UNIDAD 4. Interacción con el usuario: eventos y formularios

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Definición de eventos 3
Introducción 3
Modelo de eventos en línea 4
Modelo de eventos tradicional 5
Modelo de eventos del W3C 6
El flujo de eventos 6
Añadir un manejador de eventos: .addEventListener() 6
Eliminar un manejador de eventos: .removeEventListener() 7
Eventos DOMContentLoaded y load 7
Manejar la información de un evento 7
Propiedades de event 8
Eventos del ratón 9
Eventos del teclado 10
Eventos HTML 11
Eventos delegados: *bubbling* 12
Validación de formularios 14
Introducción 14
Validación del formularios en el lado del cliente 14
Validación básica con JavaScript 15
Validación avanzada con HTML5 y JavaScript 16
Atributos de los elementos HTML 16
Pseudoclases CSS 17
Propiedades y métodos JavaScript del DOM 17
Expresiones regulares 18
Métodos para trabajar con expresiones regulares 19
Construcción de expresiones regulares 19
Almacenar datos en el equipo cliente 24
Cookies 24
Web Storage 25
Same Origin Policy 26

Definición de eventos

Introducción

En JavaScript se puede definir código que se ejecutará cuando se produzca un determinado evento. Los **eventos** se disparan cuando el usuario realiza alguna acción sobre la página web, como, por ejemplo: pulsar una tecla `keypress`, pulsar y soltar el botón del ratón `click`, pasar el ratón por encima de un elemento `mouseover`, finalizar la carga de la página `load`, etc. y tendrán funciones asociadas que permiten atender al evento.

Las acciones de un usuario pueden dar lugar a una sucesión de eventos. Por ejemplo, al pulsar sobre un botón de tipo `<input type="submit">` se desencadenan los eventos `mousedown`, `click`, `mouseup`, `submit`.

Más formalmente: un **evento** indica la ocurrencia de un hecho que es atendido por un manejador. Un **manejador de un evento** es un bloque de código o función asociada al evento que se ejecuta al ocurrir este.

Un programa JavaScript se guía por eventos (*event driven*). El script inicial debe configurar los primeros manejadores y después solo se ejecutarán los manejadores de los eventos que ocurren.

La gestión de eventos ha resultado ser una fuente de incompatibilidades entre los navegadores web, por lo que hay varios modos o modelos de asociar código a los eventos:

- **Modelo de eventos en línea**: se añade como un atributo en la etiqueta HTML: `<a href="…" onclick=acción>Enlace</a>`
- **Modelo de eventos tradicional**: los eventos son una propiedad de los elementos: `elementoDOM.onclick = acción;`
- **Modelo de eventos de Microsoft** (IE8 y anteriores) IE11 y posteriores no soportan este modelo: utiliza `elementoDOM.attachEvent("onclick", acción);`
- **Modelo de eventos del W3C**: utiliza `addEventListener("click", acción);`

Hoy en día **se recomienda usar solamente el modelo de eventos del W3C**, que se creó para estandarizar el manejo de eventos y, por lo tanto, eliminar la incompatibilidad con navegadores.

Modelo de eventos en línea

Este modelo se aprende a utilizar por su sencillez y necesidad de adaptar páginas web antiguas, pero **no es nada recomendable ya que NO separa HTML y JavaScript**.

Los elementos **HTML** tienen **atributos** que asocian eventos, por ejemplo: `onclick` (clic), `ondbclick` (doble clic). El nombre de esos atributos es `on` + nombre del evento. Como hemos visto, para el evento `click`, existe la propiedad `onclick`.

El **valor** que tendrán esos atributos asociados a eventos será código JavaScript, que se ejecutará al ocurrir el evento, es decir, será el **manejador del evento**:

- Pueden ser sentencias JavaScript:
  Ejemplo:
```
<input type="button" value="Haz clic y verás" onclick="console.log('Gracias por hacer clic');">
```

- Se puede utilizar la palabra reservada `this`, que hace referencia al objeto DOM del elemento al que está asociado el evento:
  Ejemplo:
```
<div onmouseover="this.style.borderColor='silver'"
onmouseout="this.style.borderColor='black'">...</div>
```

- Pueden ser llamadas a funciones:
  Ejemplo:
```
<input type="button" value="Haz clic y verás" onclick="muestraMensaje();">

function muestraMensaje() {
  console.log('Gracias por pinchar');
}
```
  En las funciones externas no es posible utilizar la variable `this` como en el caso anterior. Por eso es necesario pasar la variable `this` como parámetro a la función manejadora.
  Ejemplo:
```
<h3 onclick="cambiar(this)">
    Pulsa aquí para ver qué lo que ocurre
</h3>
<script>
function cambiar(elem) {
    elem.innerHTML = "JavaScript en atributo HTML y función externa";
}
</script>
```

- Se puede evitar que se ejecute la acción por defecto devolviendo `false`:
  Ej:
```
<a href="http://www.google.com" onclick="alert('Vamos a Google');
return false;">Pulsa aquí para ver qué se ejecuta</a>
```

Evento onload
Un evento especial es el evento `onload` del elemento `body`: ocurre cuando se ha cargado todo el documento HTML.

Modelo de eventos tradicional

Este modelo se creó para poder separar el código HTML y JavaScript, pero actualmente **no se recomienda su uso**. Lo veremos para identificarlo y si fuera necesario, adaptar páginas web antiguas al siguiente modelo que veremos.

Se basa en utilizar el **DOM** (modelo de objetos del documento). Cuando se carga una página web, se crea una estructura de objetos, denominada DOM. Cada elemento HTML se representa en dicha estructura, dependiendo del tipo de elemento, tendrá unas propiedades u otras.

Algunas de las propiedades (en realidad son métodos de los objetos), permiten enlazar funciones que se disparen cuando se produzcan eventos sobre los elementos del DOM. El nombre de la propiedad es `on` + nombre del evento.

Asociar el evento a su manejador
Sintaxis:

```
document.getElementById("…").onclick = función;
```

Es importante recordar que no se escriben paréntesis después del nombre de la función: si se escribieran, se ejecutaría la función. En este modelo, las funciones que definimos para responder a los eventos, se conocen como **manejadores de eventos semánticos**.

