DESARROLLO WEB ENTORNO CLIENTE

2º DAW CFGS

5. JS: EVENTOS, MODULOS Y LOCALSTORAGE

Natalia Escrivá Núñez

IES Serra Perenxisa

n.escrivanunez@edu.gva.es

CONTENIDO

1. DOM: EVENTOS

En la programación tradicional estamos acostumbrados a que el código se ejecute siguiendo una secuencia ordenada de instrucciones.

JavaScript nos da la posibilidad de trabajar con la programación basada en eventos, de forma que ciertas instrucciones se ejecutan en el momento que ocurre algo: el **evento**.

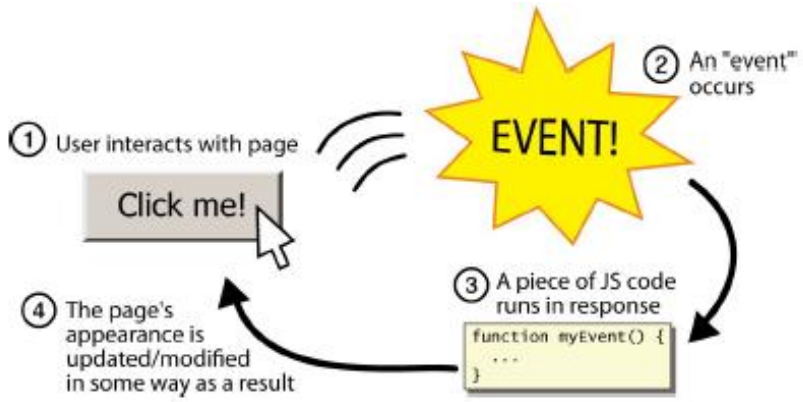
Un **evento es un suceso,** algo que ocurre como **resultado de un acto** del usuario o por otras razones (se ejecutan automáticamente).

Los eventos son el **mecanismo para comunicar la aplicación con el usuario.** En esta interacción, la aplicación variará su funcionamiento debido a las acciones realizadas normalmente por el usuario.

Como ejemplos de eventos, dar un "like" a una publicación, añadir un comentario o navegar por el scroll de una aplicación.

Los eventos se asocian a un elemento concreto del ***DOM***.

El funcionamiento es el siguiente: hay un proceso que automatiza la captura de las acciones que generan los eventos (**manejador de eventos o listener**), de modo que cuando se produce la acción (click del ratón), el proceso **listener** ejecuta el código de una **función**. Lo que debemos hacer es **preparar la escucha del evento** y esta se produce en segundo plano.



El funcionamiento de los eventos no ha sido siempre el mismo, con *listeners*.

Antes de esta implementación, se asociaban los eventos directamente a los atributos de los elementos en el propio HTML o bien en el código JS pero con limitaciones.

Vamos a ver el desarrollo de los eventos a lo largo del tiempo.

1.1. Modelos de eventos en línea

Este modelo también es conocido como "Eventos en atributos HTML".

Es el modelo más sencillo y menos profesional, por lo que **NO se recomienda**.

Hace referencia a las primeras versiones de JavaScript, en las que la captura de un evento se realizaba desde el propio código HTML.

Todavía se puede utilizar por motivos de compatibilidad, pero no es aconsejable porque **implica mezclar código HTML con JavaScript**.

Cada elemento HTML, tiene sus posibles eventos como propiedades: **solo puede tener un evento de cada tipo.**

Este método utiliza atributos en las etiquetas de los elementos de la página que comienzan con el **prefijo "on"** seguida del nombre del evento (usa los manejadores de eventos como atributos).

Por ejemplo, en el siguiente fragmento de **HTML**, vemos que un elemento "h2" tiene 3 eventos que provocarán tres reacciones diferentes.

```
   <h1>-- Modelo de eventos en línea --</h1>
  <h2 id ="cab1" onclick = "this.innerHTML='Eventos en HTML'"
  onmouseover="this.style.background='green'"
  onmouseout="this.style.background='yellow'">Haz click...</h2>
```

Véase el uso de "this". Es la manera de hacer referencia al elemento. También podemos seleccionar el elemento de esta forma….

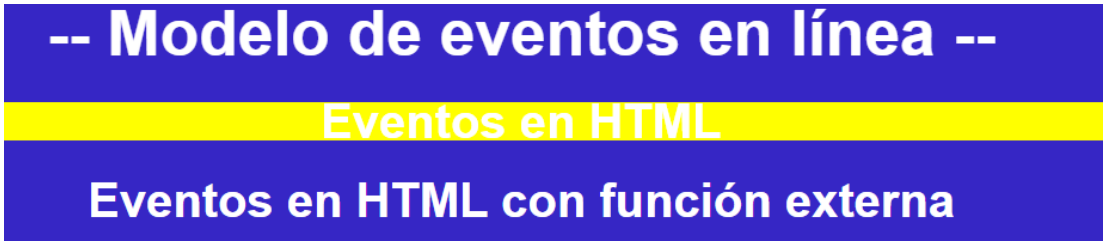
```
<h2 id="cab1" onclick="document.getElementById('cab1').innerHTML="...">...</h2>
```

Otra opción es utilizar los manejadores de eventos en un fichero externo…. Este sería el **HTML**:

```
<h2 onclick="cambiar(this)">Haz click...</h2>
```

Y este el fragmento de código del fichero **JS**:

```
function cambiar(elem) {
    elem.innerHTML = "Eventos en HTML con función externa";
}
```



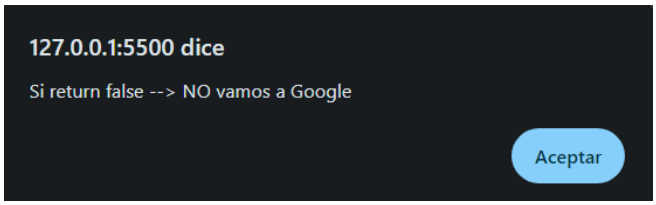
Si queremos evitar que el navegador ejecute una acción por defecto, añadimos “return false” o podemos asignar una función al return.

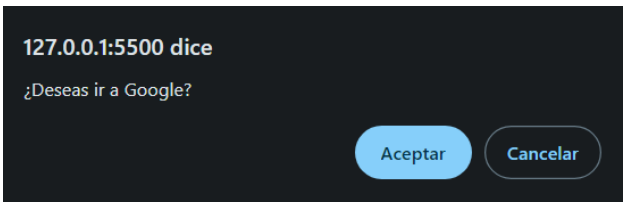
```
<a href="http://www.google.com" onclick="alertar(); return false;">Pulsa aquí para ver qué se ejecuta</a>
<br/>
<a href="http://www.google.com" onclick="return preguntar();">Pulsa aquí para ver qué se ejecuta</a>
```

Aquí tenemos las funciones en JS:

```
function alertar() {
    alert("Si return false --> NO vamos a Google");
}
```

```
function preguntar() {
    return confirm("¿Deseas ir a Google?");
}
```





1.1.1. ONLOAD

Este evento nos va a servir para hacer que se cargue la página HTML antes de cargar ningún JavaScript.

```
<body onload="alert('La página se ha cargado correctamente')">
```

1.2. Modelo de eventos tradicional

Este modelo, que tampoco va a ser el definitivo, consiste en aplicar sobre un elemento seleccionado a través de **JavaScript,** un evento con el prefijo “on”.

Este método es más correcto que el anterior ya que separa el código HTML del de JavaScript pero, aun así **NO es el recomendado,** puesto que permite asociar únicamente **UN SOLO EVENTO de cada tipo a un elemento.**

Lo que se hace en este modelo es crear los elementos en HTML con sus atributos y desde el archivo de JavaScript se hará el resto.

Tomemos como ejemplo este fragmento de HTML:

```
<h1>Modelo de eventos tradicional</h1>
<h2 id="trad">Haz click...</h2>
<h2 id="trad2">Haz click...</h2>
```

Será imprescindible cargar la página con el evento "onload". Lo haremos desde el fichero JS. Asignamos a cada elemento con su evento una función (que invocaremos SIN paréntesis:

```
window.onload = function () {
    alert("La página ha cargado correctamente");
document.getElementById("trad").onclick = tradicional; //¡¡S
document.getElementById("trad").onmouseover = tradicional1;
document.getElementById("trad2").onclick = tradicional2;
document.getElementById("trad2").onclick = tradicional3;
}
```

Después podremos ir desarrollando nuestras funciones.

