



UT05: INTERACCIÓN CON EL USUARIO. EVENTOS Y FORMULARIOS

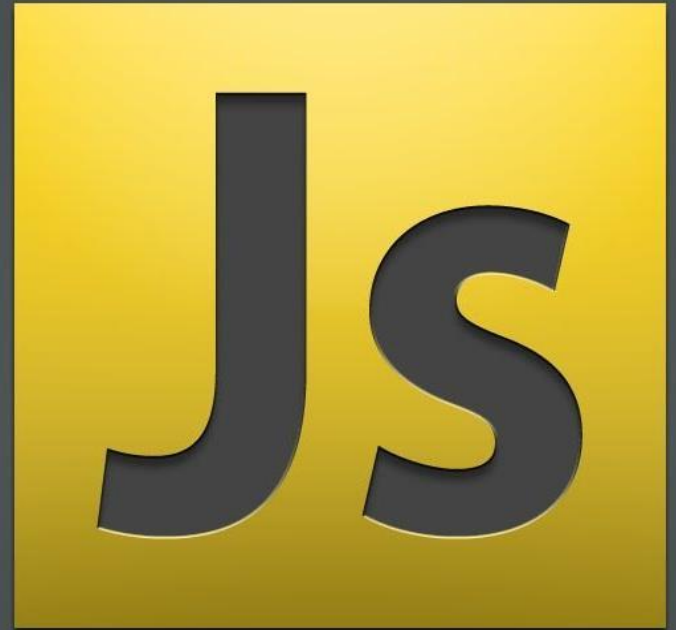
ÍNDICE

- 1.- Introducción a los eventos
- 2.- Propagación y captura. Delegación de eventos
- 3.- Eventos del ratón
- 4.- Eventos del teclado
- 5.- Propiedades y métodos de formularios
- 6.- Enfocado: enfoque/desenfoque
- 7.- Eventos: change, input, cut, copy y paste
- 8.- Formularios: evento y método submit



1

INTRODUCCIÓN A LOS EVENTOS



Un evento es una señal de que algo ocurrió en un determinado elemento.

Ejemplos de eventos:

- **click**: pulsación de ratón
- **contextmenu**: click derecho sobre un elemento
- **mouseover / mouseout**: el cursor entra o sale del elemento
- **mousedown/mouseup**: el ratón es pulsado o soltado sobre el elemento
- **mousemove**: el ratón se desplaza sobre el elemento.

- **keydown / keyup**: se ha presionado/soltado una tecla
- **submit**: se envía un formulario
- **focus**: el elemento toma el foco (p.e. un input)
- **DOMContentLoaded**: se ha cargado completamente el DOM
- **transitionend**: una animación CSS concluye

Los eventos se controlan mediante *handlers*, que son funciones que se ejecutan cuando se dispara el evento.

Hay varias formas de asignar un handler a un evento.

Atributo HTML

En el HTML se usa el atributo on<event> y se le pasa como valor el código que se quiere ejecutar.

```
<input value="Click me" onclick="alert('Click!')"  
      type="button">
```

También se puede invocar una función

```
<script>  
  function saluda( name ) {  
    console.log(`Hola ${name}`)  
  }  
</script>  
  
<input type="button" onclick="saluda('Victor')" value="Hi">
```

Propiedad de DOM

Los elementos del DOM con el mismo nombre a la que se puede asignar la función.

```
<input id="elem" type="button" value="Haz click en mí">
<script>
  elem.onclick = function() {
    alert('¡Gracias!');
  };
</script>
```


addEventListener

El problema de los métodos anteriores es que solo podemos asignar un *handler* a cada evento.

Para solucionar este problema está los listener. La sintaxis es la siguiente:

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

```
element.removeEventListener(event, handler);
```

Independientemente del método escogido, la función invocada en el evento recibirá un objeto en el que podremos acceder a información sobre el evento.

