura**.

Para eliminar un gestor de eventos de un elemento u objeto, le asignaremos ***null***:

elemento.onclick = null;

**Otra ventaja**: Como el gestor de eventos es una función, podremos realizar una llamada directa a ese gestor, con lo que estamos disparando el evento de forma manual.

elemento.onclick(); // Al hacer ésto estamos disparando el evento click de forma manual y se ejecutará la función hacerAlgo()

Sin paréntesis

En el registro del evento no usamos paréntesis **()**. El método ***onclick*** espera que se le asigne una función completa. Si haces: **element.onclick = hacerAlgo();** la función será ejecutada y el resultado que devuelve esa función será asignado a ***onclick***. Pero ésto no es lo que queremos que haga, queremos que se ejecute la función cuando se dispare el evento.

## 3.3.- Modelo de registro avanzado de eventos según W3C.

El W3C en la especificación del DOM de nivel 2, pone especial atención en los problemas del modelo tradicional de registro de eventos. En este caso ofrece una **manera sencilla de registrar los eventos que queramos, sobre un objeto determinado**.

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

La clave está en el método **addEventListener()**, que tiene tres argumentos:

* el **tipo de evento**
* la **función a ejecutar**
* **options**, que puede tener las siguientes propiedades:
  + **once**: si es true entonces el listener se remueve automáticamente después de activarlo.
  + **Capture**: la fase en la que se controla el evento (propagación/captura).
    - un **valor** booleano (**true** o **false**), que se utiliza para indicar cuándo se debe capturar el evento: en la fase de captura (**true**) o de burbujeo (**false**).
  + **Passive**: si es true entonces el handler no llamará a preventDefault().

**Por ejemplo**: para registrar la función alertar() de los ejemplos anteriores, haríamos:

document.getElementById("miEnlace").addEventListener('click', alertar, false);

const alertar = () => {

alert(`Te conectaremos con la página: ${this.href}`);

}

La ventaja de este método, es que **podemos añadir tantos eventos como queramos**. Por ejemplo:

document.getElementById("miEnlace").addEventListener('click', alertar, false);

document.getElementById("miEnlace").addEventListener('click', avisar, false);

document.getElementById("miEnlace").addEventListener('click', chequear, false);

Esto hará que cuando hagamos click en miEnlace, se disparará la llamada a las tres funciones. Aunque el W3C no indica el orden de disparo, por lo que no sabemos cuál de las tres funciones se ejecutará primero.

**Fíjate también, que el nombre de los eventos al usar addEventListener no lleva 'on' al comienzo**.

También se pueden usar funciones anónimas (sin nombre de función), haciendo uso de la función con notación flecha, como ya veníamos haciendo:

element.addEventListener('click', () => {

this.style.backgroundColor = '#cc0000';

}, false);

Uso de la palabra reservada this

Esta palabra tiene exactamente la misma funcionalidad que hemos visto en el modelo tradicional, la de poder acceder al objeto que ha invocado la función asociada a ese evento.

¿Qué eventos han sido registrados?

Este es uno de los problemas de la implementación del modelo de registro del W3C: no podemos saber con antelación los eventos que hemos registrado a un elemento.

Sin embargo, eso ha cambiado, ya que el W3C, en el reciente nivel 3 del DOM, introdujo un método llamado eventListenerList, que almacena una lista de las funciones que han sido registradas a un elemento.

**Para eliminar un evento** de un elemento, usaremos el método **removeEventListener()**:

elemento.removeEventListener('evento', función, false|true);

Es muy importante pasar exactamente la misma función que asignamos con addEventListener.

**Para cancelar un evento**: este modelo nos proporciona el método **preventDefault()**.

**Podemos capturar el evento dentro de la función que lo maneja**:

<script>

elem.onclick = function(event) {

// muestra el tipo de evento, el elemento y las coordenadas del click

alert(event.type + " en el " + event.currentTarget);

alert("Coordenadas: " + event.clientX + ":" + event.clientY);

};

</script>

**También podemos hacerlo desde handlers HTML**:

<input type="button" onclick="alert(event.type)" value="Event type">

### 3.3.1.- Objetos handlers: handleEvent.

Podemos asignar no solo una función, sino un objeto como handler del evento usando addEventListener.

<button id="elem">Haz click en mí</button>

<script>

let obj = {

handleEvent(event) {

alert(event.type + " en " + event.currentTarget);

}

};

elem.addEventListener('click', obj);

</script>

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

**También podemos usar objetos de una clase personalizada**:

<button id="elem">Haz click en mí</button>

<script>

class Menu {

handleEvent(event) {

switch(event.type) {

case 'mousedown':

elem.innerHTML = "Botón del mouse presionado";

break;

case 'mouseup':

elem.innerHTML += "...y soltado.";

break;

}

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

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

<button id="elem">Haz click en mí</button>

<script>

class Menu {

handleEvent(event) {

// mousedown -> onMousedown

let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);

this[method](event);

}

onMousedown() {

elem.innerHTML = "Botón del mouse presionado";

}

onMouseup() {

elem.innerHTML += "...y soltado.";

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

## 3.4.- Orden de disparo de los eventos.

Imagina que tenemos un elemento contenido dentro de otro elemento, y que tenemos programado el mismo tipo de evento para los dos (por ejemplo el evento click). **¿Cuál de ellos se disparará primero?** Sorprendentemente, esto va a **depender del tipo de navegador que tengamos**.

Ambos tienen programado el evento de **click**. Si el usuario hace click en el **elemento2**, provocará un click en ambos. ¿Pero cuál de ellos se disparará primero? **¿Cuál es el orden de los eventos?**

W3C decidió que, **primero se produce la fase de captura hasta llegar al elemento de destino**. Luego se produce la fase de burbujeo hacia arriba. Este modelo es el estándar que todos los navegadores deben seguir para ser compatibles entre sí.

Tú podrás decidir cuando quieres que se registre el evento: en la fase de captura o en la fase de burbujeo. **Por ejemplo**:

elemento1.addEventListener('click', hacerAlgo1 ,true);

elemento2.addEventListener('click', hacerAlgo2, false);

Si el usuario hace click en el **elemento2** ocurrirá lo siguiente:

1. El evento de click comenzará en la fase de captura. El evento comprueba si hay algún ancestro del elemento2 que tenga un evento de onclick para la fase de captura (**true**).
2. El evento encuentra un elemento1.hacerAlgo1(), que ejecutará primero, pues está programado a **true**.
3. El evento viajará hacia el destino, pero no encontrará más eventos para la fase de captura. **Entonces el evento pasa a la fase de burbujeo, y ejecuta hacerAlgo2()**, el cuál hemos registrado para la fase de burbujeo (false).
4. **El evento viaja hacia arriba de nuevo y chequea si algún ancestro tiene programado un evento para la fase de burbujeo**. Este no será el caso, por lo que no hará nada más.

Para **detener la propagación del evento** en la fase de burbujeo, disponemos del método **stopPropagation()**. En la fase de captura es imposible detener la propagación.

<body onclick="alert(`No se propagó hasta aquí`)">

<button onclick="event.stopPropagation()">Haz clic</button>

</body>

Si un elemento tiene múltiples manejadores para un solo evento, aunque uno de ellos detenga la propagación, los demás aún se ejecutarán.

En otras palabras, event.stopPropagation() detiene la propagación hacia arriba, pero todos los manejadores en el elemento actual se ejecutarán.