Podemos desactivar el evento asignándole el valor "null".

```
function tradicional() {
    alert("Holaaaa");
    document.getElementById("trad").innerHTML = "Evento en JS, NO en etiqueta HTML (tradicional)";
    //si lo queremos desactivar para que lo ejecute una única vez
   // document.getElementById("trad").onclick = null;
}
function tradicional1(){
    document.getElementById("trad").style.color = "yellow";
}
```

En este otro ejemplo, vamos a comprobar cómo **si asignamos dos eventos iguales a un elemento, se ejecutará el último (elemento con id "trad2").**

```
function tradicional2() {
    document.getElementById("trad2").style.color = "green";
    document.getElementById("trad2").innerHTML = "No lo veré....";
}
function tradicional3() {
    document.getElementById("trad2").style.color = "red";
    document.getElementById("trad2").innerHTML = "Se ejecuta el último evento asignado";
}
```

1.3. Modelo de eventos W3C

Este es el modelo **que debemos seguir** porque trabaja con "escuchadores de eventos", es decir, los procesos que están esperando que se produzca la acción para ejecutar la función.

El listener será el método **addEventListener**.

Si queremos eliminar el evento usaremos **removeEventListener**.

Ese método tiene dos parámetros:

- Nombre del evento que estamos esperando.
- Función que se va a ejecutar (sin paréntesis). Podemos llamar a una función, usar una anónima o una arrow function.

El primer evento que debemos aplicar es el de carga, tenemos varias opciones:

```
window.addEventListener("load",inicio);
```

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

En el primer caso, se ejecutaría la función inicio() cuando la página estuviese cargada y en el segundo caso, cuando el HTML estuviese cargado completamente.

En los dos casos reacciona igual.

Estos eventos de inicio son muy importantes porque podemos tener código que se ejecute antes de que tengamos nuestro proyecto cargado.

Veamos un ejemplo:

```
console.log(1);
window.addEventListener("load", () =>{
    console.log(2);
});
console.log(3);
```

Cuando carguemos la página, el resultado que tendremos es la secuencia: 1, 3, 2.

1.4. Eventos de ratón

Los más usuales son click, dblclick. También encontramos:

- Mousedown → similar al click
- Mouseup → cuando soltamos el ratón
- Mouseenter → cuando entramos en la zona del elemento
- Mouseout → cuando salimos de la zona del elemento
- Mousemove → cada vez que el ratón se mueva

Ejemplos de uso:

```html
<h1>Modelo de eventos W3C</h1>
<button id="w3c">Haz click...</button>
<button id="w3canonim">Haz click...</button>
```

```javascript
document.addEventListener("DOMContentLoaded", inicio);

function inicio(){
    document.querySelector('#w3c').addEventListener("click",saludar);
    document.querySelector('#w3c').addEventListener("click",colorear);
    document.querySelector('h1:last-of-type').addEventListener("mouseenter",colorear);
    document.querySelector('h1:last-of-type').addEventListener("mouseout", quitarColor);
    document.querySelector('#w3canonim').addEventListener("dblclick",function(){
        this.innerHTML="Modificado con función anónima";
        this.style.backgroundColor = "yellow";
    })

}
```

```javascript
function saludar(){
    alert("Holaaa, este método SI es bueno!!!!!!");
    //si solo queremos que lo ejecute una vez
    //this.removeEventListener("click",saludar);
}
function colorear(){
    this.style.color = "red";
}
function quitarColor(){
    this.style.color = "white";
}
```

Encontramos dos propiedades del objeto de evento de ratón: pageX, pageY

Devuelven la posición horizontal y vertical en px del puntero del ratón, o punto de contacto de una pantalla táctil, **relativo al documento completo** (incluyendo desplazamiento).

En relación a la ventana, encontramos clientX y clientY.

1.5. Eventos de teclado

Los eventos más comunes son:

- Keydown → cada vez que presionamos el teclado
- Keyup → recoge cuando soltamos la tecla
- Blur → cuando se pierde el foco del elemento
- Input → dispara el evento cada vez que el valor del elemento cambia (caja te texto …)
- Change → dispara el evento cada vez que el valor del elemento cambia (combo, check…) y el usuario confirma (perdiendo el foco)

**Ejemplos de uso en los ficheros de la unidad…

1.6. Objeto de tipo evento

Cuando se produce un **evento**, el **navegador crea automáticamente un objeto de tipo evento** cuyas propiedades pueden ser muy útiles para darnos información sobre el evento en sí.

La **función que se ejecuta cuando el evento** se produce, puede incluir un **parámetro que hará referencia al objeto de tipo evento que ha creado el navegador.**

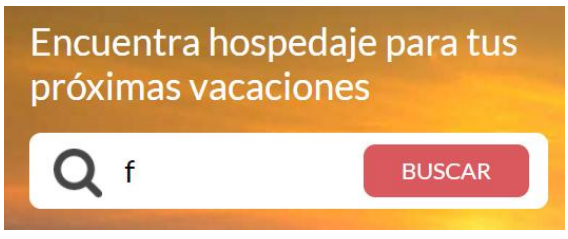
1.6.1. PROPIEDADES TARGET Y TYPE

Target hace referencia al **elemento** del ***document*** que causa el evento.

Type hace referencia al **tipo de evento** que ha ejecutado la función.

Utilizamos un formulario como ejemplo:

```
const busqueda = document.querySelector('.busqueda');
```




```
busqueda.addEventListener('input', (e) =>{
    console.log(e.target);
    console.log(e.type);
})
```

```
                                                    03-script
<input type="text" name="busqueda" class="busqueda"
placeholder="New York, Londres, Roma, Guadalajara">
input                                               03-script
```

Ejemplo 1 → Saber qué se está escribiendo en un formulario:

```
busqueda.addEventListener('input', (e) =>{
    console.log(e.target.value);
});
```


```
p
pa
par
pari
paris
```

Ejemplo 2 → Tenemos distintos **eventos asociados a distintos elementos** que llaman a una misma función, y queremos que se comporten de una manera determinada dependiendo del elemento concreto.

Partimos de este fragmento de HTML

```
<h1>Modelo de eventos W3C</h1>
<h2 id="uno">Pasa tu ratón por encima....</h2>
<h2 id="dos">Pasa tu ratón por encima....</h2>
```

Seleccionamos los elementos:

```
const unh2 = document.querySelector('#uno');
const otroh2 = document.querySelector('#dos');
```

Les asociamos los eventos:

```
unh2.addEventListener("mouseenter", cambiaFondo);
otroh2.addEventListener("mouseenter", cambiaFondo);
```

Dependiendo del h2, el color de fondo variará:

```
function cambiaFondo(e){
    if(e.target.id === "uno"){
        this.style.backgroundColor = "yellow";
    } else if (e.target.id === "dos"){
        this.style.backgroundColor = "green";
    }
}
```

También podemos hacer referencia a una clase con ***e.target.classList.contains('nombre_clase').***

1.7. Eventos de formulario

El evento submit es el que se encarga de enviar la información recogida para almacenarla o trabajar con ella conectando con una API o nuestro servidor.

Si queremos validar la información que se está introduciendo, podemos detener el proceso natural del formulario al que se le asocia el evento a través del método **preventDefault()**.

```
const formulario = document.querySelector('#formulario');

formulario.addEventListener('submit', (e) => {
    e.preventDefault();
    console.log(e.target);
    console.log(e.target.method);
    console.log(e.target.action);
});
```

```
                                                    05-scripts.js:8
▶ <form action="/buscador" method="POST" class=
  "formulario formulario-buscar" id="formulari
  o">…</form> flex
post                                                05-scripts.js:9
http://127.0.0.1:5501/buscador                      05-scripts.js:10
```

Lo que haríamos es detener la ejecución natural del navegador (en este caso, enviar los datos a través del método “post” a la página “buscador”).

En el cuerpo de esta función pondríamos el código para comunicar y enviar los datos a través de AJAX (obsoleto) o Fetch API a un servidor.

1.8. Eventos de scroll

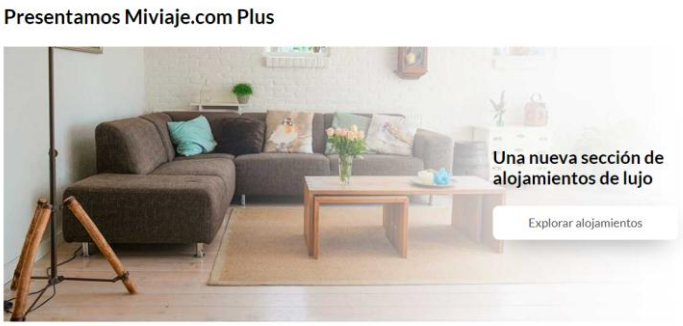
Estos eventos **suceden en la ventana global**. El evento concreto es scroll y podremos trabajar con el scroll a través de las propiedades del objeto window

- scrollY: scroll vertical en píxels
- scrollX: scroll horizontal en píxels

```
window.addEventListener('scroll', () => {
    const scrollPX = window.scrollY;
    console.log(scrollPX);
})
```

Ejemplo → Podemos saber exactamente cuando un usuario ha hecho scroll y ha llegado a un elemento concreto para aplicar una clase de CSS con alguna animación.