El primer ejemplo que vimos en el modelo de eventos en línea sería del siguiente modo:

```
function muestraMensaje() {
  console.log('Gracias por hacer clic');
}
document.getElementById("boton").onclick = muestraMensaje;

<input id="boton" type="button" value="Haz clic y verás">
```

Hemos de tener en cuenta que, con este modelo, al igual que con el anterior, solo se puede asignar una función a un evento. Si se asignaran dos funciones, solo se tendrá en cuenta la última que se asigne.

Evento onload
Un inconveniente de este método es que los manejadores se asignan mediante los métodos del objeto DOM, que solamente se pueden utilizar después de que la página se ha cargado completamente. Por tanto, para que la asignación de los manejadores no resulte errónea, es necesario asegurarse de que la página ya se ha cargado. Una forma de conseguirlo es utilizar el evento onload del objeto window.

Ejemplo:

```
window.onload = function() {
    document.getElementById("boton").onclick = muestraMensaje;
}
```

Eliminar el manejador de un evento
Se puede desenlazar una función de un evento. Basta con asignar el valor `null`.
Ejemplo:

```
document.getElementById("boton").onclick = null;
```

Modelo de eventos del W3C

Este es el modelo de eventos que vamos a estudiar más en profundidad, ya que es el que **es el más recomendable, y, por lo tanto, el que debemos utilizar**.

El flujo de eventos

Antes de estudiar el modo de asignar los manejadores de eventos, tenemos que entender el concepto **flujo de eventos** (*event flow*) de los navegadores. El flujo de eventos permite que varios elementos diferentes puedan responder a un mismo evento.

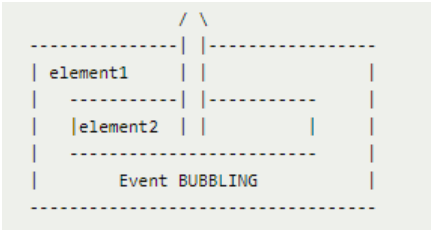
Por ejemplo: si se hace clic sobre un botón, los elementos HTML dentro de los cuales está el botón (body, div, etc.) también pueden responder al evento clic.

El orden en el que se ejecuten los eventos es lo que se conoce como flujo de eventos.

Event bubbling

Modelo desarrollado en los navegadores de Microsoft (IE).

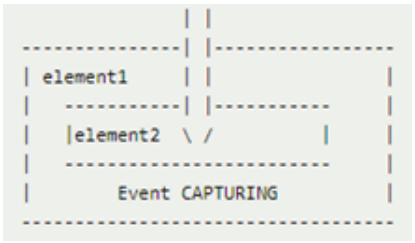
En este modelo de flujo de eventos, el orden que se sigue es desde el elemento más específico al menos específico.



Event capturing

Modelo desarrollado por Netscape.

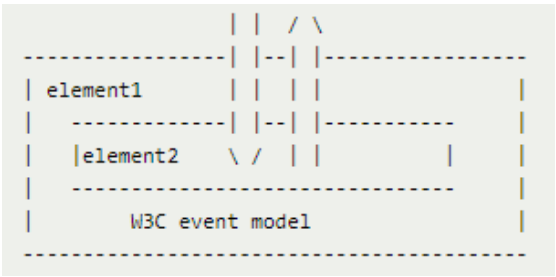
Es el mecanismo contrario al *event bubbling*: el orden es del menos específico al más específico.



Modelo del W3C

El W3C tomó una postura intermedia. Todo evento que tiene lugar en el modelo de eventos del W3C, es capturado primero hasta que alcanza el elemento destino (*capturing*), y después se transmite hacia arriba (*bubbling*).

El desarrollador web puede elegir si registrar el manejador de eventos en la fase de *capturing* o de *bubbling*.



Añadir un manejador de eventos: .addEventListener()

Sintaxis:

```
elemento.addEventListener("evento", miFunción [, useCapture]);
```

Parámetros:

- *evento*: "click", "mouseover", etc.
- *miFunción*: puede ser el nombre de una función, una función anónima o una función flecha (pero no una llamada a una función).
- *useCapture*: opcional; es una variable booleana; el valor por defecto es *false*.
  - Si es *true*, el manejador se emplea en la fase de *capture*.
  - Si es *false*, se asocia a la fase de *bubbling*.

Notas:

- Es posible asignar varios manejadores de eventos distintos a un evento sobre un elemento. En este modelo se puede, en los dos modelos anteriores, no.
- Uso de **`this`** dentro del manejador de eventos: es una referencia al elemento para el cual se está ejecutando el manejador. Tiene el mismo valor que la propiedad `currentTarget` del objeto de tipo evento que se pasa al manejador.

Eliminar un manejador de eventos: .removeEventListener()

Sintaxis:

```
elemento.removeEventListener("evento", miFunción [, useCapture]);
```

Parámetros:

Los mismos que para addEventListener().

Notas:

- Si se asocia una función a un flujo de eventos, esa función solo se puede desasociar en el mismo flujo de eventos (es decir, si en el `addEventListener()` el tercer parámetro es false, en el `removeEventListener()` también deberá ser false).

Ejemplo:

```
function muestraMensaje() {
    console.log("Has pulsado el ratón");
}
let d = document.getElementById("divPrincipal");
d.addEventListener("click", muestraMensaje, false);

//Más adelante se decide desasociar la función al evento
d.removeEventListener("click", muestraMensaje, false);
```

Eventos DOMContentLoaded y load

Como hemos ido viendo crear nuestros scripts, nos interesa que el código se ejecute cuando se termine de cargar la página completamente, o al menos, el árbol DOM.

Para ello, podemos utilizar el evento `DOMContentLoaded`. En `miFuncion` estará todo el código que queremos que se ejecute cuando se termine de carga el DOM.

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

El evento **`DOMContentLoaded`** indica cuando finaliza la carga de la página, aunque los demás recursos (imágenes, ficheros JavaScript o CSS, etc.) no hayan cargado todavía. El evento **`load`** ocurre en cambio cuando se ha cargado la página y todos sus recursos, por lo tanto, `DOMContentLoaded` ocurre antes que `load` y es preferible utilizarlo.

Manejar la información de un evento

El objeto ***event*** se crea automáticamente cuando se produce un evento y se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento.

El estándar DOM especifica que el objeto *event* es el único parámetro que se debe pasar a las funciones encargadas de procesar los eventos.

```
document.getElementById("…").addEventListener("evento", miFuncion);

function miFuncion(e){
  // e es un objeto de tipo event
}
```

COMPATIBILIDAD con IE

En IE, *event* no se pasa como parámetro al manejador de eventos, sino que es una propiedad del objeto *window*.