Si queremos prevenir que la propagación de absolutamente **TODOS los manejadores del elemento actual**, necesitamos usar el método evento.stopImmediatePropagation().

Problemas con la cancelación de la propagación

No siempre es bueno detener la propagación. A veces event.stopPropagation() crea trampas ocultas que luego se convierten en problemas. Por ejemplo:

1. Creamos un menú anidado. Cada submenú maneja los clics en sus elementos y ejecuta *stopPropagation* para que el menu de arriba no se desencadene.
2. Luego decidimos atrapar los clic en toda la ventana, para seguir el rastro del comportamiento del usuario (donde hacen clic). Algunos sistemas de análisis hacen eso. Usualmente el código usa *document.addEventListener('click'…)* para atrapar todos los clics.
3. Nuestro análisis no funcionará sobre el área dónde los clics son detenidos por *stopPropagation*. Tristemente, tenemos una “zona muerta”.

Usualmente no hay una necesidad real para prevenir la propagación, pero una tarea que aparentemente lo requiera puede ser resuelta por otros medios. Uno de ellos es usar eventos personalizados, cubriremos eso más tarde.

Por otra parte, También podemos escribir nuestros datos en el objeto event en un manejador y leerlo en otro, para así poder pasar información sobre el proceso de abajo a los manejadores en los padres.

### 3.4.1.- Actualización de contenidos: fases de captura y propagación.

Hay otra fase en el procesamiento de eventos llamada “captura”. Es raro usarla en código real, pero a veces puede ser útil.

**El estándar de**[**eventos del DOM**](https://www.w3.org/TR/DOM-Level-3-Events/)**describe 3 fases de la propagación de eventos**:

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

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

elem.addEventListener(..., {capture: true})

// o solamente "true". Es una forma más corta de {capture: true}

elem.addEventListener(..., true)

Normalmente, por defecto, si no indicamos nada, el manejador es preparado para la fase de propagación (false por defecto). Si indicamos que debe ser atrapado en la fase de captura, será true.

Hay un propiedad **event.eventPhase que nos dice el número de fase** (captura=1, objetivo=2, propagación=3) **en la que el evento fue capturado**. Pero es raramente usada, ya que usualmente lo sabemos en el manejador.

Manejadores de eventos en la misma fase, asignados al mismo elemento con addEventListener

Se ejecutarán en el orden que fueron creados. Por ejemplo:

elem.addEventListener("click", e => alert(1)); // garantizado que se ejecutará primero

elem.addEventListener("click", e => alert(2));

evento.stopPropagation durante fase de captura

Tanto event.stopPropagation() como event.stopImmediatePropagation(), detienen no solo las capturas, sino también la propagación.

Conclusión de este apartado

La propagación y captura ponen los cimientos para “delegación de eventos”: un extremadamente poderoso patrón de manejo de eventos que se estudia en el siguiente capítulo.

## 3.5.- Delegación de eventos.

La idea es que si tenemos muchos elementos manejados de manera similar podemos, en lugar de asignar un manejador a cada uno de ellos, poner un único manejador a su ancestro común.

**Por ejemplo**: el diagrama Pa kua que refleja la antigua filosofía china:

Tabla

Descripción generada automáticamente con confianza baja

En lugar de asignar un manejador onclick a cada td, configuramos un manejador “atrapa-todo” en el elemento <table>. Este usará *evento.target* para obtener el elemento del clic y destacarlo.

let selectedTd;

table.onclick = function(event) {

let target = event.target; // ¿dónde fue el clic?

if (target.tagName != 'TD') return; // ¿no es un TD? No nos interesa

highlight(target); // destacarlo

};

function highlight(td) {

if (selectedTd) { // quitar cualquier celda destacada que hubiera antes

selectedTd.classList.remove('highlight');

}

selectedTd = td;

selectedTd.classList.add('highlight'); // y destacar el nuevo td

}

**Pero hay un inconveniente**: el clic puede ocurrir no sobre <td>, sino dentro de él.

<td>

<strong>Northwest</strong>

...

</td>

Con lo cual, si el clic ocurre dentro del strong, a evento.target lo que llega es el strong, y por tanto, nunca llegaría a ejecutarse la función highlight.

**Código mejorado**:

table.onclick = function(event) {

let td = event.target.closest('td'); // (1)

if (!td) return; // (2)

if (!table.contains(td)) return; // (3)

highlight(td); // (4)

};

1. El elemento **elem.closest(selector) devuelve el ancestro más cercano del evento.target que coincide con el selector**. En nuestro caso, buscamos <td> hacia arriba desde el elemento de origen.
2. **Si event.target no ocurrió dentro de algún <td>, el llamado retorna inmediatamente**, pues no hay nada que hacer.
3. **En caso de tablas anidadas**, event.target podría ser un <td>, pero fuera de la tabla actual. Entonces verificamos que realmente sea un <td> de nuestra tabla.
4. **Y, si es así, destacarla**.

Ejemplo de delegación: acciones en markup

Digamos que queremos hacer un menú con los botones “Save”, “Load”, “Search” y así. Y hay objetos con los métodos save, load, search… ¿Cómo asociarlos?

La primera idea podría ser asignar un controlador separado para cada botón. **Pero hay una solución más elegante. Podemos agregar un controlador para el menú completo** y un atributo data-action a los botones con el método a llamar. **El manejador lee el atributo y ejecuta el método. Puedes ver el siguiente ejemplo en funcionamiento**:

<div id="menu">

<button data-action="save">Save</button>

<button data-action="load">Load</button>

<button data-action="search">Search</button>

</div>

<script>

class Menu {

constructor(elem) {

this.\_elem = elem;

elem.onclick = this.onClick.bind(this); // (\*)

}

save() {

alert('saving');

}

load() {

alert('loading');

}

search() {

alert('searching');

}

onClick(event) {

let action = event.target.dataset.action;

if (action) {

this[action]();

}

};

}

new Menu(menu);

</script>

¿Qué ventajas nos ofrece la delegación aquí?

* No necesitamos escribir el código para asignar el manejador a cada botón. Simplemente hacer un método y ponerlo en el markup.
* La estructura HTML es flexible, podemos agregar y quitar botones en cualquier momento.
* Podríamos usar clases .action-save, .action-load, pero un atributo data-action es mejor semánticamente. Y podemos usarlo con reglas CSS también.

El patrón “comportamiento”

También podemos usar delegación de eventos para agregar “comportamiento” a los elementos de forma *declarativa*, con atributos y clases especiales. **El patrón tiene dos partes**:

1. Agregamos un atributo personalizado al elemento que describe su comportamiento.
2. Un manejador rastrea eventos del documento completo, y si un evento ocurre en un elemento con el atributo ejecuta la acción.

**Ejemplo de comportamiento: contador**: aquí el atributo data-counter agrega un comportamiento: “incrementar el valor con un clic” a los botones:

Counter: <input type="button" value="1" data-counter>

One more counter: <input type="button" value="2" data-counter>

<script>

document.addEventListener('click', function(event) {

if (event.target.dataset.counter != undefined) { // si el atributo existe...

event.target.value++;

}

});

</script>



Puede haber tantos atributos data-counter como queramos. Podemos agregar nuevos al HTML en cualquier momento. Usando delegación de eventos “extendimos” el HTML, agregando un atributo que describe un nuevo comportamiento.