Para saber la ubicación concreta de un elemento usaremos el método **getBoundingClientRect()**



```
window.addEventListener('scroll', () => {
    const premium = document.querySelector('.premium');
    const ubicacion = premium.getBoundingClientRect();

    if (ubicacion.top < 750 && ubicacion.top > -500){
        console.log('El elemento ya está visible');
    } else {
        console.log('Aún no, da más scroll...');
    }
})
```

```
DOMRect {x: 107.0666732788086, y: 199.61668395996094, width: 48.0000305175781, top: 199.61668395996094, …}
    bottom: 647.6167144775391
    height: 448.0000305175781
    left: 107.0666732788086
    right: 1207.0666732788086
    top: 199.61668395996094
    width: 1100
    x: 107.0666732788086
    y: 199.61668395996094
    [[Prototype]]: DOMRect
El elemento ya está visible
```

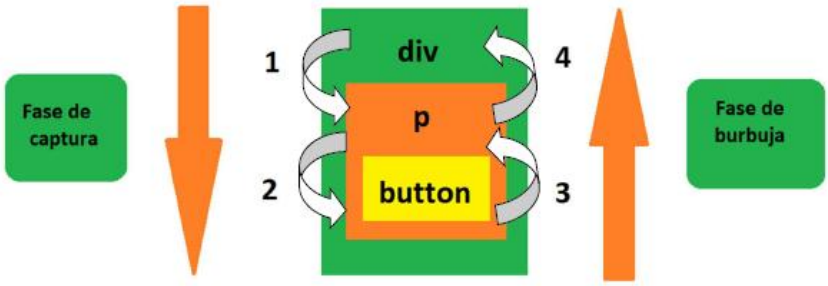
1.9. Propagación de eventos

Los eventos a veces pueden provocar comportamientos que no esperamos.

Esto es debido a la propagación de eventos o ***Event Bubbling***.

La ejecución de los eventos se va a propagar de hijos a padres. Esto es porque los eventos de gestionan en dos fases:

- Fase de captura: se capturan los eventos de forma individual
- Fase de burbuja: se lanzan de abajo a arriba.



Ejemplo

Partimos de estos elementos: div card → div info → p titulo

```
<div class="card">
  <img src="img/hacer1.jpg">
  <div class="info">
    <p class="categoria concierto">concierto</p>
    <p class="titulo">Música electrónica 2021</p>
    <p class="precio">$1,200 por persona</p>
  </div>
```

Los seleccionamos:

```
const cardDiv = document.querySelector('.card');
const infoDiv = document.querySelector('.info');
const titulo = document.querySelector('.titulo');
```

Caso 1: cliclamos sobre la foto, el elemento "abuelo"→ card

```
estás en card: abuelo
```

Caso 2: clicamos sobre la categoría (concierto), el elemento "padre" → info

```
estás en info: papá
estás en card: abuelo
```

Caso 3: clicamos sobre el título, el elemento con jerarquía más baja

```
estás en titulo: hijo
estás en info: papá
estás en card: abuelo
```

1.9.1. EVITAR LA PROPAGACIÓN DE EVENTOS

Para solucionarlo, utilizaremos el método del evento **stopPropagation()** en cada evento:

```
titulo.addEventListener('click', (e) =>{
    e.stopPropagation();
    console.log("estás en titulo: hijo");
});
```

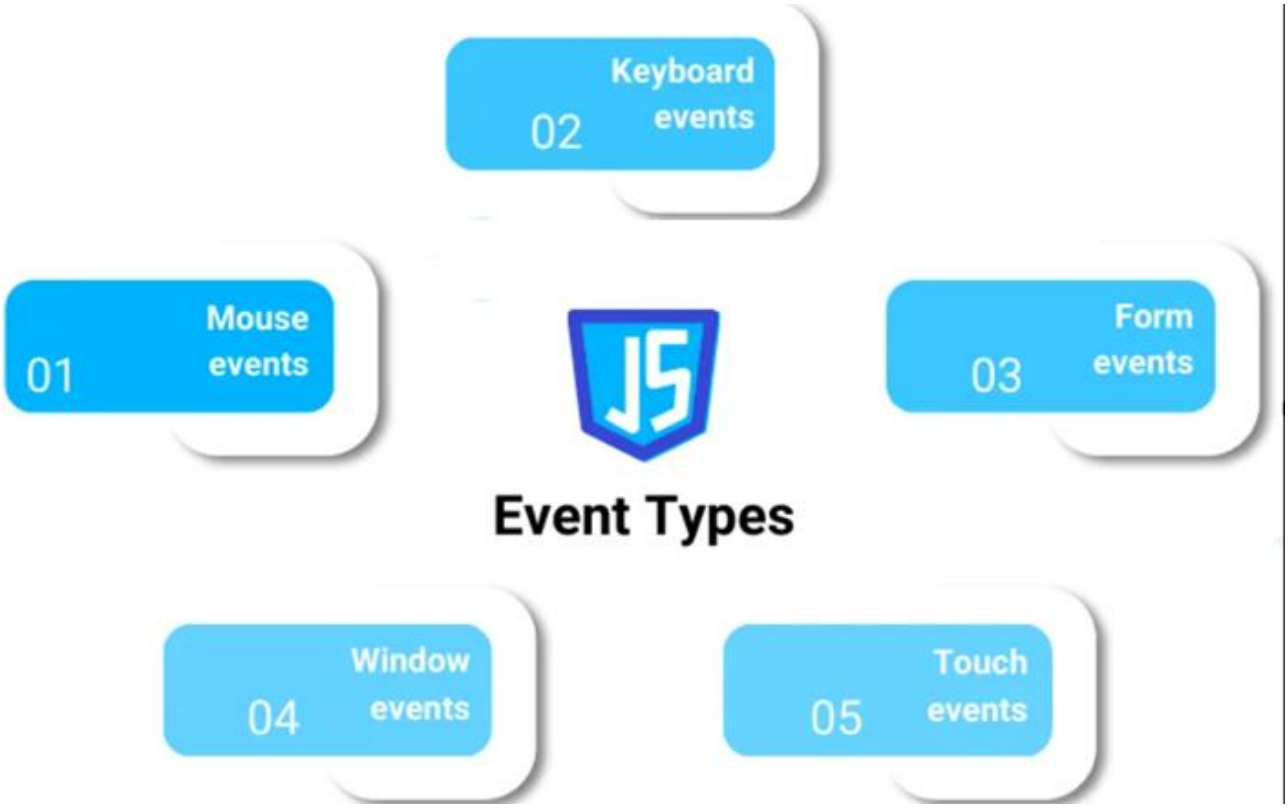
1.10. Principal listado de eventos

Podemos encontrar los eventos disponibles, en la página oficial de w3schools o en la página oficial de mozilla developer.

Los principales tipos de eventos son:

- ✓ **Eventos de ratón**: se incluyen aquí los provocados en pantallas táctiles: click, dblclick, mousedown, mouseup, mousecenter, mouseleave, mousemove, mouseover, mouseout, contextmenu.
- ✓ **Eventos de teclado**: keypress, keydown, keyup
- ✓ **Eventos de movimiento en la ventana**: scroll, resize.
- ✓ **Eventos sobre carga y descarga de elementos**: load, DOMContentLoad, abort, error, progress, readystatechange.

- ✓ **Eventos sobre el historial**: popstate.
- ✓ **Eventos sobre la reproducción de medios**: play, pause, ended, suspend, waiting, playing, canplay…
- ✓ **Eventos de arrastre**: Para que un elemento pueda ser "arrastrable" debe incluir el atributo "draggable" = true. Sus eventos son: dragstart, drag, dragstop. Por otro lado, los eventos del elemento destino son: dragenter, dragover, dragleave, drop (este último necesita eliminar el comportamiento de dragover).
- ✓ **Eventos sobre animaciones y transiciones**: animationstart, animationinteraction, animationend, transitionrun, transitionstart, transitionend.
- ✓ **Eventos del portapapeles**: cut, copy, paste.