Una forma de conseguir la compatibilidad en los manejadores sería:

```
document.getElementById("…").addEventListener("evento", miFuncion);
function miFuncion(e){
  let evento = e || window.event;
  //si e es null, se asigna window.event a la variable evento
}
```

Propiedades de event

Algunas propiedades y métodos en los navegadores que siguen los estándares:

Propiedad	Tipo	Contenido
type	Cadena de texto	El nombre del evento.
timeStamp	Número	La fecha y hora en la que se ha producido el evento.
target	Element	El elemento que origina el evento.
currentTarget	Element	El elemento que es el objetivo del evento.
cancelable	Boolean	Indica si el evento se puede cancelar.
bubbles	Boolean	Indica si el evento pertenece al flujo de eventos de *bubbling*.
cancelBubble	Boolean	Indica si se ha detenido el flujo de eventos de tipo *bubbling.*

Método	Devuelve	Descripción
preventDefault()	`undefined`	Se emplea para cancelar la acción predefinida del evento.
stopPropagation()	`undefined`	Impide que se siga propagando el evento en las fases de *capturing* y *bubbling*.

Ejemplos:

1. Obtener el elemento que origina el evento (más recomendable que el uso de `this`):
   ```
   elEvento.target
   ```

COMPATIBILIDAD con IE
`Event.srcElement` es un alias (implementado en Internet Explorer) de Event.target, que es soportado por algunos navegadores por motivos de compatibilidad.

2. Obtener el tipo de evento:

```
elEvento.type
```

```
const procesaEvento = (elEvento)=> {
  if(elEvento.type === "click") {
    console.log("Has pulsado el ratón.");
  }
  else if(elEvento.type === "mouseover") {
    console.log("Has movido el ratón.");
  }
}

d.addEventListener("click", procesaEvento);
d.addEventListener("mouseover", procesaEvento);
```

3. Detener completamente la propagación del evento hacia arriba (*bubbling*) o hacia abajo (*capturing*):

```
elEvento.stopPropagation();
```

4. Impedir que se complete el comportamiento por defecto de un evento:

```
elEvento.preventDefault();
```

Eventos del ratón

Estos eventos están definidos para todos los elementos HTML.

Evento	Se produce
`click`	Al pulsar el botón izquierdo del ratón. También se produce cuando el foco de la aplicación está situado en un botón y se pulsa la tecla `ENTER`.
`dblclick`	Al pulsar dos veces el botón izquierdo del ratón.
`mousedown`	Al pulsar cualquier botón del ratón.
`mouseup`	Al soltar cualquier botón del ratón que haya sido pulsado.
`mouseover`	El ratón “entra” en el elemento.
`mouseout`	El ratón “sale” del elemento.
`mousemove`	Se produce (de forma continua) cuando el puntero del ratón se encuentra sobre un elemento.

Notas:

- Cuando se pulsa un botón del ratón, la secuencia de eventos que se produce es la siguiente:

  `mousedown >> mouseup >> click`

- Por tanto, la secuencia de eventos necesaria para llegar al doble clic llega a ser tan compleja como la siguiente:

  `mousedown >> mouseup >> click >> mousedown >> mouseup >> click >> dblclick`

Propiedades de objeto *event* específicas de los eventos de ratón

Propiedad	Tipo	Contenido
button (*)	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: 0. Ningún botón pulsado 1. Se ha pulsado el botón izquierdo 2. Se ha pulsado el botón derecho 3. A la vez el botón izquierdo y el derecho 4. Se ha pulsado el botón central 5. A la vez el botón izquierdo y el central 6. A la vez el botón derecho y el central 7. Se pulsan a la vez los 3 botones
clientX	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana.
clientY	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana.
detail	Número entero	El número de veces que se han pulsado los botones del ratón.
pageX	Número entero	Coordenada X de la posición del ratón respecto de la página.
pageY	Número entero	Coordenada Y de la posición del ratón respecto de la página.
screenX	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa.
screenY	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa.

(*) `button`: solo en los eventos `mousedown`, `mousemove`, `mouseout`, `mouseover` y `mouseup`

Eventos del teclado

Estos eventos se producen sobre los elementos de formulario y <body>.

Evento	Se produce
keydown	Al pulsar cualquier tecla. También se produce de forma continua si se mantiene pulsada la tecla.
keypress	Al pulsar una tecla correspondiente a un carácter alfanumérico (no se tienen en cuenta telas como `SHIFT`, `ALT`, etc.). También se produce de forma continua si se mantiene pulsada la tecla.
keyup	Al soltar cualquier tecla pulsada.

Propiedades del objeto *event* específicas de los eventos de teclado

Propiedad	Tipo	Contenido
altKey	Boolean	*true* si se ha pulsado la tecla `ALT` y *false* en otro caso.

Propiedad	Tipo	Contenido
ctrlKey	Boolean	*true* si se ha pulsado la tecla `CTRL` y *false* en otro caso.
shiftKey	Boolean	*true* si se ha pulsado la tecla `SHIFT` y *false* en otro caso.
code	String	Código de la tecla física pulsada. Ej: "KeyY", "Key1", "NumPad1", "KeyA", "KeyA"… Ej: "AltLeft", "ShiftRight", "ArrowDown"…
key	String	Valor de la tecla pulsada. Ej. "Y", "1", "a", "A". Ej: "Alt", "Shift", "ArrowDown"…

Notas:

- Cuando se pulsa una tecla correspondiente a un carácter alfanumérico, se produce la siguiente secuencia de eventos:
  `keydown >> keypress >> keyup`
- Cuando se pulsa otro tipo de tecla, se produce la siguiente secuencia de eventos:
  `keydown >> keyup`
- Si se mantiene pulsada la tecla
  - Si la tecla corresponde a un carácter alfanumérico:
    `keydown >> keypress >> keydown >> keypress >> … >> keyup`
  - En otro tipo de teclas:
    `keydown >> keydown >> … >> keyup`

Ejemplo: impedir que en el <`textarea`> se introduzca ningún carácter

```
<textarea onkeypress="return false;"></textarea>
```

Ejemplo: limitar el número de caracteres en un textarea

```
function limita(maximoCaracteres) {
  let elemento = document.getElementById("texto");
  if(elemento.value.length >= maximoCaracteres ) {
    return false;
  } else {
    return true;
  }
}
<textarea id="texto" onkeypress="return limita(100);"></textarea>
```