**Advertencia**: cuando asignamos un manejador de evento al objeto document, usar siempre addEventListener, ya que document.on<evento>, lo que hace es que los manejadores nuevos sobreescriben a los viejos. Por otra parte, En proyectos reales es normal que haya muchos manejadores en document, asignados en diferentes partes del código.

**Ejemplo: comportamiento conmutador**:

<button data-toggle-id="subscribe-mail">

Show the subscription form

</button>

<form id="subscribe-mail" hidden>

Your mail: <input type="email">

</form>

<script>

document.addEventListener('click', function(event) {

let id = event.target.dataset.toggleId;

if (!id) return;

let elem = document.getElementById(id);

elem.hidden = !elem.hidden;

});

</script>

Imagen que contiene Texto

Descripción generada automáticamente

De esta forma, no hay necesidad de conocer JavaScript, basta con usar el atributo data-toggle-id. El manejador a nivel de documento hace el trabajo para cualquier elemento de la página.

Podemos combinar múltiples comportamientos en un único elemento también. El patrón “comportamiento” puede ser una alternativa a los mini-fragmentos de JavaScript.

Resumen

El algoritmo:

1. Pone un único manejador en el contenedor.
2. Dentro del manejador revisa el elemento de origen event.target.
3. Si el evento ocurrió dentro de un elemento que nos interesa, maneja el evento.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

## 3.6.- Acciones predeterminadas del navegador.

Por ejemplo:

* **Un clic en un enlace**: inicia la navegación a su URL.
* **Un clic en el botón de envío de un formulario** inicia su envío al servidor.
* **Al presionar un botón del ratón sobre un texto y moverlo**, se selecciona el texto.

Si manejamos un evento en JavaScript, **es posible que no queramos que suceda la acción correspondiente del navegador e implementar en cambio otro comportamiento**.

**Dos formas de prevenir el comportamiento por defecto del navegador**:

* **La forma principal es** utilizar el objeto event. Hay un **método event.preventDefault()**.
* **Si el controlador se asigna usando *on<event>*** (no por *addEventListener*), entonces **devolver false también funciona igual**.

<a href="/" onclick="return false">Haz clic aquí</a>

o

<a href="/" onclick="event.preventDefault()">aquí</a>

**Ejemplo: el menú**:

<ul id="menu" class="menu">

<li><a href="/html">HTML</a></li>

<li><a href="/javascript">JavaScript</a></li>

<li><a href="/css">CSS</a></li>

</ul>



**Los elementos del menú se implementan como enlaces HTML <a>**, no como botones <botón>. **Hay varias razones para hacerlo**, por ejemplo:

* **“clic derecho”** – “abrir en una nueva ventana”. Si usamos <button> o <span>, eso no funciona.
* **Los motores de búsqueda siguen los enlaces <a href="...">** durante la indexación.

menu.onclick = function(event) {

if (event.target.nodeName != 'A') return;

let href = event.target.getAttribute('href');

alert( href ); // ...se puede cargar desde el servidor, generación de interfaz de usuario, etc.

return false; // evitar la acción del navegador (no vaya a la URL)

};

Por cierto, usar la delegación de eventos aquí hace que nuestro menú sea muy flexible. Podemos agregar listas anidadas y diseñarlas usando CSS para “deslizarlas hacia abajo”.

Eventos de seguimiento

**Ciertos eventos fluyen unos a otros. Si evitamos el primer evento, no habrá segundo**.

<input value="Enfoque funciona" onfocus="this.value=''">

<input onmousedown="return false" onfocus="this.value=''" value="Haz clic en mí">

El focus funcionará en el primer input, pero no en el segundo, porque la acción del navegador se cancela en *mousedown*. El enfoque aún es posible si usamos otra forma de ingresar la entrada, por ejemplo, la tecla Tab.

### 3.6.1.- La opción de controlador “pasivo”.

**La opción opcional passive:true de addEventListener indica al navegador que el controlador no llamará a preventDefault()**. ¿Para qué podría ser necesario?

Hay algunos eventos como touchmove en dispositivos móviles (cuando el usuario mueve el dedo por la pantalla), que provocan el desplazamiento por defecto, pero ese desplazamiento se puede evitar usando preventDefault() en el controlador.

Entonces, cuando el navegador detecta tal evento, primero tiene que procesar todos los controladores, y luego, si no se llama a preventDefault en ninguna parte, puede continuar con el desplazamiento. Eso puede causar retrasos innecesarios y “movimientos de salto repentinos” en la interfaz de usuario.

**Las opciones passive: true le dicen al navegador que el controlador no va a cancelar el desplazamiento**. Entonces el navegador se desplaza de inmediato para brindar una experiencia con la máxima fluidez, y el evento se maneja de inmediato.

**Para algunos navegadores (Firefox, Chrome), passive es true por defecto** para los eventos touchstart y touchmove.

### 3.6.2.- evento.defaultPrevented.

La propiedad event.defaultPrevented es true si se impidió la acción predeterminada y false en caso contrario.

Hay un caso de uso interesante para ello. ¿Recuerdas que en el capítulo [Propagación y captura](https://es.javascript.info/bubbling-and-capturing) hablamos sobre event.stopPropagation() y por qué detener propagación es malo?

**A veces podemos usar event.defaultPrevented en su lugar, para señalar a otros controladores de eventos que el evento fue manejado**.

**Por ejemplo**: Por defecto, el navegador en el evento contextmenu (clic derecho del ratón) muestra un menú contextual con opciones estándar. Podemos prevenirlo y mostrar el nuestro, así:

<button>El clic derecho muestra el menú contextual del navegador</button>

<button oncontextmenu="alert('Dibuja nuestro menú'); return false">

El clic derecho muestra nuestro menú contextual

</button>

Si ahora implementamos un menú contextual para todo el documento, al hacer clic derecho, debería aparecer el menú contextual más cercano:

<p>Haz clic derecho aquí para el menú contextual del documento</p>

<button id="elem">Haz clic derecho aquí para el menú contextual del botón</button>

<script>

elem.oncontextmenu = function(event) {

event.preventDefault();

alert("Menú contextual del botón");

};

document.oncontextmenu = function(event) {

event.preventDefault();

alert("Menú contextual del documento");

};

</script>

El problema es que cuando hacemos clic en elem, obtenemos dos menús: el de nivel de botón y (el evento emerge) el menú de nivel de documento.

**Formas de arreglarlo**:

1. **Usar evento.stopPropagation() dentro del manejador de elem.oncontextmenu**: Pero el precio es alto. Siempre negamos el acceso a la información sobre los clics con el botón derecho del ratón para cualquier código externo, incluidos los contadores que recopilan estadísticas, etc. Eso es bastante imprudente.
2. **Verificar evento.defaultPrevented dentro del manejador de document.oncontextmenu**: permite saber si el evento por defecto ya ha sido prevenido (y por tanto, sustituido por una acción concreta.

<p>Haz clic con el botón derecho en el menú del documento (se agregó una marca de verificación para event.defaultPrevented)</p>

<button id="elem">Haz clic derecho para el menú de botones</button>

<script>

elem.oncontextmenu = function(event) {

event.preventDefault();

alert("Menú contextual del botón");

};

document.oncontextmenu = function(event) {

if (event.defaultPrevented) return;

event.preventDefault();

alert("Menú contextual del documento");

};

</script>

Ahora todo también funciona correctamente. Si tenemos elementos anidados, y cada uno de ellos tiene un menú contextual propio, eso también funcionaría. Solo asegúrate de buscar event.defaultPrevented en cada controlador de contextmenu.