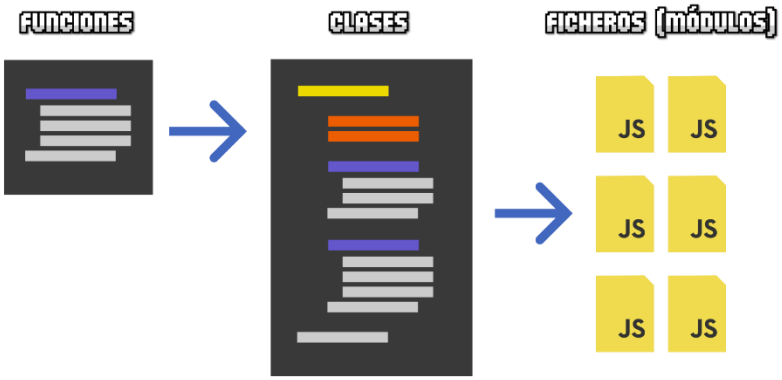
2. MODULOS

A medida que una aplicación web crece y se agregan más funciones, la modularidad del código mejora la legibilidad y el mantenimiento.

Un módulo es un conjunto de funciones, constantes, clases, datos primitivos, objetos y demás elementos que se utilizan como una librería que podremos reutilizar para facilitar el desarrollo de aplicaciones.

Los módulos nos van a permitir de forma eficaz utilizar código de librerías de terceros, como crear código que pueda reutilizarse en otras aplicaciones.

En JavaScript no se utilizaban módulos, estaba fuera de la norma, pero para cubrir la necesidad de reutilización de código y favorecer la modularidad, la comunidad inventó una variedad de formas para organizar el código en módulos o bibliotecas especiales para cargarlo a demanda.



2.1. Creación de módulos

Para poder utilizar código de un archivo a otro, debo importar y exportar la información.

Partimos de un HTML con dos ficheros .js:

```
<script src="js/app.js"></script>
<script src="js/cliente.js"></script>
```

En nuestros ficheros js, en "app.js" queremos mostrar por consola el valor de una variable que está declarada en "cliente.js". Vemos este error:

```
▶Uncaught ReferenceError: nombreCliente is not defined
    at app.js:3:13
```

Este error lo podríamos solucionar invirtiendo el orden en el que hacemos referencia a los ficheros .js, pero aun así nos podemos encontrar con variables repetidas….

Si queremos utilizar libremente la información de otros ficheros en nuestro proyecto a través de **módulos,** debemos:

1. Indicar en el fichero html que ese script al que hacemos referencia es un módulo.

```
<script src="js/app.js" type="module"></script>
```

2. Indicar, a través de la palabra reservada ***export*** el código que permitimos que otros ficheros utilicen.

```
> FICHEROS UNIDAD > 07-Modulos > js > JS cliente.js > ...

export const nombreCliente = 'Natalia Escrivá';
```

3. Indicar, a través de la palabra reservada ***import*** el código que vamos a utilizar de otros ficheros.

```
> FICHEROS UNIDAD > 07-Modulos > js > JS app.js

import { nombreCliente } from "./cliente.js";
console.log(nombreCliente);
```

Si nos fijamos, ya no necesitamos incluir en el html todos los ficheros que vamos a utilizar. Simplemente importaremos código a través de los módulos.

2.2. Importar/Exportar variables

Podemos exportar e importar varias variables a la vez.

Exportaremos cada uno de los elementos y podremos importarlos separados por comas.

En el fichero “cliente.js” …

```
export const nombreCliente = 'Natalia Escrivá';
export const tipoCliente = 'VIP';
```

```
import { nombreCliente, tipoCliente } from "./cliente.js";
console.log(nombreCliente, tipoCliente);
```

```
Natalia Escrivá VIP
```


2.3. Importar/Exportar funciones

Podemos exportar e importar funciones. El procedimiento es el mismo:

En el fichero "cliente.js"…

```
export function mostrarInfo(nombre, tipo){
    return `Cliente: ${nombre}; Tipo: ${tipo}`;
}
```

En el fichero "app.js"…

```
import { nombreCliente, tipoCliente, mostrarInfo } from "./cliente.js";

console.log(mostrarInfo(nombreCliente,tipoCliente));
```

```
Cliente: Natalia Escrivá; Tipo: VIP
```

2.4. Importar/Exportar clases

De la misma forma que hemos hecho con variables y funciones, lo haremos con las clases:

En el fichero "cliente.js"…

```
export class Cliente{
    constructor(nombre, tipo, ahorro){
        this.nombre = nombre;
        this.tipo = tipo;
        this.ahorro = ahorro;
    }
    mostrarInfo(){
        return `Cliente: ${this.nombre}; Tipo: ${this.tipo}; Saldo: ${this.ahorro}`;
    }
}
```

En el fichero "app.js", después de importar la clase…

```
console.log(cliente.mostrarInfo());
```

```
Cliente: Natalia Escrivá; Tipo: VIP; Saldo: 200
```

2.4.1. HEREDAR UNA CLASE IMPORTADA

Podremos heredar una clase en otro fichero y usar esa segunda clase en un tercer fichero.

En el fichero "empresa.js" vamos a crear una subclase de Cliente. Para ello haremos el ***import*** de la clase "Cliente":

```
import { Cliente } from './cliente.js';

export class Empresa extends Cliente {
    constructor(nombre, ahorro, tipo, categoria){
        super(nombre, ahorro, tipo);
        this.categoria = categoria;
    }
}
```

En el fichero "app.js" haremos el ***import*** de la clase "Empresa" y podremos utilizar su código.

```
const empresa = new Empresa('Botica Mariano', 25000, 'Silver', 'ParaFarmacia');
console.log(empresa);
console.log(empresa.mostrarInfo());
```

```
▼ Empresa {nombre: 'Botica Mariano', tipo: 'Silver', ahorro: 25000, categoria: 'ParaFarmacia'}   app.js:15
    ahorro: 25000
    categoria: "ParaFarmacia"
    nombre: "Botica Mariano"
    tipo: "Silver"
  ▶ [[Prototype]]: Cliente
Cliente: Botica Mariano; Tipo: Silver; Saldo: 25000   app.js:16
```

2.5. Export default

Los ejemplos que estamos viendo incluyen la palabra reservada **export** por cada bloque de código o elemento que permitimos que ser reutilice (variables, funciones y clases).

Podemos utilizar también ***export default***.

Su función es la misma que export pero cuando la vayamos a importar, NO la incluiremos en las llaves.

Debemos tener algo muy presente: **SOLO HABRÁ UN EXPORT DEFAULT por fichero.**

Ejemplo 1 → export default function

En el fichero "cliente.js" vamos a definir una **función** con ***export default***:

```
export default function funcionExportDefault(){
    console.log('Ejemplo de export default');
}
```

En el fichero "app.js" la importamos y usamos:

```
import funcionExportDefault, { ahorro, mostrarInfo, nombreCliente, tipoCliente, verificarSaldo, Cliente } from "./cliente.js";
import { Empresa } from './empresa.js';
```

```
funcionExportDefault();
```

```
Ejemplo de export default
```

Podemos usar un alias, y se interpretará sin problemas (porque solo hay uno).

```
import miFuncion, { ahorro, mostrarInfo, nombreCliente, tipoCliente, verificarSaldo, Cliente } from "./cliente.js";
```

```
miFuncion();
```

```
Ejemplo de export default
```

Podríamos incluso declarar la función sin nombre y exportarla con nuestro alias

```
export default function (){
    console.log('Ejemplo de export default');
}
```

La importación y el uso de la función, así como el resultado, será igual que el anterior.

Ejemplo 2 → export default class

En el fichero "cliente.js" vamos a definir una **clase** con ***export default***:

```
export default class Cliente{
    constructor(nombre, tipo, ahorro){
        this.nombre = nombre;
        this.tipo = tipo;
        this.ahorro = ahorro;
    }
    mostrarInfo(){
        return `Cliente: ${this.nombre}; Tipo: ${this.tipo}; Saldo: ${this.ahorro}`;
    }
}
```

Debemos modificar todos los imports que hay en otros ficheros.