Eventos HTML

Evento	Descripción
load	Se produce en el objeto `window` cuando la página y todos sus recursos, se cargan por completo. En el elemento <`img`> cuando se carga por completo la imagen.
DOMContentLoaded	Se produce cuando finaliza la carga de la página, aunque los demás recursos (imágenes, ficheros JavaScript o CSS, etc.) no hayan cargado todavía, pero el árbol DOM sí.

unload	Se produce en el objeto `window` cuando la página desaparece por completo (al cerrar la ventana del navegador, por ejemplo). En el elemento `object` cuando desaparece el objeto.
abort	Se produce en un elemento `object` cuando el usuario detiene la descarga del elemento antes de que haya terminado.
error	Se produce en el objeto `window` cuando se produce un error de JavaScript. En el elemento `<img>` cuando la imagen no se ha podido cargar por completo y en el elemento `object` cuando el elemento no se carga correctamente.
select	Se produce cuando se seleccionan varios caracteres de un cuadro de texto (<`input`> y <`textarea`>).
change	Se produce cuando un cuadro de texto (<`input`> y <`textarea`>) pierde el foco y su contenido ha variado. También se produce cuando varía el valor de un elemento <`select`>.
submit	Se produce cuando se pulsa sobre un botón de tipo `submit` (<`input type="submit"`>)
reset	Se produce cuando se pulsa sobre un botón de tipo `reset` (<`input type="reset"`>).
resize	Se produce en el objeto `window` cuando se redimensiona la ventana del navegador.
scroll	Se produce en cualquier elemento que tenga una barra de `scroll`, cuando el usuario la utiliza. El elemento <`body`>contiene la barra de `scroll` de la página completa.
focus	Se produce en cualquier elemento (incluido el objeto `window`) cuando el elemento obtiene el foco.
blur	Se produce en cualquier elemento (incluido el objeto `window`) cuando el elemento pierde el foco.

Notas:

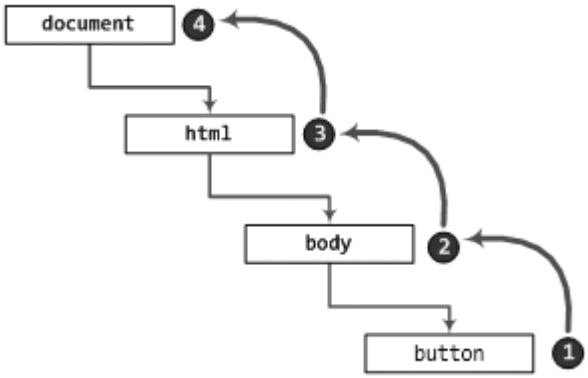
- Uno de los eventos más utilizados es el evento `DOMContentLoaded`, ya que todas las acciones que se realizan accediendo al DOM requieren que la página esté cargada por completo y, por tanto, el árbol DOM se haya construido completamente.
- Si lo que necesitamos es que no solo se cargue la página, sino que además lo hagan todos sus recursos (imágenes, ficheros JavaScript o CSS, etc.), utilizaremos `load`.
- `DOMContentLoaded` ocurre antes que `load` y es preferible utilizarlo.
- El elemento <`body`> define las propiedades `scrollLeft` y `scrollTop` que se pueden emplear junto con el evento `scroll`.

Eventos delegados: *bubbling*

El *bubbling* nos permite programar los eventos de forma delegada, lo que simplifica y clarifica mucho el código, especialmente cuando tenemos varios manejadores de eventos asociados al mismo evento, pero generados por diferentes elementos HTML.

Ejemplo: evento `click`

Cuando nosotros hacemos clic sobre un botón, se mira si hay un manejador asociado a ese elemento HTML, si hay, se atiende y finaliza, si no hay, se pasa al nivel siguiente, en este caso sería el `body`, se mira si hay un manejador asociado a `body`, si hay si atiende y si no, se pasa al siguiente nivel, así hasta llegar a `document`.
Lo que hacemos es manejar los eventos en `document` directamente, hacemos un único manejador para todos los eventos `click` y trabajamos con la información del elemento HTML.



Para saber cuál ha sido el elemento HTML de todos los posibles que pueden lanzar el evento, e invocar la acción correspondiente, tenemos el siguiente método del DOM:

Método	Devuelve	Descripción
`matches(`*`selectorCSS`*`)`	`Boolean`	Devuelve *true* si el elemento tiene asociado el *selectorCSS*, en caso contrario, devuelve *false*.

Ejemplo:

```
document.addEventListener("click", ev => {
    if (ev.target.matches("#n1")) vaciar();
    else if (ev.target.matches("#b1")) cuadrado();
});
```

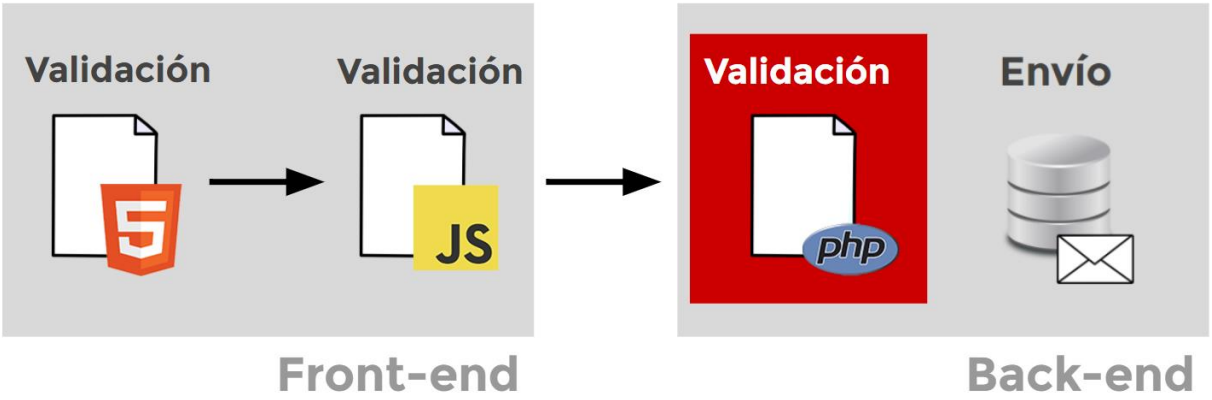
Validación de formularios

Introducción

Al crear un formulario para recoger los datos que introduzca un usuario en una web, debemos ser conscientes de un detalle ineludible: **los usuarios se equivocan al rellenar un formulario**. Debemos estar preparados y anticiparnos a estos errores, para intentar que los datos lleguen correctamente a su destino.