**event.stopPropagation() y evento.preventDefault()**: Como podemos ver claramente, event.stopPropagation() y event.preventDefault() (también conocido como return false) son dos cosas diferentes. No están relacionados entre sí.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### 3.6.3.- Resumen.

Hay muchas acciones predeterminadas del navegador:

* **mousedown** – inicia la selección (mueva el ratón para seleccionar).
* **click** en <input type="checkbox"> – marca/desmarca el input.
* **submit** – dar clic en <input type="submit"> o presionar Enter dentro de un campo de formulario hace que suceda este evento y el navegador envía el formulario a continuación.
* **keydown** – presionar una tecla puede llevar a agregar un carácter a un campo u otras acciones.
* **contextmenu** – el evento ocurre con un clic derecho, la acción es mostrar el menú contextual del navegador.
* …hay mas…

Todas las acciones predeterminadas se pueden evitar si queremos manejar el evento exclusivamente mediante JavaScript.

**Para evitar una acción predeterminada, utiliza event.preventDefault()** o return false. El segundo método funciona solo para los controladores asignados con on<event>.

**La opción passive: true de addEventListener** le dice al navegador que la acción no se evitará. Eso es útil para algunos eventos móviles, como touchstart y touchmove, para decirle al navegador que no debe esperar a que todos los controladores terminen antes de desplazarse.

**Si se evitó la acción predeterminada**, el valor de **event.defaultPrevented** se convierte en true, de lo contrario, es false.

## 3.7.- Listado de eventos comunes.

Eventos del mouse

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

Eventos del teclado

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

Eventos del elemento form

* **submit** – cuando el visitante envía un <form>.
* **focus** – cuando el visitante se centra sobre un elemento, por ejemplo un <input>.

Evento del documento

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

Evento del CSS

* **transitionend** – cuando una animación CSS concluye.

Hay muchos más eventos. Entraremos en más detalles con eventos particulares en los siguientes capítulos.

# 4.- Envío y validación de formularios.

La validación de un formulario es un proceso que consiste en chequear un formulario y comprobar que todos sus datos han sido introducidos correctamente. Por ejemplo, **si tu formulario contiene un campo de texto en el que hay que escribir un e-mail, sería interesante comprobar si ese e-mail está escrito correctamente**.

**Hay dos métodos principales de validación de formularios**: en el lado del **servidor** (usando scripts CGI, PHP, ASP, etc.) y en el lado del **cliente** (generalmente usando JavaScript).

* **La validación en el lado del servidor es más segura**, pero a veces también es más complicada de programar.
* La **validación en el lado del cliente es más fácil y más rápida de hacer**.

La **idea general** que se persigue al validar un formulario, es **que cuando se envíen los datos al servidor, éstos vayan correctamente validados**.

Hay **dos formas de programar la validación** en JavaScript:

* Podremos hacerlo a medida que vamos metiendo datos en el formulario, por ejemplo campo a campo
* O cuando se pulse el botón de envío del formulario.

**JavaScript** añade a tus formularios **dos características muy interesantes**:

* JavaScript te permite **examinar y validar las entradas de usuario directamente, en el lado del cliente**.
* JavaScript **te permite dar mensajes instantáneos**, con información de la entrada del usuario.

**La validación de datos** del usuario **en la entrada**, generalmente **suele fallar en alguna de las 3 siguientes categorías**:

* **Existencia**: comprueba cuando existe o no un valor.
* **Numérica**: que la información contenga solamente valores numéricos.
* **Patrones**: comprueba que los datos sigan un determinado patrón, con el formato de un e-mail, una fecha, un nº de teléfono, un nº de la seguridad social, etc.

JavaScript también **se puede utilizar para modificar los elementos de un formulario**, basándose en los datos introducidos por el usuario: tal como cubrir un campo de selección con una lista de nombres de ciudades, cuando una determinada provincia está seleccionada, etc.

Al usar JavaScript, no debes olvidar de que existe la posibilidad de que el usuario desactive JavaScript en su navegador, por lo que JavaScript no debería ser una dependencia en la acción de envío de datos desde un formulario.

"Acuérdate de que JavaScript está para mejorar, no para reemplazar".

La validación de un formulario en el lado del cliente puede ahorrar algunas idas y vueltas a la hora de enviar los datos, pero aún así, tendrás que realizar la validación de datos en el servidor, puesto que es allí realmente donde se van a almacenar esos datos y el origen de los mismos puede venir por cauces que no hemos programado.

## 4.1.- Ejemplo sencillo de validación de un formulario.

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>DWEC05 - Validación de un Formulario</title>

<script type="text/javascript">

const validar = () => {

let edad = document.getElementById('edad').value;

if ( edad >= 18 && confirm("¿Deseas enviar el formulario?"))

return true;

else

{

alert('Lo siento, pero no eres mayor de edad.')

this.event.preventDefault();

return false;

}

}

</script>

<style type="text/css">

label{

width: 150px;

float:left;

margin-bottom:5px;

}

input, select {

width:150px;

float:left;

margin-bottom:5px;

}

fieldset{

background:#CCFF99;

width:350px;

}

.error{

border: solid 2px #FF0000;

}

</style>

</head>

<body>

<fieldset>

<legend>DWEC05 - Validación de un Campo -</legend>

<form name="formulario" id="formulario" action="http://www.google.es" method="get">

<label for="edad">Edad:</label>

<input name="edad" type="text" id="edad" maxlength="3" />

<input type="reset" name="limpiar" id="limpiar" value="Limpiar" />

<input type="submit" name="enviar" id="enviar" value="Enviar" onClick="validar()"/>