En el fichero "empresa.js"

```
import Cliente from './cliente.js';
```

En el fichero "app.js":

```
import miCliente, { funcionExportDefault, ahorro, mostrarInfo, nombreCliente, tipoCliente, verificarSaldo } from "./cliente.js
```

```
const cliente = new miCliente(nombreCliente, tipoCliente, ahorro);
console.log(cliente.mostrarInfo());
```

```
Cliente: Natalia Escrivá; Tipo: VIP; Saldo: 200
```

Ejemplo 3 → export default variable;

En el fichero "cliente.js" vamos a definir una **variable** con ***export default***:

```
const ahorro = 200;
export default ahorro;
```

En el fichero "app.js"

```
import ahorro, { mostrarInfo, nombreCliente, tipoCliente, verificarSaldo, Cliente } from "./cliente.js";
```


2.6. Alias

Cuando estamos usando módulos, podemos utilizar alias para renombrar lo que estamos importando.

Ya hemos visto que, cuando utilizamos ***export default***, podemos usar un alias directamente pero si usamos ***export*** necesitaremos la palabra reservada ***as***.

```
import miFuncion, { ahorro, mostrarInfo, nombreCliente, tipoCliente as nivel, verificarSaldo, Cliente } from "./cliente.js";
```

En el ejemplo, hemos renombrado **tipoCliente** como **nivel.** Lo usaremos como nivel en todo nuestro código.

```
const cliente = new Cliente(nombreCliente, nivel, ahorro);
console.log(cliente.mostrarInfo());
```

```
Cliente: Natalia Escrivá; Tipo: VIP; Saldo: 200
```

2.7. Otras formas de exportar/importar

Hemos visto que cada elemento que queremos exportar va precedido de la palabra reservada **export**.

Hay otras opciones para expresar la exportación de nuestro código.

Ejemplo 1 → Exportamos varios elementos, entre llaves. Podremos mezclar variables, funciones…

En el fichero "cliente.js" hemos declarado unas variables y las queremos exportar.

```
const nombreCliente = 'Natalia Escrivá';
const tipoCliente = 'VIP';
const ahorro = 200;
```

```
export{
    nombreCliente,
    tipoCliente,
    ahorro
}
```

Ejemplo 2 → Importamos todo el fichero

En el fichero "app.js" queremos importar todo el archivo "cliente.js". Le ponemos un alias

```
import * as Cliente from "./cliente.js";
```

Debemos tener en cuenta que usaremos el alias cada vez que hagamos referencia a un elemento del fichero que hemos importado.

```
console.log(Cliente.nombreCliente, Cliente.tipoCliente);
console.log(Cliente.mostrarInfo(Cliente.nombreCliente,Cliente.tipoCliente));
```

```
const cliente = new Cliente.Cliente(Cliente.nombreCliente, Cliente.nivel, Cliente.ahorro);
console.log(cliente.mostrarInfo());
```


3. LOCALSTORAGE Y SESSIONSTORAGE

Pertenecen al objeto global Window y permiten el **almacenamiento de datos relacionados con las sesiones de los navegadores en local.**

Su finalidad es NO depender del uso de las cookies para realizar esta labor.

Almacenan información de tipo clave-valor, pero con menos restricciones y más ventajas que las cookies.

Los datos que se almacenan NO se envían al servidor en ningún momento, se quedan el equipo del cliente. NO forman parte de ninguna petición ni respuesta http (como sí ocurre en las cookies).

3.1. LocalStorage

Si abrimos una ventana de Chrome con un buscador cualquiera y en la consola ponemos window.localStorage, vemos **una única propiedad (length)** con valor porque no tiene nada previsto para guardar.

Solo podremos almacenar Strings, ningún otro tipo de dato es almacenado por Web Storage.

Guardará los datos en el equipo del usuario SIN caducidad.

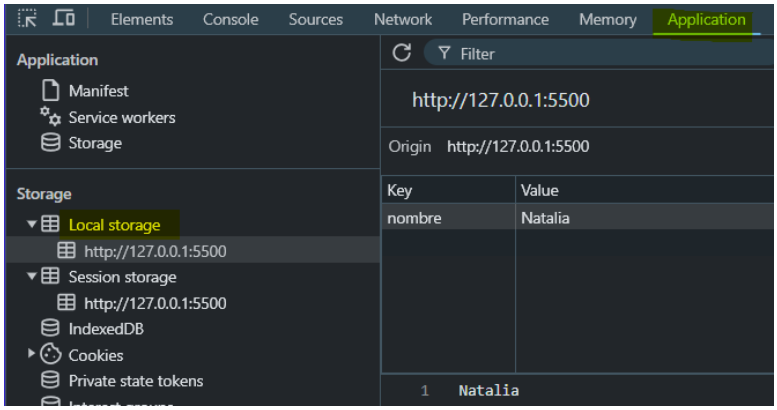
NO guardaremos datos sensibles en localStorage, para eso utilizaremos el backend de nuestro proyecto que gozará de un mayor nivel de seguridad, pero lo podremos usar para ofrecer al usuario una mejor experiencia, que de eso se encarga el frontend.

Para **añadir o editar** elementos a este almacenamiento, usamos *setItem('clave', valor)*

```
localStorage.setItem('nombre','Natalia');
console.log(window.localStorage);
```

```
▼ Storage {nombre: 'Natalia', length: 1} i
    nombre: "Natalia"
    length: 1
  ▶ [[Prototype]]: Storage
```

Si los quiero consultar en el navegador…. En Chrome lo buscamos en Application y en Firefox en Almacenamiento:

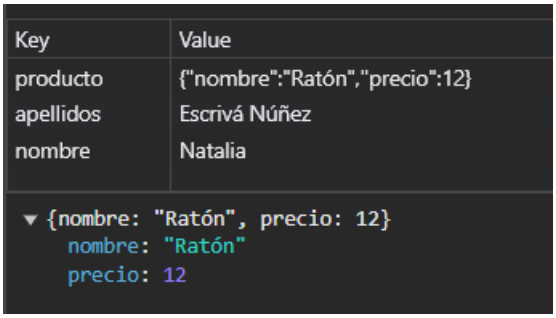


También podemos pasar el valor de variables:

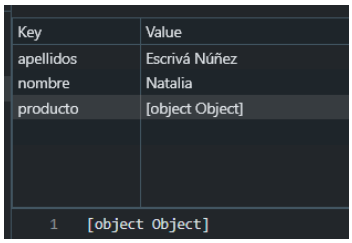
```
const apellidos = "Escrivá Núñez";
localStorage.setItem('apellidos', apellidos);
```

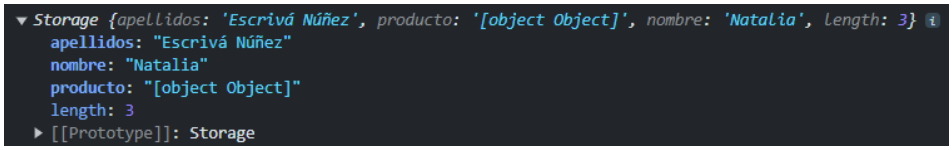
Si necesitamos almacenar algún **elemento que NO es un String** podremos utilizar el método ***JSON.stringify()***

```
const producto = {
    nombre: "Ratón",
    precio: 12
}
const productoString = JSON.stringify(producto);
localStorage.setItem('producto', productoString);
console.log(window.localStorage);
```



Si no utilizamos el método ***JSON.stringify(),*** lo que hará localStorage es almacenar el objeto así:

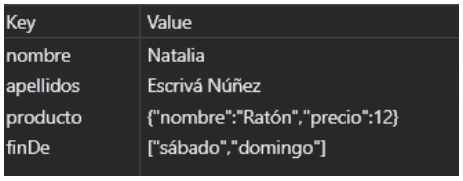


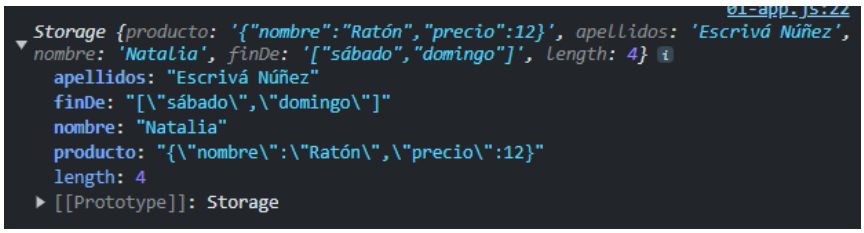


Pasa lo mismo con los **arrays.**

```
const finDeSemana = ['sábado','domingo'];
localStorage.setItem('finDe',JSON.stringify(finDeSemana));
```

Salida utilizando el método ***JSON.stringify()***





Salida SIN utilizar el método JSON.stringify()

Key	Value
apellidos	Escrivá Núñez
nombre	Natalia
producto	{"nombre":"Ratón","precio":12}
finDe	sábado,domingo

```
1   sábado,domingo
```

```
▼ Storage {producto: '{"nombre":"Ratón","precio":12}', apellidos: 'E
    apellidos: "Escrivá Núñez"
    finDe: "sábado,domingo"
    nombre: "Natalia"
    producto: "{\"nombre\":\"Ratón\",\"precio\":12}"
    length: 4
  ▶ [[Prototype]]: Storage
```

Para **consultar los elementos** de este almacenamiento, usamos ***getItem('clave')*** o simplemente ***localStorage.clave***

```
const usuario = `${localStorage.getItem('nombre')} ${localStorage.apellidos}`
console.log(usuario);
```

```
Natalia Escrivá Núñez
```

Si el elemento que buscamos NO existe, NO dará error, devolverá null.