El esquema ideal de validación es aquel en que el formulario tiene validación en el lado del cliente y en el lado del servidor, también denominado **doble validación**.

El formulario estaría diseñado en HTML5 y validado mediante JavaScript y/o HTML5 para comprobar que los datos son correctos, si supera esa validación, vuelve a pasar otro proceso de validación en el lado del servidor. Una vez superadas ambas, el formulario será procesado para enviarlo por email o almacenar los datos en la base de datos (por ejemplo).



Validación del formularios en el lado del cliente

Aunque la **validación de formularios en el lado del cliente** no reemplaza la validación del lado del servidor, que todavía es necesaria por seguridad e integridad de la información, nos aporta claras ventajas:

- Mejora la experiencia de usuario: indicándole cómo rellenar el formulario y ofreciéndole mensajes de error claros e inmediatos.
- Evita retrasos entre el cliente y el servidor, aunque le facilite el trabajo al servidor, él es el que debe hacer la validación real antes de procesar los datos.

Tenemos dos tipos:

- **Validación** de formularios **incorporada** HTML5 y CSS: nos aporta mayor rendimiento, pero es menos personalizable.
- **Validación** de formularios avanzada **JavaScript**: nos aporta menor rendimiento, pero es más personalizable, pero debemos crearlo todo o utilizar alguna biblioteca (por ejemplo: http://rickharrison.github.io/validate.js/).

Se puede trabajar con una de las dos validaciones o mejor, combinar ambas.

Validación básica con JavaScript

El navegador genera un array de formularios `forms` y un array de elementos de cada formulario `elements`. Tanto los formularios como sus elementos son accesibles también a través de su atributo `id` o `name`.

Cómo seleccionar el formulario

- Conociendo el id del formulario (ej. miFormulario):

```
let formulario = document.getElementById("miFormulario");
let formulario = document.forms.miFormulario;
let formulario = document.forms["miFormulario"];
```

- Conociendo el número del formulario en la página (ej. pos):

```
let formulario = document.getElementsByTagName("forms")[pos];
let formulario = document.forms[pos];
```

Cómo seleccionar los elementos de un formulario

- Conociendo el id del elemento (ej. miCampo):

```
let campo = document.getElementById("miCampo");
```

- Accediendo al array de elementos del formulario:

```
let campos = formulario.elements;
```

- Accediendo al array de un tipo de elemento determinado:

```
let campos = document.getElementsByTagName("etiqueta");
let campos = formulario.getElementsByTagName("etiqueta");
```

- Accediendo al array de elementos que tienen un valor name (ej. nombre)

```
let campos = document.getElementsByName("nombre");
```

Cómo validar

```
document.addEventListener("DOMContentLoaded", () => {
    document.getElementById("enviar").addEventListener("click",
validar, false);
});
```

Funciones de comprobación

Algunas cosas que se utilizan:

```
elemento.value    //Valor del elemento del formulario
isNaN()    //Comprueba que sea número
elemento.checked    //Comprueba si está marcado un checkbox
elemento.className    //Clase CSS asociada al elemento
elemento.focus()    //Pone el foco en el elemento
```

Función que valida varios campos:

```
function validar(e) {
      if (valida1() && valida2() && confirm(...))
            return true;
      else {
            //Evitamos que se envíe el formulario
            e.preventDefault();
            return false;
      }
}
```

Validación avanzada con HTML5 y JavaScript

La llegada de HTML5 introdujo en el lenguaje la posibilidad validación de formularios del lado del cliente, conocida como **validación de restricciones**, que se basa en:

- Atributos de los elementos HTML
- Pseudoclases CSS
- Propiedades y métodos JavaScript del DOM

Atributos de los elementos HTML

Atributo	Elemento HTML	Descripción
required	`<input>`,`<select>`, `<textarea>`	Indica que se debe introducir algún dato.
pattern	`<input>`	Restringe el valor para que concuerde con una expresión regular específica.
min	`<input>`	Restringe el valor mínimo que puede ser introducido.
max	`<input>`	Restringe el valor máximo que puede ser introducido.
step	`<input>`	Especifica en qué cantidad aumenta o disminuye el valor cuando se utilizan los controles de entrada ↑↓
minlength	`<input>` `<textarea>`	Restringe el número mínimo de caracteres que el usuario puede introducir.
maxlength	`<input>` `<textarea>`	Restringe el número máximo de caracteres que el usuario puede introducir.
type	`<input>`	Los valores `url` y `email` para `type` restringen el valor para una URL o dirección de correo válida respectivamente.
novalidate	`<form>`	Deshabilita la validación de restricciones de HTML5.
formnovalidate	`<button>` `<input>`	Deshabilita la validación de restricciones de HTML5. En el elemento input solo es válido si el `type` es `submit` o `image`.

Si los datos que se introducen en un campo de formulario siguen todas las reglas que especifican los atributos anteriores, se consideran válidos. Si no, se consideran no válidos.

Cuando un elemento es válido, ocurre lo siguiente:

- El elemento coincide con la pseudoclase `:valid` de CSS, lo que te permite aplicar un estilo específico a los elementos válidos.
- Si el usuario intenta enviar los datos, el navegador envía el formulario siempre que no haya nada más que lo impida (por ejemplo, JavaScript).

Cuando un elemento no es válido, ocurre lo siguiente:

- El elemento coincide con la pseudoclase `:invalid` de CSS, y a veces con otras pseudoclases como por ejemplo, `:out-of-range`, dependiendo del error, que nos permite aplicar un estilo específico a elementos no válidos.
- Si el usuario intenta enviar los datos, el navegador bloquea el formulario y muestra un mensaje de error.

Pseudoclases CSS

Pseudoclase	Descripción
`:invalid`	Nos permite aplicar estilos a elementos con resultado no válido tras la validación de restricciones.
`:optional`	Nos permite aplicar estilos a elementos opcionales, es decir, que no tienen el atributo `required`.
`:required`	Nos permite aplicar estilos a elementos obligatorios, es decir, que tienen el atributo `required`.
`:valid`	Nos permite aplicar estilos a elementos válido tras la validación de restricciones..