Por ejemplo:

- Elemento que disparó el evento
- Coordenadas del ratón al recibir el evento
- Tipo de evento

```
<div id="to-click">Click me!!</div>
<div id="event-data"></div>
```

```
let div = document.querySelector('#to-click');
let divData = document.querySelector('#event-data');

div.addEventListener( 'click', (event) => {
  divData.innerHTML = `
    <p><b>Coordenada X: </b> ${event.pageX}</p>
    <p><b>Coordenada Y: </b> ${event.pageY}</p>
    <p><b>Tipo de evento: </b> ${event.type}</p>
  `
})
```

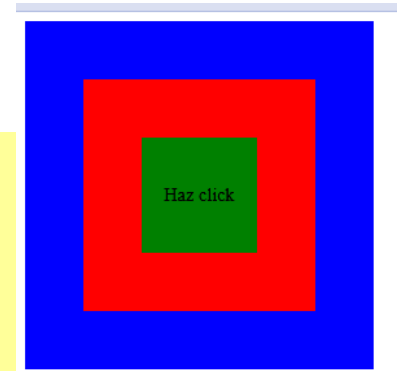
2

PROPAGACIÓN DE EVENTOS



En HTML los elementos están anidados unos dentro de otros de forma que, cuando hacemos click (o cualquier otro evento) dentro de un elemento también lo estamos haciendo en el elemento padre, en el padre del padre y así sucesivamente.

A esto se le llama **propagación de eventos**



```
<div id="primero">
  <div id="segundo">
    <div id="tercero">Haz click</div>
  </div>
</div>
```

```
let divs = document.querySelectorAll('div');

divs.forEach( (div) => {
  div.addEventListener('click', handleClick);
} )

function handleClick(event) {
  console.log(`El evento ha sido capturado por
               #${event.currentTarget.id}`);
}
```

Hay dos propiedades el objeto evento que nos dan información sobre el elemento que recogió el evento.

- **event.target:** elemento que recogió el evento en primer lugar, independientemente de que se haya propagado a otros elementos.
- **event.currentTarget:** elemento que está recogiendo la propagación en este momento.

```
// Modificado del ejemplo anterior
function handleClick(event) {
  console.log(`CurrentTarget: ${event.currentTarget.id}`);
  console.log(`Target:          ${event.target.id}`);
}
```

Salida al hacer
click en el
elemento
interior,
identificado
como #tercero



Filtrar salida

CurrentTarget: tercero

Target: tercero

CurrentTarget: segundo

Target: tercero

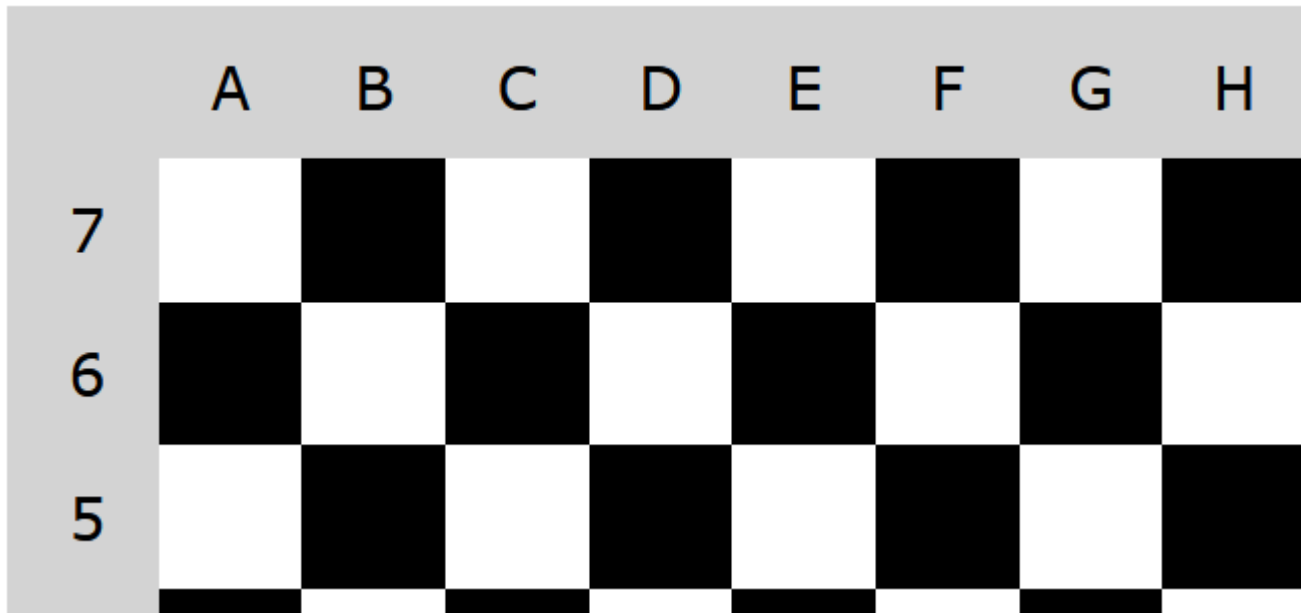
CurrentTarget: primero

Target: tercero



Un efecto muy interesante derivado de la propagación es la **delegación de eventos**.