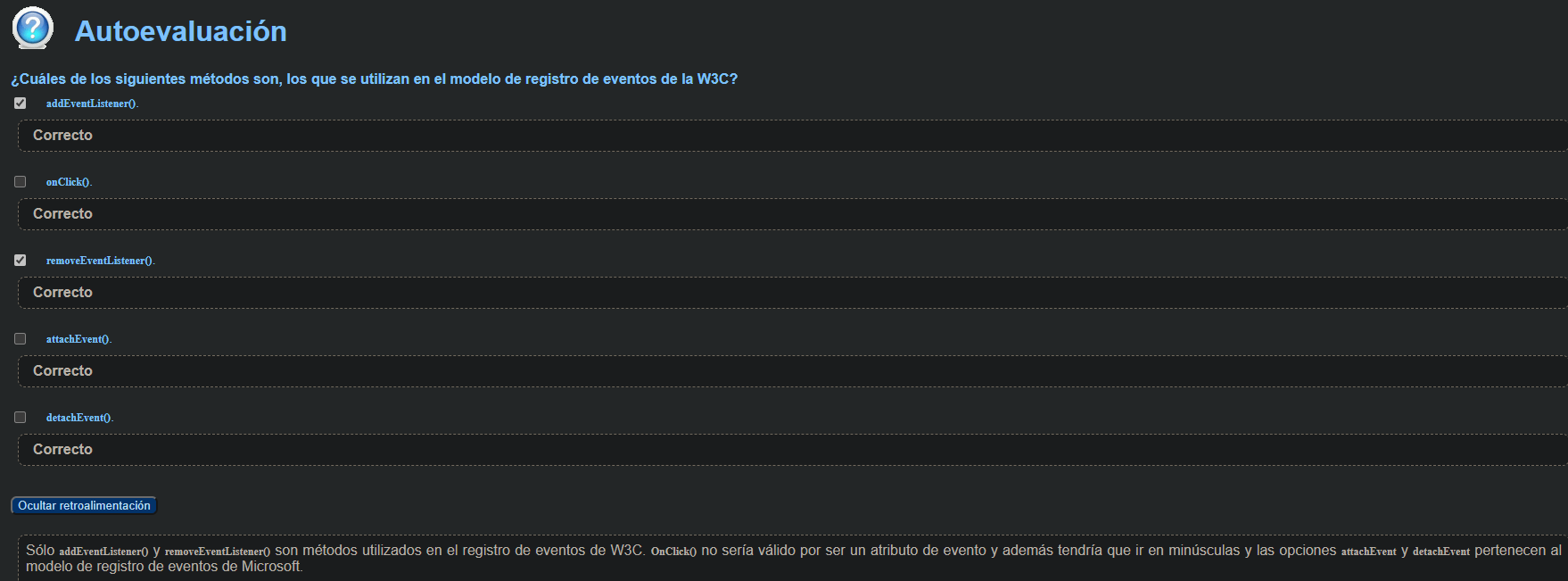
</form>

</fieldset>

</body>

</html>

**Reflexiona**: Te recomiendo que también veas el vídeo en el que se realiza la validación de un formulario, siguiendo el modelo de registro de eventos en línea. Aunque este modelo se utiliza cada vez menos, todavía te encontrarás muchísimos ejemplos que hacen uso del mismo.



## 4.2.- Funcionamiento del evento y método submit.

El evento submit se activa cuando el formulario es enviado, normalmente se utiliza para validar el formulario antes de ser enviado al servidor o bien para abortar el envío y procesarlo con JavaScript.

El método form.submit() permite iniciar el envío del formulario mediante JavaScript. Podemos utilizarlo para crear y enviar nuestros propios formularios al servidor.

### 4.2.1.- Evento: submit.

Un formulario puede enviarse de dos maneras:

1. Haciendo clic en <input type=”submit”> o en <input type”image”>
2. Pulsando la Tecla Enter en un campo del formulario.

El handler puede comprobar los datos, y si hay errores, mostrarlos e invocar event.preventDefault(), entonces el formulario no será enviado al servidor.

Relación entre submit y click

Cuando un formulario es enviado utilizando *Enter* en un campo tipo texto, un evento *click* se genera en el <input type=”submit”>.

Muy curioso, dado que no hubo ningún click en absoluto. Aquí esta la demo:

<form onsubmit="return false">

<input type="text" size="30" value="Sitúa el cursor aquí y pulsa Enter">

<input type="submit" value="Submit" onclick="alert('click')">

</form>

### 4.2.2.- Método submit.

**Para enviar un formulario al servidor manualmente, podemos usar form.submit(). Entonces el evento submit no será generado**. Se asume que si el programador llama form.submit(), entonces el script ya realizó todo el procesamiento relacionado.

A veces es usado para crear y enviar un formulario manualmente, como en este ejemplo:

let form = document.createElement('form');

form.action = 'https://google.com/search';

form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// el formulario debe estar en el document para poder enviarlo

document.body.append(form);

form.submit();

## 4.2.- Propiedades y métodos de formularios.

Los formularios del documento son miembros de la colección especial document.forms. Es una colección asociativa, de manera que **podemos usar el índice para localizar el formulario, o bien su nombre** (name).

document.forms.my; // el formulario con name="my"

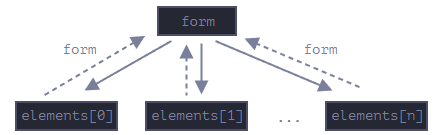
document.forms[0]; // el primer formulario en el documento

En ese caso form.elements[name] es una *colección*

|  |
| --- |
| **Radio buttons:** |
| Cuando tenemos un formulario, cualquier elemento se encuentra disponible en la colección nombrada.  <form name="my">  <input name="one" value="1">  <input name="two" value="2">  </form>    <script>  // obtención del formulario  let form = document.forms.my; // elemento <form name="my">    // get the element  let elem = form.elements.one; // elemento <input name="one">    alert(elem.value); // 1  </script>  Puede haber múltiples elementos con el mismo nombre. Esto es típico en el caso de los botones de radio y checkboxes. |
| **Fieldsets como “sub-formularios”** |
| Un formulario puede tener uno o varios elementos <fieldset> dentro. Estos también tienen la propiedad elements que lista los controles del formulario dentro de ellos. |
| **Notación corta: form.name** |
| En lugar de usar form.elements.nombreElemento, podemos acceder a los elementos del formulario usando la siguiente notación:  form[index/name].  **Ejemplo**: en lugar de form.elements.login, podemos escribir form.login.  <form id="form">  <input name="login">  </form>    <script>  alert(form.elements.login == form.login); // true, el mismo <input>    form.login.name = "username"; // cambiamos el nombre el <input>    // form.elements actualiza el nombre:  alert(form.elements.login); // undefined  alert(form.elements.username); // input    // form permite ambos nombres: el nuevo y el viejo  alert(form.username == form.login); // true  </script>  Esto usualmente no es un problema, porque raramente se cambian los nombres de los elementos de un formulario. |

### 4.2.1.- Referencia inversa: element.form.

Para cualquier elemento, el formulario está disponible como element.form. Así que un formulario referencia todos los elementos, y los elementos referencian el formulario.



### 4.2.2.- Elementos del formulario.

|  |  |
| --- | --- |
| Elementos del formulario | Propiedades más importantes accesibles desde JavaScript |
| Input y textarea | input.value = "New value";  textarea.value = "New text";    input.checked = true; // para checkboxes o radios |
| Usa textarea.value, no textarea.innerHTML. Ya que el innerHTML solo guarda el HTML que había inicialmente en la página, no su valor actual. |
| Select y option | **Un** **elemento <select> tiene 3 propiedades importantes**:   1. select.options – la colección de subelementos <option>, 2. select.value – el valor del <option> seleccionado actualmente, y 3. select.selectedIndex – el número del <option> seleccionado actualmente. |
| Existen **tres formas diferentes de asignar un valor** para un elemento <select>:  <select id="select">  <option value="apple">Apple</option>  <option value="pear">Pear</option>  <option value="banana">Banana</option>  </select>    <script>  // las tres líneas hacen lo mismo  select.options[2].selected = true;  select.selectedIndex = 2;  select.value = 'banana';  // Recuerda que las opciones comienzan en cero, así que index 2 significa la tercera opción.  </script> |
| Si ponemos el atributo multiple a “select”, tenemos un menú como este:    En este caso, necesitaríamos usar la primera forma, es decir usar la propiedad selected de los subelementos <option> |
| **Podemos obtener la colección de opciones**, e incluso, filtrarlos únicamente por aquellos que estén seleccionados:  <select id="select" multiple>  <option value="blues" selected>Blues</option>  <option value="rock" selected>Rock</option>  <option value="classic">Classic</option>  </select>    <script>  // obtener todos los valores seleccionados del multi-select  let selected = Array.from(select.options)  .filter(option => option.selected)  .map(option => option.value);    alert(selected); // blues,rock  </script>  [Especificación complete del element <select>](https://html.spec.whatwg.org/multipage/forms.html#the-select-element). |
| new Option | **Existe una sintaxis muy corta para crear elementos <option>**:  option = new Option(text, value, defaultSelected, selected);  **En cualquier caso, esta sintaxis es opcional**. Se puede seguir utilizando document.createElement(‘option’) y luego asignar atributos manualmente. Pero es mucho más cómoda esta otra sintaxis.   * **text** – el texto dentro del option, * **value** – el valor del option, * **defaultSelected** – si es true, entonces se le crea el atributo HTML selected, * **selected** – si es true, el option se selecciona.   La diferencia entre defaultSelected y selected es que defaultSelected asigna el atributo HTML, el que podemos obtener usando option.getAttribute('selected'), mientras que selected hace que el option esté o no seleccionado.  **En la práctica, uno debería usualmente establecer *ambos* valores en true o false**. |
| **Por defecto**, si no se indica en el constructor, **serán false ambos parámetros**. **Por ejemplo**:  let option = new Option("Text", "value");  // crea <option value="value">Text</option>  El mismo elemento, pero seleccionado:  let option = new Option("Text", "value", true, true); |
|  | **Los elementos Option tienen propiedades**:   * option.selected (es el option seleccionado). * option.index (nº del option respecto a los demás en su select). * option.text (contenido del option visto por el visitante). |