Propiedades y métodos JavaScript del DOM

Propiedad	Tipo	Contenido
`willValidate`	`Boolean`	Devuelve `true` si el elemento se valida cuando se envía el formulario; `false` en caso contrario
`validity`	`ValidityState`	Devuelve un objeto que contiene varias propiedades que describen el estado de validez del elemento, algunas de ellas: ◦ **`patternMismatch`**: `true` si no coincide con el `pattern` especificado. ◦ **`tooLong`**: `true` si es mayor que el valor de `maxlength`. ◦ **`tooShort`**: `true` si es menor que el valor de `minlength`. ◦ **`rangeOverflow`**: `true` si es mayor que el valor de `max`.

Propiedad	Tipo	Contenido
		◦ **rangeUnderflow**: `true` si es menor que el valor de `min`. ◦ **typeMismatch**: `true` si no cumple con la sintaxis requerida por el atributo `type`. ◦ **valid**: `true` si cumple con todas sus restricciones de validación. ◦ **valueMissing**: `true` si tiene un atributo `required` pero no tiene valor.
validationMessage	`String`	Contiene el mensaje que mostrará el navegador si el elemento no cumple la validación.

Método	Devuelve	Descripción
checkValidity()	`Boolean`	Devuelve `true` si todos los elementos que necesitan validación satisfacen las restricciones, y `false` si no lo hacen.
setCustomValidity()	`undefined`	Modifica el valor de la propiedad `validationMessage`, lo que nos permite personalizar el mensaje en caso de error.

Ejemplo:

```
function validaNombre() {
    let elemento = document.getElementById("nombre");
    if (!elemento.checkValidity()) {
        if (elemento.validity.valueMissing) {
            mensaje = "Debe introducir un nombre";
        }
        if (elemento.validity.patternMismatch) {
            mensaje = "El nombre debe ser de entre 2 y 15";
        }
        return false;
    }
    return true;
}
```

Expresiones regulares

Las expresiones regulares son un elemento de uso muy habitual en la mayoría de lenguajes de programación. Se utilizan principalmente, para definir patrones que reconocen cadenas de caracteres específicas. Estas condiciones nos facilitan la tarea de búsqueda de textos dentro de otros textos, validación de formularios o extracción avanzada de subcadenas.

Las expresiones regulares de JavaScript son objetos de tipo **RegExp** que se pueden crear delimitándolas entre dos barras (/) al principio y al final. Tras las barras se pueden indicar

controles o banderas para crear expresiones regulares avanzadas. También se pueden crear utilizando su constructor. La sintaxis sería la siguiente:

```
let expReg = /expresión/[controles];
let expReg = new RegExp("expresión", [controles]);
```

Una expresión regular se puede crear a través de su literal o. Ejemplo:

```
let expReg1 = /\w+/;
let expReg2 = new RegExp("\\w+");
```

Métodos para trabajar con expresiones regulares

Las expresiones regulares se utilizan fundamentalmente con dos métodos de la clase **`String`**:

Método	Devuelve	Descripción
`cadena.match(/`*`expReg`*`/)`	`[String]`	Devuelve un array con primer substring que casa o `null`. Algunos navegadores devuelven un array con más parámetros, pero conviene utilizar solo el primero.
`cadena.replace(/`*`expReg`*`/,` **`repuesto`**`)`	`String`	Devuelve un string resultado de sustituir el primer substring que casa por *repuesto*.

Ejemplos:

```
"Es hoy".match(/hoy/) => devuelve ["hoy"]
"Número: 142719".replace(/1/, "x") => devuelve "Número: x42719"
```

Cuando se validan campos de formularios, es muy frecuente tener que comprobar si una cadena tiene un formato determinado. Esto se puede hacer comprobando si una cadena casa con una expresión regular utilizando el siguiente método de la clase **`RegExp`**:

Método	Devuelve	Descripción
`expReg.test(`*`cadena`*`)`	`Boolean`	Devuelve `true` si la cadena casa con la expresión regular, `false` en otro caso.
`exReg.exec(`*`cadena`*`)`	`[String]`	Devuelve un array con primer substring que casa o `null`. Se comporta como `match()`.

Ejemplo:

```
let expReg = /[0-9]{8}[a-zA-Z]/
expReg.test("012345678A")  => devuelve true
```

Construcción de expresiones regulares

Algunos patrones básicos

Carácter	Significado
`c`	Carácter. Siendo c un carácter cualquiera, encaja solamente con ese carácter.
`cde`	Secuencia. Siendo c, d, y e tres caracteres cualquiera, encaja solamente si esos caracteres aparecen de esa manera. Ejemplo: `/hola/` encaja con textos que contengan la palabra hola

^	Principio de cadena. Ejemplo: `/^A/` encaja con la "A" en "Asturias" pero no encaja con la "A" en "Es Asturias".
$	Final de cadena. Ejemplo: `/s$/` encaja con última "s" en "Asturias" pero no con la primera (segunda letra).
.	Cualquier carácter excepto fin de línea.

Ejemplos:

```
"Es hoy".match(/hoy$/) => devuelve ['hoy'] (está al final)
"Es hoy".match(/^hoy/) => devuelve null (no está al principio)
"Es hoy".match(/^..../) => devuelve los 4 primeros caracteres:
['Es h']
```

Clases y rangos de caracteres

Carácter	Significado
[xyz]	Clase de caracteres. Encaja con uno de los caracteres que están entre los corchetes. Ejemplos: `/chic[ao]/` cadenas `"chico"` y `"chica"`. `/[aeiou]/` cualquier vocal (minúscula).
[^xyz]	Clase de caracteres negada. Encaja con un carácter que NO esté entre los corchetes. Ejemplo: `/[^aeiou]/` no debe ser vocal (minúscula).
[a-z]	Rango de caracteres. En este corresponde al rango “a-z” de letras ASCII. Ejemplo: `/[0-9][0-9][0-9]/` las cadenas que contienen tres números seguidos, como `"123"` y `"676"` encajan con el patrón.
[^a-z]	Rango de caracteres negado. Ejemplo: `/[^c-z]a/` las cadenas `"aa"` y `"ba"` encajan, pero la cadena `"za"` no encaja.