Vamos a verlo con un ejemplo partiendo de la práctica que hicimos del tablero de ajedrez

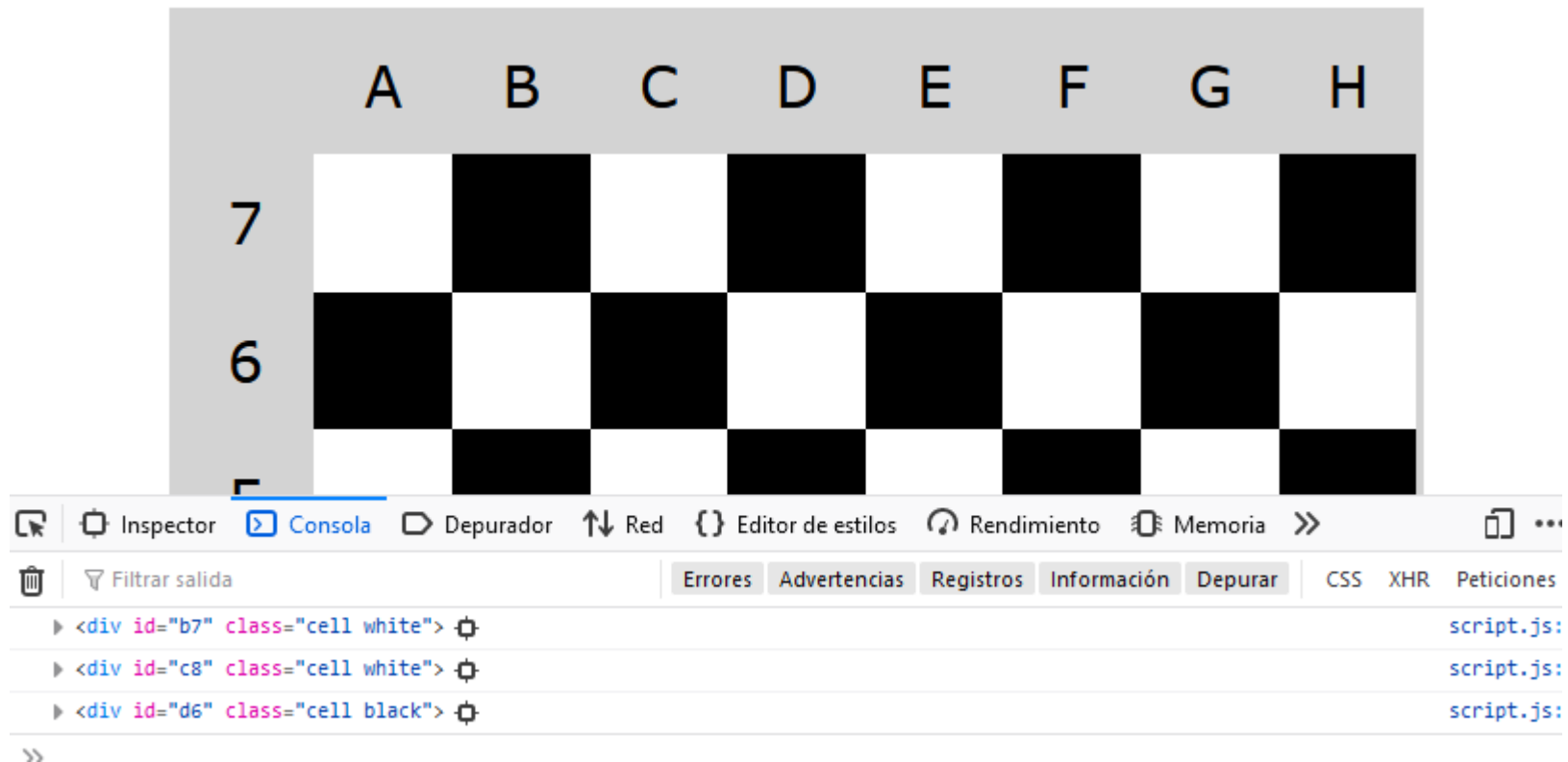


Si queremos seguir desarrollando el juego tendremos que identificar cuando el usuario hace click en una casilla.