## 4.3.- Enfocado: enfoque/desenfoque.

**Un elemento se enfoca cuando el usuario hace click sobre él o al pulsar Tab en el teclado**. **Existen también un atributo autofocus** de HTML que enfoca un elemento por defecto cuando una página carga, y otros medios de conseguir el enfoque.

* **Enfocarse sobre un elemento** significa “prepárate para aceptar estos datos”.
* **Perder el foco** significa que “los datos ya han sido introducidos”, entonces podemos correr el código para validarlos antes de enviarlos al servidor.

El elemento que en el momento tiene el foco está disponible como ***document.activeElement***.

### 4.3.1.- Eventos focus/blur.

Pongamos un ejemplo:

* **El manejador *blur*** comprueba si se ha introducido un correo, en caso contrario, muestra un error (por ejemplo, aplicando un estilo CSS que establezca el color de la fuente a rojo).
* **El manejador *focus*** esconde el mensaje de error (en *blur* se volverá a comprobar).

<style>

.invalid { border-color: red; }

#error { color: red }

</style>

Su correo por favor: <input type="email" id="input">

<div id="error"></div>

<script>

input.onblur = function() {

if (!input.value.includes('@')) { // not email

input.classList.add('invalid');

error.innerHTML = 'Por favor introduzca un correo válido.'

}

};

input.onfocus = function() {

if (this.classList.contains('invalid')) {

// quitar la indicación "error", porque el usuario quiere reintroducir algo

this.classList.remove('invalid');

error.innerHTML = "";

}

};

</script>



El **HTML** actual **nos permite efectuar diversas validaciones utilizando atributos de entrada**: *required*, *pattern*, etc. Y muchas veces son todo lo que necesitamos. JavaScript puede ser utilizado cuando queremos más flexibilidad. También podríamos enviar automáticamente el valor modificado al servidor si es correcto.

### 4.3.2.- Métodos focus/blur.

Los métodos elem.focus() y elem.blur() ponen/quitan el foco sobre el elemento. La idea es que Si introducimos algo en la entrada y luego intentamos pulsar Tab o hacer click fuera del <input>, entonces onblur lo vuelve a enfocar.

Texto

Descripción generada automáticamente



**La pega**: Funciona en todos los navegadores excepto Firefox ([bug](https://bugzilla.mozilla.org/show_bug.cgi?id=53579)).

Por favor tened en cuenta que **no podemos “prevenir perder el foco”** llamando a event.preventDefault() en onblur, porque onblur funciona *después* de que el elemento perdió el foco.

En la práctica, el usuario podría querer llenar otros campos primero, por lo que lo suyo es limitarnos simplemente a mostrar errores al usuario, no evitar que siga adelante al llenar nuestro formulario.

Pérdida de foco iniciada por JavaScript

Hay varias maneras de perder el foco:

* Que el usuario salte de un campo hacia a otro, clique en otra parte…
* **Un alert** traslada el foco hacia sí mismo, lo que causa la pérdida de foco sobre el elemento (evento blur). Y cuando el alert es cerrado, el foco vuelve (evento focus).
* **Si un elemento es eliminado del DOM, también causa pérdida de foco**. Si es reinsertado el foco no vuelve.

Estas situaciones a veces causan que los manejadores focus/blur no funcionen adecuadamente y se activen cuando no son necesarios.

Es recomendable tener cuidado al utilizar estos eventos. Si queremos monitorear pérdidas de foco iniciadas por el usuario deberíamos evitar causarlas nosotros mismos.

### 4.3.3.- Permitir enfocado sobre cualquier elemento: tabIndex.

Por defecto, muchos elementos no permiten enfoque.

**La lista varía un poco entre navegadores, pero una cosa es siempre cierta: focus/blur está garantizado para elementos con los que el visitante puede interactuar**: <button>, <input>, <select>, <a>, etc.

En cambio, elementos que existen para formatear algo, tales como <div>, <span>, <table>, **por defecto no son posibles de enfocar ni de capturas eventos focus/blur** (ya que no se desencadenan). **Esto puede ser modificado usando el atributo HTML tabindex.**

**Cualquier elemento se vuelve enfocable si contiene tabindex**. El valor del atributo es el número de orden del elemento cuando Tab (o algo similar) es utilizado para cambiar entre ellos.

**El orden de cambio es el siguiente**: los elementos con tabindex de valor “1” y mayores tienen prioridad (en el orden tabindex) y después los elementos sin tabindex. Los elementos sin *tabindex* van cambiando en el orden del código fuente del documento.

**Existen dos valores especiales**:

* **tabindex="0"** incluye al elemento entre los que carecen de tabindex.
  + Esto es, cuando cambiamos entre elementos, elementos con tabindex="0" van después de elementos con tabindex ≥ "1".
  + **Habitualmente se utiliza para hacer que un elemento sea enfocable** y a la vez mantener intacto el orden de cambio por defecto
* **tabindex=”-1”**: restringe el focus a la tecla Tab, para que solo se pueda enfocar el elemento a través de código (es decir, solo con elem.focus, o bien hacer clic, pero no vale usar la tecla Tab).

**El orden de prioridad de tabIndex es el siguiente**: 1 - 2 - 0. Normalmente, <li> no admite enfocado, pero tabindex lo habilita, junto con eventos y estilado con :focus

|  |
| --- |
| **La propiedad elem.tabIndex también funciona**:  Podemos añadir *tabIndex* desde JavaScript utilizando la propiedad elem.tabIndex. Se consigue el mismo resultado. |

### 4.3.4.- Delegación: focusin/focusout.

Los eventos focus y blur no se propagan. **Por ejemplo**:

<!-- enfocando en el formulario -- añadir la clase -->

<form onfocus="this.className='focused'">

<input type="text" name="name" value="Name">

<input type="text" name="surname" value="Surname">

</form>

<style> .focused { outline: 1px solid red; } </style>

Esto no funcionará, porque cuando el usuario enfoca sobre un input text, el evento “focus” se dispara solamente sobre esa entrada y no se propaga, por lo que form.onfocus nunca se dispara.

**Existen dos soluciones**:

1. Existe una peculiar característica histórica: focus/blur no se propagan hacia arriba, pero lo hacen hacia abajo en la fase de captura.