Ejemplos:

```
"canciones".match(/[aeiou]/) => devuelve ['a']
"canciones".match(/c[aeiou]/) => devuelve ['ca']
"canciones".match(/n[aeiou]/) => devuelve ['ne']
```

Otras clases y rangos de caracteres

Carácter	Significado
\d	Cualquier dígito. Equivale a `[0-9]`. Ejemplo: `/\d/` o `/[0-9]/` encaja con "2" en "B2".
\D	Cualquier carácter que no sea un dígito. Equivale a `[^0-9]`. Ejemplo: `/\D/` o `/[^0-9]` encaja con "B" en "B2".
\w	Cualquier carácter alfanumérico, incluido el subrayado. Equivale a [A-Za-z0-9_]. Ejemplo: `/\w/` encaja con "a" en "apple", "5" en "$5.28" y "3" in "3D".

\W	Cualquier carácter que no es un carácter alfanumérico o subrayado. Equivale a [^A-Za-z0-9_] Ejemplo: `/\W/` encaja con "." y con "$" en "$5.28".
\s	Cualquier carácter de tipo espacio en blanco, como espacio en blanco, tabulador, salto de línea. Reconoce separadores [\f\n\r\t\v\u00a0\u1680 .....]. Ejemplo: `/\s\w*/` encaja con " bar" en "foo bar".
\S	Cualquier carácter que no sea de tipo espacio en blanco.
\t	Tabulador horizontal.
\r	Salto de línea.
\n	Nueva línea.
\v	Tabulador vertical.
\f	Salto de página.
****	Para escapar un carácter que tiene un significado especial, como [,], /, \, etc. Por ejemplo \\ es la forma de representar la barra invertida.
\uffff	Permite indicar un carácter Unicode mediante su código hexadecimal.
\p{PropUnicode}	Cualquier carácter que pertenezca a la propiedad Unicode indicada.
\P{PropUnicode}	Cualquier carácter que no pertenezca a la propiedad Unicode.

Controles i, g, m

Carácter	Significado
i	Búsqueda insensible a mayúsculas.
g	Búsqueda global de todos los substrings que casan con el patrón (no solamente con el primero).
m	Búsqueda multilínea, donde ^ y $ representan principio y fin de línea.

Ejemplos con match():

```
"canciones".match(/[aeiou]/g) => devuelve ['a', 'i', 'o', 'e']
"canciones".match(/c[aeiou]/g) => devuelve ['ca', 'ci']
"Hoy dice hola".match(/ho/i) => devuelve ['Ho']
"Hoy dice hola".match(/ho/ig) => devuelve ['Ho', 'ho']
"Hola Pepe\nHoy vás".match(/^Ho/g) => devuelve ['Ho']
"Hola Pepe\nHoy vás".match(/^ho/gim) => devuelve ['Ho', 'Ho']
```

Ejemplos con replace():

```
"Número: 142719".replace(/1/, 'x') => devuelve 'Número: x42719'
"Número: 142719".replace(/1/g, 'x') => devuelve 'Número: x427x9'
"Número: 142719".replace(/[0-9]+/, '<número>') => devuelve
'Número: <número>'
```

Operadores de repetición

Carácter	Significado
*	Cero o más veces. Ejemplo: `/a*/` encaja con: “”, “a”, “aa”, “aaa”, ...
+	Una o más veces. Ejemplo: `/a+/` encaja con: “a”,“aa”, “aaa”, …

?	Cero o una vez. Ejemplo: `/a?/` encaja solo con: "" y "a"
{n}	n veces. Ejemplo: `/a{2}/` encaja solo con: "aa"
{n,}	n o más veces. Ejemplo: `/a{2,}/` encaja con: "aa", "aaa", "aaaa", ...
{n, m}	Entre n y m veces. Ejemplo: `/a{2,3}/` encaja solo con: "aa" y "aaa"

Ejemplos:

```
"tiene".match(/[aeiou]+/g) => devuelve ['ie', 'e']
// cadenas no vacías de vocales
"tiene".match(/[aeiou]?/g) => devuelve ['', 'i', 'e', '', 'e', '']
// vocal o nada
"tiene".match(/[aeiou]*/g) => devuelve ['','ie', '', 'e', '']
// cadenas de vocales incluyendo ""
"Había un niño.".match(/[a-zñáéíóú]+/ig) => devuelve ['Había',
'un', 'niño']
// palabras en castellano: ASCII extendido con ñ, á, é, í, ó, ú
```

Los operadores de repetición son "ansiosos" y siempre buscarán encajar o casar con la cadena más larga posible, pero pueden volverse "perezosos" añadiendo `?` detrás del operador de repetición, en ese caso casarán con la cadena más corta posible.

Ejemplos:

```
"aaabb".match(/a+/) => devuelve ['aaa']
"aaabb".match(/a+?/) => devuelve ['a']
"ccaaccbccaa".match(/.+cc/) => devuelve ['ccaaccbcc']
"ccaaccbccaa".match(/.+?cc/) => ['ccaacc']
```

Estos dos patrones se utilizan mucho:

- `.+` cualquier cadena con longitud uno o más
- `.*` cualquier cadena con longitud cero o más

Patrones alternativos

Carácter	Significado
`x|y`	Encaja con x o con y. Ejemplo: `/green|red/` encaja con "green" en "green apple" y con "red" in "red apple".

Ejemplos:

```
"canciones".match(/ci|ca/) => devuelve ['ca']
"canciones".match(/ci|ca/g) => devuelve ['ca','ci']
"1 + 2 --> tres".match(/[a-z]+|[0-9]+/g) => devuelve ['1','2','tres']
```

Subpatrones

Carácter	Significado
`(x)`	Subpatrón. Encaja con x y recuerda la cadena encajada. El patrón y los subpatrones pueden reutilizarse referenciándolos con: o $0 representa todo el patrón o $1 representa el primer subpatrón o $2 el match el segundo subpatrón y así sucesivamente Ejemplos: `/(c)([aeiou])/` patrón c seguido de vocal, delimitando la c como primer subpatrón ($1) y las vocales como segundo subpatrón ($2) `/([0-9]+)(,[0-9]*)?/` patrón que representa números enteros o decimales, tiene dos subpatrones `([0-9]+)` y `(,[0-9]*)`

El método `match()` realmente busca patrones y subpatrones, por lo tanto, `cadena.match(/patrón/)` devolverá el match completo seguido de los subpatrones: `[match$0, macth$1, match$2,…]`