Si necesitamos obtener algún **elemento que NO era un String** podremos utilizar el método ***JSON.parse()*** para volver a tener el objeto o el array y trabajar con ellos.

```
const articulo = JSON.parse(localStorage.getItem('producto'));
console.log(articulo);
const dias = JSON.parse(localStorage.finDe);
console.log(dias);
```

```
▼ {nombre: 'Ratón', precio: 12}
    nombre: "Ratón"
    precio: 12
  ▶ [[Prototype]]: Object
▼ (2) ['sábado', 'domingo']
    0: "sábado"
    1: "domingo"
    length: 2
  ▶ [[Prototype]]: Array(0)
```

Para **eliminar elementos** de este almacenamiento, usamos ***removeItem('clave')***

```
localStorage.removeItem('apellidos');
```

Key	Value
producto	{"nombre":"Ratón","precio":12}
finDe	["sábado","domingo"]
nombre	Natalia

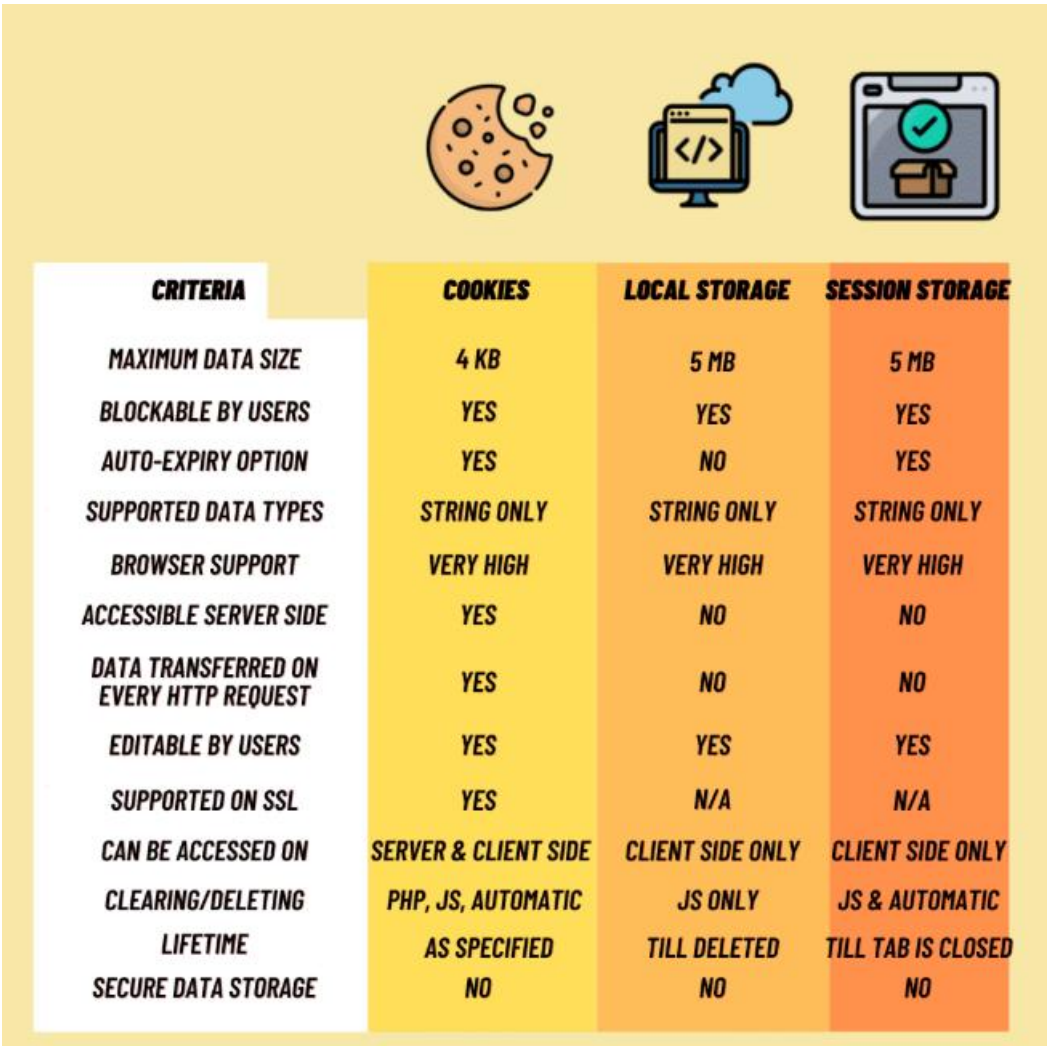
Para **eliminar todos los elementos** de este almacenamiento, usamos *localStorage.clear()*

3.2. SessionStorage

Los datos que se almacenan se pierden cuando salimos del navegador, es decir, son de sesión.

```
const puntos = prompt('Puntúa la experiencia del 1 al 10');
sessionStorage.setItem('puntuaciónWeb', puntos);
```

Cada vez que cierre el navegador esa información se perderá.



CRITERIA	COOKIES	LOCAL STORAGE	SESSION STORAGE
MAXIMUM DATA SIZE	4 KB	5 MB	5 MB
BLOCKABLE BY USERS	YES	YES	YES
AUTO-EXPIRY OPTION	YES	NO	YES
SUPPORTED DATA TYPES	STRING ONLY	STRING ONLY	STRING ONLY
BROWSER SUPPORT	VERY HIGH	VERY HIGH	VERY HIGH
ACCESSIBLE SERVER SIDE	YES	NO	NO
DATA TRANSFERRED ON EVERY HTTP REQUEST	YES	NO	NO
EDITABLE BY USERS	YES	YES	YES
SUPPORTED ON SSL	YES	N/A	N/A
CAN BE ACCESSED ON	SERVER & CLIENT SIDE	CLIENT SIDE ONLY	CLIENT SIDE ONLY
CLEARING/DELETING	PHP, JS, AUTOMATIC	JS ONLY	JS & AUTOMATIC
LIFETIME	AS SPECIFIED	TILL DELETED	TILL TAB IS CLOSED
SECURE DATA STORAGE	NO	NO	NO

4. ANEXO: FORMULARIOS

Los formularios constituyen la base de la comunicación de las aplicaciones web con el usuario.

Hasta ahora nos han sido útiles los métodos del objeto **window alert, prompt y confirm**, pero en la práctica, las aplicaciones profesionales **NO las utilizan**.

Si hay que mostrar mensajes se pueden utilizar elementos HTML como capas, paneles.

La introducción de datos por parte del usuario se hace siempre con formularios.

El objeto **document** dispone de una **propiedad** llamada **forms** que devuelve una **colección con todos los formularios del documento**. De esta forma, **document.forms[0]** será el primer formulario…. En cualquier caso, debemos saber que **la forma más recomendable de acceder al formulario es a través de un identificador.**

Debemos tener en cuenta que **la validación de formularios** desde el lado del cliente, en este caso, con JavaScript **NO garantiza** que los datos enviados son correctos.

Lo que vamos a evitar con esta validación es que se manden datos erróneos y hacer operaciones innecesarias, pero debemos tener claro que **la validación de un formulario se debe realizar en el lado del servidor y en la base de datos.**

4.1. Acceso a los formularios

Tenemos distintas formas de acceder a los formularios de nuestro proyecto.