<form id="form">

<input type="text" name="name" value="Name">

<input type="text" name="surname" value="Surname">

</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

// pon el manejador en fase de captura (último argumento "verdadero")

form.addEventListener("focus", () => form.classList.add('focused'), true);

form.addEventListener("blur", () => form.classList.remove('focused'), true);

</script>

1. Existen los eventos ***focusin*** y ***focusout***, exactamente iguales a focus/blur, pero estos sí se propagan.
   1. Hay que tener en cuenta que han de asignarse utilizando elem.addEventListener, no on<event>.

<form id="form">

<input type="text" name="name" value="Name">

<input type="text" name="surname" value="Surname">

</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

form.addEventListener("focusin", () => form.classList.add('focused'));

form.addEventListener("focusout", () => form.classList.remove('focused'));

</script>

## 4.4.- Eventos: change, input, cut, copy, paste.

### 4.4.1.- Evento: change.

El **evento change** se activa cuando el elemento finaliza un cambio.

* **Para ingreso de texto** significa que el evento ocurre cuando se pierde foco en el elemento. Es decir, tú en un input text, modificas su contenido, escribiendo o borrando, y al perder el foco pinchando en cualquier otra parte, se desencadena el evento.

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

* **Para otros elementos**: select, input type=checkbox/radio: se dispara inmediatamente después de cambiar la opción seleccionada:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### 4.4.2.- Evento: input.

El evento **input** **se dispara cada vez que un valor es modificado por el usuario**.

A diferencia de los eventos de teclado, ocurre con el cambio a cualquier valor, incluso aquellos que no involucran acciones de teclado: copiar/pegar con el mouse o usar reconocimiento de voz para dictar texto.



**Dejar claro dos cosas**:

* **El evento *input* no se activa con entradas de teclado u otras acciones que no involucren modificar un valor**, por ejemplo, presionar las flechas de dirección ⇦ ⇨ mientras se está en el input.
* El evento input se dispara después de que el valor es modificado, por lo que **no podemos usar evento.preventDefault()**, aquí, es demasiado tarde y no tendría efecto.

### 4.4.3.- Eventos: cut, copy, paste.

**Estos eventos ocurren al cortar/copiar/pegar un valor**. Estos pertenecen a la clase [ClipboardEvent](https://www.w3.org/TR/clipboard-apis/" \l "clipboard-event-interfaces) y dan acceso a los datos cortados/copiados/pegados.

También podemos usar event.preventDefault() para cancelar la acción y que nada sea cortado/copiado/pegado.

**El siguiente código evita también todo evento cut/copy/paste** y muestra qué es lo que estamos intentando cortar/copiar/pegar:

<input type="text" id="input">

<script>

input.onpaste = function(event) {

alert("paste: " + event.clipboardData.getData('text/plain'));

event.preventDefault();

};

input.oncut = input.oncopy = function(event) {

alert(event.type + '-' + document.getSelection());

event.preventDefault();

};

</script>

**Notar dos cosas**:

* Dentro de los manejadores cut y copy, llamar a event.clipboardData.getData(...) devuelve un string vacío. Esto es porque el dato no está en el portapapeles aún.
  + En su lugar, hemos utilizado document.getSelection() para obtener el texto seleccionado. Obtener más detalles acerca de selección en el artículo [Selection y Range](https://es.javascript.info/selection-range).
* Y si usamos event.preventDefault() no será copiado en absoluto.

**No solo es posible copiar/pegar texto, sino cualquier cosa**. Por ejemplo, podemos copiar un archivo en el gestor de archivos del SO y pegarlo. **Esto es porque clipboardData implementa la interfaz DataTransfer, usada comúnmente para “arrastrar y soltar” y “copiar y pegar”**. Ahora esto está fuera de nuestro objetivo, pero puedes encontrar sus métodos [en la especificación DataTransfer](https://html.spec.whatwg.org/multipage/dnd.html#the-datatransfer-interface).

Hay además una API asincrónica adicional para acceso al portapapeles: navigator.clipboard. Más en la especificación [Clipboard API and events](https://www.w3.org/TR/clipboard-apis/), [no soportado en Firefox](https://caniuse.com/async-clipboard).

#### 4.4.3.1.- Restricciones de seguridad.

El portapapeles es algo a nivel “global” del SO. Por ello la mayoría de los navegadores dan acceso al portapapeles únicamente bajo determinadas acciones del usuario, como copiar y pegar.

**Está prohibido generar eventos “personalizados” del portapapeles con dispatchEvent** en todos los navegadores excepto Firefox. Incluso si logramos enviar tal evento, la especificación establece que tal evento “sintético” no debe brindar acceso al portapapeles. Incluso **si alguien decide guardar event.clipboardData en un manejador de evento para accederlo luego, esto no funcionará**.

Por otro lado, [navigator.clipboard](https://www.w3.org/TR/clipboard-apis/" \l "h-navigator-clipboard) es una API más reciente, pensada para el uso en cualquier contexto. Esta pide autorización al usuario cuando la necesita.

### 4.4.4.- Resumen.

|  |  |  |
| --- | --- | --- |
| Evento | Descripción | Especiales |
| change | Un valor fue cambiado. | En ingreso de texto, se dispara cuando el elemento pierde el foco |
| input | Cada cambio de entrada de texto | Se dispara de inmediato con cada cambio, a diferencia de change. |
| cut/copy/paste | Acciones cortar/copiar/pegar | La acción puede ser cancelada. La propiedad event.clipboardData brinda acceso al portapeles. |

# 5.- Expresiones regulares y objetos RegExp.

**Caso práctico**: hay dos formas de validar que los datos sigan un formato o estructura en un campo de texto.

* **Forma lenta**: consiste en trabajar con esa cadena de texto, y con los métodos del objeto String comprobar que se cumple el formato solicitado. Implica bastante lógica de programación.
* **Forma rápida y recomendada**: expresiones regulares. De esta forma, se podría realizar cualquier validación en dos o tres líneas prácticamente.
  + Solo es necesario aprender como crear una expresión regular y los caracteres utilizados en expresiones regulares.

Las expresiones regulares son patrones de búsqueda, que se pueden utilizar para encontrar texto que coincida con el patrón especificado.

**Ejemplo de búsqueda de una cadena de texto sin usar expresiones regulares**:

let texto = "La linea de alta velocidad llegará pronto a toda España,";

let subcadena = "velocidad";

let indice = texto.indexOf(subcadena); // devuelve 17, índice de donde se encuentra la subcadena

if (indice != -1)

// correcto, se ha encontrado la subcadena

....

Este código funciona porque estamos buscando una subcadena de texto exacta. **¿Qué pasaría si hiciéramos una búsqueda más general?** Por ejemplo, si quisiéramos buscar la cadena “car” en textos como “cartón, “bicarbonato”, “practicar”, …

**Cuando estamos buscando cadenas que cumplen un patrón en lugar de una cadena exacta, necesitaremos usar expresiones regulares**.

En JavaScript las expresiones regulares se gestionan a través del objeto **RegExp**.