Ejemplos con match():

```
"canciones".match(/(c)([aeiou])/) => devuelve ['ca','c','a']
"canciones".match(/c([aeiou])n/) => devuelve ['can','a']
"canciones".match(/(..)..(..)/) => devuelve ['cancio','ca','io']
```

Ejemplos con replace():

```
//Sustituye por el primer subpatrón
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '$1') => devuelve
'Número: 142'
//Sustituye por 0 y segundo supbatrón
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '0$2') =>
devuelve 'Número: 0,719'
//Sacamos la , del subpatrón para que la sustituya por el .
"Número: 142,719".replace(/([0-9]+),([0-9]*)?/, '$1.$2') =>
devuelve 'Número: 142.719'
```

Ejemplo con test() y exec():

```
//Patrón que representa un código postal (formado por 5 números
del 00000 al 52999)
let cp = /^((5[012])|([0-4][0-9]))([0-9]{3})$/;
cp.test("49345") => devuelve true
cp.test("53345") => devuelve false
cp.exec("49345") => devuelve ['49345']
cp.exec("53345") => devuelve null
```

Almacenar datos en el equipo cliente

Cookies

Con `document.cookie` se obtienen y establecen las cookies asociadas al documento.
Las cookies son datos almacenados en nuestro ordenador, en pequeños archivos de texto.
Van ligadas a los navegadores y a los documentos: las que vemos en un navegador, no se pueden acceder desde otro.
Pueden tener fecha de expiración. Si no la tienen, expiran al acabar la sesión.

Leer todas las cookies accesibles

```
let todasCookies = document.cookie;
```

En el código anterior, `todasCookies` tendrá una cadena formada por una lista de todas las cookies (con formato clave=valor) separadas por punto y coma.

Ejemplo: Esta es la cadena obtenida al ejecutar el código anterior con https://devdocs.io:

```
"_gauges_unique_month=1; _gauges_unique_year=1;
docs=css/dom/dom_events/html/http/javascript; schema=2;
_ga=GA1.2.135871839.1553426731; _gid=GA1.2.1768504745.1554392662;
count=5; news=1537660800000; version=1548019000;
_gauges_unique_hour=1; _gauges_unique_day=1; _gauges_unique=1"
```

Ejemplos:

```
alert(document.cookie);
document.cookie = "nombre = Ada";
```

Crear una nueva cookie

```
document.cookie = nuevaCookie;
```

En el código anterior, `nuevaCookie` es una cadena con la forma `clave = valor`.
Solo se puede establecer/actualizar una cookie de cada vez utilizando este método.

Después del valor de la cookie se pueden especificar algunos atributos de la cookie. Se escriben con un punto y coma y después el nombre del atributo y su valor:

- `path = path;` (ej. path='/'). Si no se especifica, se toma como valor por defecto el path del documento actual.
- `max-age = max-age-en-segundos;`
- `expires = fecha-en-formato-GMT;`
  Si no es especifican `max-age` o `expires`, la cookie expira al terminar la sesión.

Ejemplo:

```
document.cookie = "nombre = pepe; max-age = 3600";
```

Modificar una cookie
Modificar una cookie es sobrescribirla.

Ejemplo:

```
document.cookie = "nombre = Laura";
```

Borrar una cookie

Para borrar una cookie se puede dar una fecha de expiración anterior a la actual.

Ejemplo:

```
document.cookie = "nombre =; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Web Storage

JavaScript soporta persistencia de datos en el navegador a través de Web Storage. Esta técnica, nos permite almacenar más información y de forma más intuitiva y segura que con las cookies. Otra ventaja de Web Storage es que almacena los datos por origen, es decir, todas las páginas con el mismo origen (protocolo, dominio y puerto), podrán acceder a los mismos datos. Por estos motivos, hoy en día **se recomienda utilizar esta técnica en vez de cookies**.

Disponemos de dos mecanismos distintos para trabajar con Web Storage, ambos disponibles con las siguientes propiedades del objeto `Window`:

- **`localStorage`**: permite crear contenedores de datos permanentes que solo se eliminan si se borran desde JavaScript.
- **`sessionStorage`**: permite crear contenedores de datos asociados a la sesión.
  - Comienzo de sesión: apertura de navegador o pestaña.
  - Final de sesión: cierre de navegador o pestaña.

Ambas propiedades deben almacenar siempre strings, no pueden almacenar otro tipo de datos.

Los métodos que se pueden utilizar para gestionar los datos almacenados son:

Establecer un item: .setItem()

```
localStorage.setItem("nombre", "valor");
sessionStorage.setItem("nombre", "valor");
```

También se puede hacer mediante propiedades dinámicas, pero hoy en día no se recomienda:

```
localStorage.nombre = "valor";
sessionStorage.nombre = "valor";
```

Obtener el valor de un ítem: .getItem()

```
localStorage.getItem("nombre");
sessionStorage.getItem("nombre");
```

También se puede hacer mediante propiedades dinámicas, pero hoy en día no se recomienda:

```
localStorage.nombre;
sessionStorage.nombre;
```

Eliminar un ítem: .removeItem()

```
localStorage.removeItem("nombre");
sessionStorage.removeItem("nombre");
```

Eliminar todos los ítems: .clear()

```
localStorage.clear();
sessionStorage.clear();
```

Comprobar si el navegador soporta Storage

```
if (typeof(Storage) != "undefined") {
      //Soporta Web Storage
}
```

Same Origin Policy

Hemos de tener en cuenta que los contenedores de `localStorage` y `sessionStorage` siguen la política de seguridad **same-origin-policy**, es decir, un script solo puede acceder a contenedores creados por otros scripts que vinieron del mismo origen, es decir, del mismo servidor. El origen de un script son el protocolo, dominio y puerto del servidor.

Si se viola esta política de seguridad, el navegador lanzará la excepción ***security error*** o simplemente no funcionará correctamente. Esto suele ocurrir siempre que trabajemos con archivos locales. En ocasiones lo podemos solucionar accediendo por medio de localhost en vez de a través de la ruta a nuestro archivo, pero la solución real es alojar nuestros archivos en servidores, por ejemplo: https://neocities.org/ nos ofrece servicio gratuito.

También hemos de tener en cuenta que el usuario puede tener su navegador configurado para denegar permisos a datos persistentes para el origen especificado, en ese caso no podemos hacer nada, salvo pedirle que los permita.