Partimos de este fragmento HTML:

```
<body>
    <h1>Formulario</h1>
<form action="procesar.php" method="post" id="miFormulario">
```

Podremos acceder a él a través de su nombre o id:

```
let formulario = document.forms.miFormulario;
let form2 = document.forms["miFormulario"];
let form1 = document.getElementById("miFormulario");
```

También podremos acceder al formulario si sabemos cuál es dentro de la página:

```
let form3 = document.getElementsByTagName("form")[0];
let form4 = document.forms[0];
```

4.2. Propiedades y métodos de los formularios

MÉTODO/PROPIEDAD	SIGNIFICADO

elements	Devuelve una colección que contiene todos los controles del formulario.
	`console.log(formulario.elements);`
	`HTMLFormControlsCollection { 0: input#nombre , 1: input#telefono , 2: input#dia , 3: input#mes , 4: input#ano , 5: input , 6: input , 7: input#mayor , 8: input#enviar , 9: input#borrar , … }`

length	Devuelve el número de controles del formulario. `console.log(form2.length); //10`
action	Devuelve el contenido de la propiedad action, que marca la URL destino de los datos del formulario. `console.log(form3.action);` `http://127.0.0.1:5500/procesar.php`
method	Devuelve el contenido de la propiedad method del formulario, que marca la forma de envío de los datos (get o post). `console.log(form4.method);//post`
submit()	Envía los datos del formulario a su destino `<input type="submit" value="Enviar" id="enviar"/>`
reset()	Deja los valores de los controles del formulario a su estado por defecto. `<input type="reset" value="Borrar" id="borrar"/>`

Respecto a **elements**, podemos encontrar controles como radiobutton o checkbos que tengan el mismo nombre… Veamos el ejemplo HTML:

```
<input type="radio" name="sexo" value="H" /> Hombre
<input type="radio" name="sexo" value="M" /> Mujer
```

Seleccionamos el elemento del formulario:

```
let sexoElem = formulario.elements.sexo;
console.log(sexoElem);
```

```
RadioNodeList { 0: input , 1: input , value: "", length: 2 }
  0: <input type="radio" name="sexo" value="H">
  1: <input type="radio" name="sexo" value="M">
  length: 2
  value: ""
```

Podremos recorrer ese elemento y ver sus valores o trabajar con ellos:

```
for (let i=0; i<sexoElem.length; i++){
    console.log(sexoElem[i].value);
}
```

```
H
M
```

4.3. Propiedades de los controles

PROPIEDAD	SIGNIFICADO
name	Nombre del control: prácticamente todos los controles
type	Valor del atributo type: controles de tipo input
value	Devuelve el valor actual del control: prácticamente todos los controles
checked	Puede valer true o false. Indica sobre un botones de tipo radio o tipo check si está clicado o no.
disabled	Indica, con true o false, si el control está deshabilitado o no. Prácticamente todos los controles.
readonly	Indica, con true o false, si el control es de solo lectura o no. Prácticamente todos los controles.
required	Indica, con true o false, si es obligatorio o no cambiar el valor de control Prácticamente todos los controles. Prácticamente todos los controles
min /max	Valor mínimo/máximo posible para el control: Input de tipo numérico o de fecha.
pattern	Patrón a seguir para validar un control.

4.4. Métodos de los controles

MÉTODO	SIGNIFICADO
focus()	Fuerza a que el control obtenga el foco: prácticamente todos los controles
blur()	Provoca la pérdida del control del foco.

4.5. Validación de los formularios

Podemos realizar validaciones de los formularios en el lado del cliente de distintas formas: solo con HTML5 y también con JavaScript.

Veamos algunos ejemplos de validación de JavaScript:

Partimos de este elemento HTML:

```
<input type="text" name="nombre" id="nombre" value = "Natalia Escrivá" />
```

```
//Validamos los campos del formulario a través de varias funciones
function validaNombre(){
    //accedemos al elemento del formulario
    let elem = form1.elements.nombre;
    let exp = /^[A-Z][a-záéíóúñ]+( [A-Z][a-zñáéíóú]+)+$/;
    //Por si lo hemos escrito mal antes, se lo quitamos
    limpiarError(elem);
    if (elem.value == ""){
        alert("El campo 'nombre' NO puede estar vacío");
        error(elem);
        return false;
    }
    if (exp.test(elem.value) == false){
        alert("El campo 'nombre' debe ser correcto");
        error(elem);
        return false;
    }
    return true;
}
```

```
//Cuando hay errores...
function error(elemento) {
    elemento.className = "error";
    elemento.focus();
}
//si tenemos un error y ya lo ponemos bien, quitamos esa clase
function limpiarError(elemento){
    elemento.className = "";
}
```

Veamos otro ejemplo, con un control distinto. Partimos de este HTML:

```
<input type="checkbox" name="mayor" id="mayor" checked/>
```

Y lo validamos así:

```
function validaEdad(){
    let check = document.getElementById("mayor");
    if (!check.checked){
        alert("Debe ser mayor de edad");
        return false;
    }
    return true;
}
```

4.6. API de validación de los formularios

Vamos a utilizar el método **checkValidity()**, que lo que valida es si el elemento HTML tiene restricciones y las cumple.

Este método tiene unas propiedades:

PROPIEDAD	SIGNIFICADO
validity	Propiedad booleana que valora la validez de otras propiedades.
validationMessage	Contiene el mensaje que mostrará el navegador cuando la validez sea falsa

La propiedad que evalúa la validez de un elemento de HTML (validity) tiene, a su vez, unas propiedades booleanas:

PROPIEDAD	SIGNIFICADO
valueMissing	Será true si un elemento requerido no tiene valor.
patternMismatch	El valor del elemento NO coincide con el patrón
rangeUnderflow	El valor de un elemento es menor que el mínimo requerido
rangeOverflow	El valor de un elemento es mayor que el máximo requerido
typeMismatch	El valor del elemento no es válido por el tipo de atributo

Ejemplo → Aquí tenemos el HTML

```
<td>Nombre*: </td>
<td>
    <input type="text" name="nombre" id="nombre" maxlength="15" pattern="[A-Za-z ]{2,15}"
    title="Introduce entre 2 y 15 letras" required/>
</td>
```

Vamos a ver la validación.

```
function validaNombre() {
    //let elemento = form.elements.nombre;
    let elemento = document.getElementById("nombre");
    if (!elemento.checkValidity()) {
        //PRIMER EJEMPLO DE VALIDACIÓN: con validationMessage
        error(elemento);
        return false;
    }
    return true;
}
```

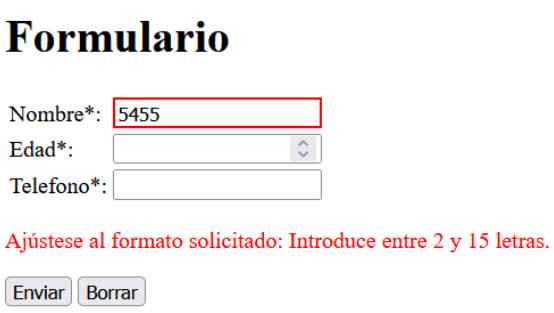
Utilizamos **validationMessage** para mostrar los mensajes de error. No decidimos nosotros el contenido del mensaje, lo tiene el navegador.

```
function error(elemento) {
    //validationMessage: mensaje propio del navegador, NO lo configuramos nosotros
    document.getElementById("mensajeError").innerHTML = elemento.validationMessage;
    elemento.className = "error";
    elemento.focus();
}
```

Veamos qué ocurre cuándo ejecutamos:

Si dejamos el campo nombre vacío:

Si no cumplimos el patrón:



Ahora vamos a seguir utilizando **checkValidity()** pero sin los mensajes predeterminados. Vamos a introducir mensajes personalizados atendiendo al tipo de error.

Partiendo del mismo HTML, veamos la validación de nombre:

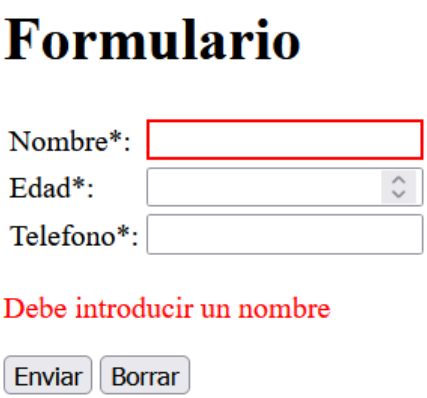
```
function validaNombre() {
    //let elemento = form.elements.nombre;
    let elemento = document.getElementById("nombre");
    if (!elemento.checkValidity()) {
        //PRIMER EJEMPLO DE VALIDACIÓN: con validationMessage
        //error(elemento);
        //SEGUNDO METODO DE VALIDACIÓN: con JS
        if (elemento.validity.valueMissing) {
            error2(elemento, "Debe introducir un nombre")
        }
        if (elemento.validity.patternMismatch) {
            error2(elemento, "El nombre debe tener entre 2 y 15 caracteres");
        }
        return false;
    }
    return true;
}
```

Vemos dos de las propiedades mencionadas de **validity**: **valueMissing** y **patternMismatch**.