Para crear un literal del tipo **RegExp**, tendrás que usar la siguiente sintaxis:

let expresion = /expresión regular/;

**La expresión regular está contenida entre la barras** **/**, y fíjate que no lleva comillas. Las comillas sólo se pondrán en la expresión regular, cuando formen parte del patrón en si mismo

Las **expresiones regulares están hechas de caracteres**, solos o **en** **combinación con caracteres especiales**, para búsquedas más complejas. **Por ejemplo**: la siguiente es una expresión regular que realiza una búsqueda que contenga las palabras **Aloe Vera**, en ese orden y separadas por uno o más espacios en medio:

let expresion = /Aloe\s+Vera/;

En este ejemplo, los caracteres especiales son:

* **Barra invertida (\)**: tiene dos efectos:
  + O bien **se utiliza con un carácter regular**, para **indicar que se trata de un carácter especial**.
  + O **se usa con un carácter especial**, tales como el signo más (**+**), **para indicar que el carácter debe ser tratado literalmente**.
  + En este caso, la barra invertida se utiliza con “s”, que transforma la letra *s* en un carácter especial indicando un espacio en blanco, un tabulador, un salto de línea, etc.
* **El símbolo +**: indica que el carácter anterior puede aparecer una o más veces.

## 5.1.- Caracteres especiales en expresiones regulares.

Caracteres más utilizados para la construcción de Expresiones Regulares:

|  |  |  |
| --- | --- | --- |
| Carácter | Coincidencias | Ejemplo |
| ^ | Al inicio de una cadena | /^Esto/ |
| Coincidencia en "Esto es...". |
| $ | Al final de la cadena | /final$/ |
| Coincidencia en "Esto es el final". |
| \* | Coincide 0 o más veces | /se\*/ |
| Que la "e" aparezca 0 o más veces: "seeee" y también "se". |
| ? | Coincide 0 o 1 vez | /ap?/ |
| Que la p aparezca 0 o 1 vez: "apple" y "and". |
| + | Coincide 1 o más veces | /ap+/ |
| Que la "p" aparezca 1 o más veces: "apple" pero no "and". |
| {n} | Coincide exactamente n veces | /ap{2}/ |
| Que la "p" aparezca exactamente 2 veces: "apple" pero no "apabullante". |
| {n,} | Coincide n o más veces | /ap{2,}/ |
| Que la "p" aparezca 2 o más veces: "apple" y "appple" pero no en "apabullante". |
| {n,m} | Coincide al menos n, y máximo m veces | /ap{2,4}/ |
| Que la "p" aparezca al menos 2 veces y como máximo 4 veces: "apppppple" (encontrará 4 "p"). |
| . | Cualquier carácter excepto nueva línea | /a.e/ |
| Que aparezca cualquier carácter, excepto nueva línea entre la a y la e: "ape" y "axe". |
| […] | Cualquier carácter entre corchetes | /a[px]e/ |
| Que aparezca alguno de los caracteres "p" o "x" entre la a y la e: "ape", "axe", pero no "ale". |
| [^…] | Cualquier carácter excepto los que están entre corchetes. | /a[^px]/ |
| Que aparezca cualquier carácter excepto la “p” o “x” después de la letra a: “ale |
| \b | Coincide con el inicio de una palabra | /\bno/ |
| Que "no" esté al comienzo de una palabra: "novedad". |
| \B | Coincide al final de una palabra | **/\Bno/** |
| Que "no" esté al final de una palabra: "este invierno" ("no" de "invierno"). |
| \d | Dígitos del 0 al 9 | /\d{3}/ |
| Que aparezcan exactamente 3 dígitos: "Ahora en 456". |
| \D | Cualquier carácter que no sea un dígito | /\D{2,4}/ |
| Que aparezcan mínimo 2 y máximo 4 caracteres que no sean dígitos: encontrará la cadena "Ahor" en "Ahora en 456". |
| \w | Coincide con caracteres del tipo (letras, dígitos, subrayados) | /\w/ |
| Que aparezca un carácter (letra, dígito o subrayado): "J" en "JavaScript". |
| \W | Coincide con caracteres que no sean (letras, dígitos, subrayados) | /\W/ |
| Que aparezca un carácter (que no sea letra, dígito o subrayado): "%" en "100%". |
| \a | Coincide con una nueva línea |  |
| \s | Coincide con un espacio en blanco |  |
| \S | Coincide con un carácter que no es un espacio en blanco |  |
| \t | Un tabulador |  |
| (x) | Capturando paréntesis | Recuerda los caracteres. |
| \r | Un retorno de carro |  |
| ?=n | Cualquier cadena que esté seguida por la cadena n indicada después del igual | /la(?= mundo) |
| Hola mundo mundial. |

## 5.2.- El objeto RegExp.

El objeto **RegExp** es tanto un literal como un objeto de JavaScript, por lo que también se podrá crear usando un constructor:

let expresionRegular = new RegExp("Texto Expresión Regular");

¿Cuándo usar el literal o el objeto?

* La expresión **RegExp** **literal es compilada** cuando se ejecuta el script, por lo tanto, se recomienda **usar el literal cuando sabemos que la expresión no cambiará**. Una versión compilada es mucho más eficiente.
* **Usaremos el objeto, cuando sabemos que la expresión regular va a cambiar**, o cuando vamos a proporcionarla en tiempo de ejecución.

Propiedades del objeto RegExp

|  |  |
| --- | --- |
| Propiedad | Descripción |
| global | Especifica que sea utilizado el modificador "g". |
| ignoreCase | Especifica que sea utilizado el modificador "i". |
| lastIndex | El índice donde comenzar la siguiente búsqueda. |
| multiline | Especifica si el modificador "m" es utilizado. |
| source | El texto de la expresión regular **RegExp**. |

Métodos del objeto RegExp

|  |  |
| --- | --- |
| Método | Descripción |
| compile() | Compila una expresión regular. |
| exec() | Busca la coincidencia en una cadena. Devolverá la primera coincidencia. |
| test() | Busca la coincidencia en una cadena. Devolverá true o false. |

## 5.3.- Ejemplos de uso de expresiones regulares.

Validar un número de teléfono

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">

<title>Ejemplo de RegExp</title>

<script type="text/javascript">

const iniciar = () => {

document.getElementById("comprobar").onclick = comprobarTelefono;

}

const comprobarTelefono = () => {

let telefono = document.getElementById("telefono").value;

let patron = /^\d{9}$/;

if (telefono.match(patron))

alert('Teléfono Correcto!!!!');

else

alert('Teléfono INCORRECTO!!!!');

}

window.onload = iniciar;

</script>

</head>

<body>

<form name="formulario">

<label for="telefono">Teleono:</label>

<input type="text" name="telefono" id="telefono" />

<input type="button" name="comprobar" id="comprobar" value="Comprobar Formato" />

</form>

</body>

</html>

Validación de un número de DNI

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">

<title>Ejemplo de RegExp</title>

<script type="text/javascript">

const iniciar = () => {

document.getElementById("comprobar").onclick = comprobarDni;

}

const comprobarDni = () => {

let dni = document.getElementById("dni").value;

let patron = /^\d{8}[A-Z]$/;

if (dni.match(patron))

alert('DNI Correcto!!!!');

else

alert('DNI INCORRECTO!!!!');

}

window.onload = iniciar;

</script>

</head>

<body>

<form name="formulario">

<label for="dni">DNI:</label>

<input type="text" name="dni" id="dni" />

<input type="button" name="comprobar" id="comprobar" value="Comprobar Formato" />

</form>

</body>

</html>

**Debes conocer**: En el siguiente [enlace](https://regexr.com/) podrás validar las expresiones regulares que introduzcas y además te irá explicando cómo analiza dicha expresión regular. Te aconsejo su utilización ya que te va a permitir practicar con ellas y entenderlas mejor.

1. Se refiere a la especificidad con la que se define un nivel de detalle. [↑](#footnote-ref-1)