Una opción será añadir un *listener* a cada casilla. En este caso ya son 64 *listeners*, pero en otras situaciones pueden ser 1000 o las que sean.

Una alternativa es colocar el *listener* en el elemento padre y verificar cuál ha sido el elemento `event.target`, que es el primero en el que se hizo click.

```
board.addEventListener( 'click', (event) => {  
  console.log(event.target);  
} )
```



De esta forma, con un único *listener* puedo gestionar fácilmente sobre qué elemento se ha hecho click.

Aun así hay dos problemas que habría que solucionar:

- Cuando el elemento padre tiene otros elementos diferentes que no quiero asociar al listener (p.e. la primera fila y columna del tablero)
- Cuando el elemento que quiero tiene otro elemento dentro, que es quien realmente recogerá el listener (p.e. si cada celda tiene dentro un `img` con la figura de ajedrez)

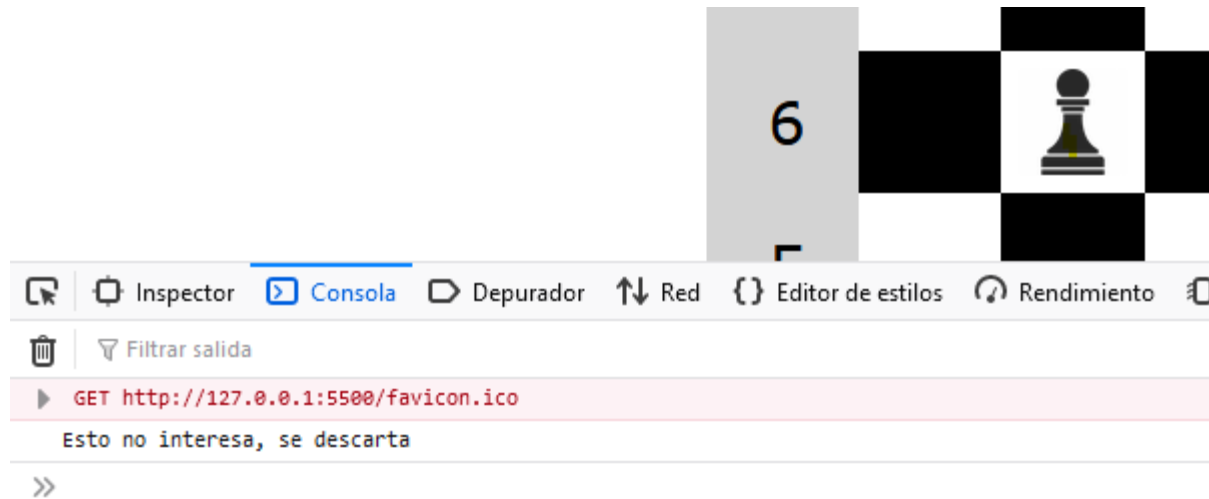


Para el primer problema puedo comprobar en la función algunas de las propiedades de `event.target` para verificar si es el que quiero. Por ejemplo:

- `event.target.tagName`
- `event.target.classList.contains()`

```
board.addEventListener( 'click', (event) => {  
  if (event.target.classList.contains('cell')) {  
    console.log(`Casilla: ${event.target.id}`);  
  } else {  
    console.log('Esto no interesa, se descarta');  
  }  
} )
```





En la imagen anterior podemos ver el segundo problema. Se ha hecho click en el peón y, como target apunta al peón y no a la casilla la descarta.

La solución está en la función *element.closest(selector)*.

Esta función navega hacia arriba en el árbol DOM desde el elemento seleccionado hasta el primer elemento que se ajuste al selector que se pasa como parámetro.

Si no encuentra ninguno devuelve *null*.

```
board.addEventListener( 'click', (event) => {  
  let celda = event.target.closest('.cell');  
  if ( celda ) {  
    console.log(`Casilla: ${celda.id}`);  
  } else {  
    console.log('No ha encontrado ninguna así que se  
descarta');  
  }  
} )
```


Otro ejemplo de delegación

Otro ejemplo de uso de la delegación es cuando queremos hacer un menú.