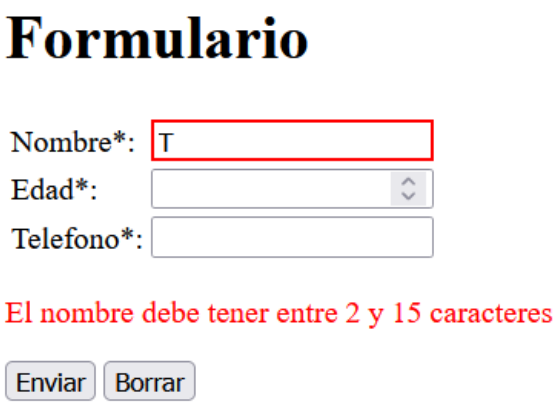
```
function error2(elemento, mensaje) {
    document.getElementById("mensajeError").innerHTML = mensaje;
    elemento.className = "error";
    elemento.focus();
}
```

ación:

Dejamos el campo sin valor:



No coincide con el patrón:



Vamos a validar ahora la edad. Partimos de este HTML:

```
<td>Edad*: </td>
<td>
    <input type="number" name="edad" id="edad" min="18" max="100" required/>
</td>
```

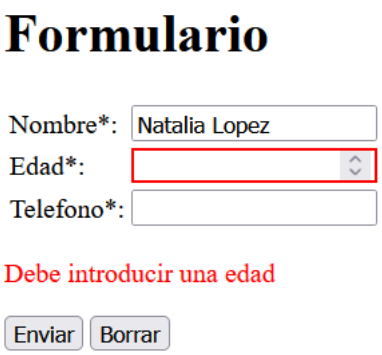
Veamos el JS:

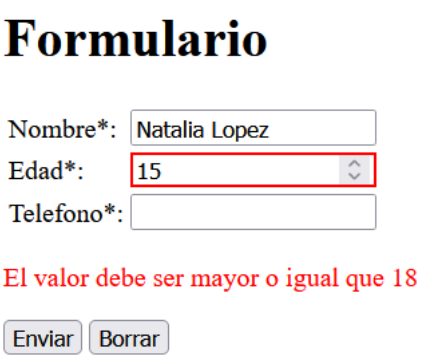
```
function validaEdad() {
    let elemento = document.getElementById("edad");
    if (!elemento.checkValidity()) {
        //PRIMER EJEMPLO DE VALIDACIÓN: con validationMessage
        //error(elemento);
        //SEGUNDO METODO DE VALIDACIÓN: con JS
        if (elemento.validity.valueMissing) {
            error2(elemento, "Debe introducir una edad")
        }
        if (elemento.validity.rangeOverflow) {
            error2(elemento, "El valor debe ser menor de 100")
        }
        if (elemento.validity.rangeUnderflow) {
            error2(elemento, "El valor debe ser mayor o igual que 18");
        }

        return false;
    }
    return true;
}
```

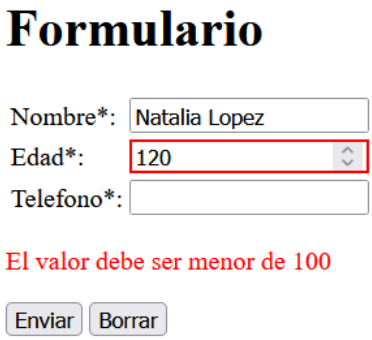
Aquí vemos otras dos propiedades de **validity: rangeOverflow** y **rangeUnderflow.**

Lo ejecutamos:

Dejamos el campo vacío o ponemos un valor fuera de rango:







Más información sobre formularios y su validación:

- https://www.w3schools.com/js/js_validation.asp
- https://www.w3schools.com/js/js_validation_api.asp

5. ANEXO: COOKIES

Las **cookies** son archivos de texto que las aplicaciones web guardan en el navegador que contienen **datos que la aplicación puede volver a recuperar.**

Cada navegador tendrá unas cookies independientes. El **protocolo http** es un protocolo sin estado, por lo que cada petición es distinta al anterior.

Las **cookies se envían automáticamente en la cabecera de cada petición** permitiendo a las aplicaciones recordar aspectos de peticiones anteriores gracias a los datos que han grabado las cookies. Por ejemplo, cuando accedemos a un sitio web e indicamos algunos datos (nombre, idioma y ubicación) y después cerramos el navegador, el sitio pierde la información relacionada con la visita y, cada vez que entramos, nos la tendría que volver a pedir.

Esa "pérdida" de los datos se soluciona con las cookies, que, como tiene grabados esos datos, una vez que vuelvas a la aplicación, aparecen, o permiten recuperar un carrito sin acabar de confirmar. Es decir, **permiten recordar la información de un usuario, aunque se cierre el navegador o se desconecte del servidor.**

¿Qué se puede almacenar? Todo: información del usuario, preferencias de sesión (colores…), número de veces que ha entrado a la página, carrito... Hay cookies del lado del cliente y del servidor.

Las cookies **tienen fecha de expiración.** Si no decimos nada, caducan cuando se cierre el navegador.

La **UE incluye una directiva** relativa al uso de las cookies que obliga a las aplicaciones web a pedir permiso al usuario antes de poder grabarlas. Este aviso debe incluir una explicación (o enlace al mismo) que aclare la utilización y necesidad de las cookies.

Esta directiva distingue entre dos **tipos de cookies**:

- ✓ Las estrictamente necesarias: Imprescindibles para el correcto funcionamiento de la aplicación, como son datos de inicio de sesión para que se acceda por ejemplo a un carrito de compra o un espacio personal.
- ✓ Las técnicamente necesarias: Se permiten grabar y, tras pedir pedir al usuario, se deberán borrar o se podrán mantener. Están son las de análisis de la acción del usuario o de seguimiento.

El excesivo o mal uso de las cookies les ha creado mala fama, por lo que muchas aplicaciones recurren a otras opciones como la **API localStorage**, que ofrece más posibilidades para almacenar datos.

5.1. Crear y modificar una cookie

Es el **objeto document** el que posee la propiedad cookie que es la que controla las cookies.

La información de la cookie se guarda en formato "nombre = valor" y, seguido de punto y coma otros parámetros, por ejemplo, fechas de expiración (formato UTC), la ruta, el dominio….

- ✓ **name**: nombre de la cookie, su identificador
- ✓ **expires**: fecha de caducidad, en formato UTC

- ✓ **max-age**: número de segundos que la cookie estará activa
- ✓ **path**: se suele poner la barra que indica el directorio raíz del dominio, ya que una cookie solo podrá ser leída por aplicaciones del dominio que creó la cookie y que estén en el mismo directorio.
- ✓ **domain**: en el que se crea la cookie. Usaremos uno general, ya que si la grabamos en un subdominio y queremos que otro subdominio lea esa cookie, por defecto NO podrá. Usaremos empresa.net y NO practicas.empresa.net

Ejemplo de cookie de usuario que caducará o se borrará cuando el navegador se cierre.

```
document.cookie = "usuario = Natalia Escrivá;";
```

Ejemplo de cookie de usuario con fecha de caducidad.

```
document.cookie = "usuario = Natalia Escrivá; expires = Sun, 01 Jan 2023 12:00:00 GMT; path =/;";
```

Modificar una cookie NO es más que sobre escribirla, por ejemplo:

```
document.cookie = "usuario = Natalia Escrivá Nunez;";
```

Si modificamos una cookie deberemos respetar TODOS los parámetros utilizados, de lo contrario, el navegador entenderá que se trata de una cookie distinta

5.2. Leer una cookie

Podemos hacer referencia a las cookies. Si tenemos más de una, las veremos como cadenas separadas por puntos y comas.

Partiendo de este código

```
document.cookie = "usuario = Natalia Escrivá;";
document.cookie = "sexo = mujer;";
```

```
let miCookie = document.cookie;
console.log(miCookie);
```

Así es como lo muestra el navegador:

```
usuario=Natalia Escrivá; sexo=mujer
```

5.3. Borrar una cookie

Podremos borrar una cookie poniendo como fecha de expiración una fecha pasada.

```
document.cookie = "usuario = Natalia Escrivá Nunez; expires=Thu, 01 Jan 1970 00:00:01 GMT";
document.cookie = "sexo = mujer; expires=Thu, 01 Jan 1970 00:00:01 GMT";
console.log(document.cookie);
```

```
<empty string>
```

Es importante saber que algunos navegadores no borran las cookies si no ponemos la ruta.