En lugar de asignar un *handler* a cada entrada del menú se puede usar un atributo personalizado *data-action* que identifique el tipo de acción y un único manipulador asociado a todo el menú.

```
board.addEventListener( 'click', (event) => {  
  let celda = event.target.closest('.cell');  
  if ( celda ) {  
    console.log(`Casilla: ${celda.id}`);  
  } else {  
    console.log('No ha encontrado ninguna así que se  
descarta');  
  }  
} )
```

Abrir

Guardar

Cerrar

Inspector Consola Depurador Red Editor de estilos Rendimiento Memoria



Filtrar salida

Se ha hecho click en ABRIR

Se ha hecho click en CERRAR

Se ha hecho click en GUARDAR

Otro ejemplo: patrón comportamiento

Se puede utilizar la delegación para agregar *comportamiento* a elementos de forma declarativa a través de sus atributos.

El patrón tiene dos partes:

- Se crea un atributo personalizado que describe el comportamiento
- Un manejador asociado a todo el documento verifica si el elemento que dispara el evento tiene dicho atributo.

Ejemplo con un contador.

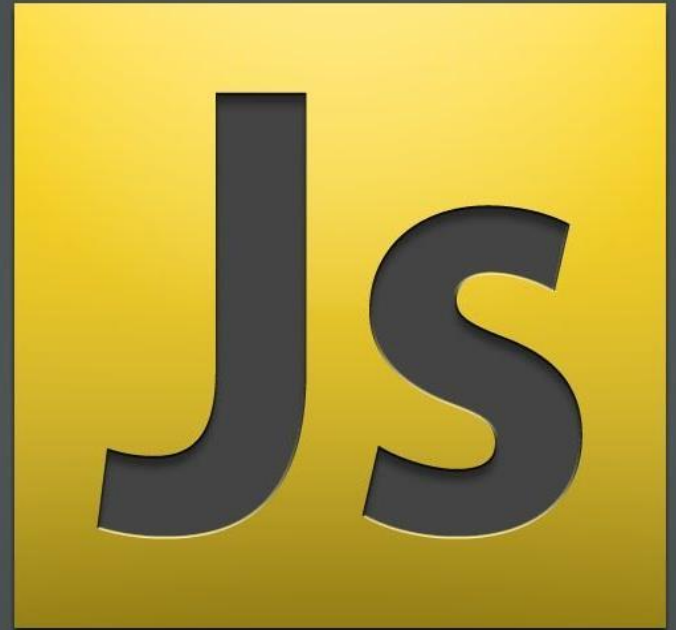
Contador: `<input type="button" value="1" data-counter>`
Otro cont.: `<input type="button" value="2" data-counter>`

```
document.addEventListener('click', function(event) {  
    if (event.target.dataset.counter != undefined) { // si el  
        atributo existe...  
        event.target.value++;  
    }  
});
```

Ahora puedo crear todos los contadores que quiera simplemente añadiendo al HTML el atributo *data-counter*.

3

EVENTOS DEL
RATÓN



Hay múltiples eventos relacionados con el ratón:

- **mousedown/mouseup**: el botón se pulsa/suelta
- **mouseover/mouseout**: el cursor se mueve/sale del elemento
- **mousemove**: el cursor se mueve sobre el elemento
- **click**: se ha hecho click en el elemento. Antes se ha producido un evento mousedown y un mouseup
- **dblclick**: se ha hecho doble click
- **contextmenu**: se ha pulsado el botón derecho del ratón

Los eventos relacionados con la pulsación del botón del ratón siempre tienen una propiedad **`button`**, que indica qué botón ha sido pulsado.

| Estado del botón | <code>event.button</code> |
|----------------------------|---------------------------|
| Botón izquierdo (primario) | 0 |
| Botón central (auxiliar) | 1 |
| Botón derecho (secundario) | 2 |
| Botón X1 (atrás) | 3 |
| Botón X2 (adelante) | 4 |

Estos eventos también tienen información sobre las teclas que han sido pulsadas simultáneamente con el ratón.

```
shiftKey: Shift
```

```
altKey: Alt (p Opt para Mac)
```

```
ctrlKey: Ctrl
```

```
metaKey: Cmd para Mac
```

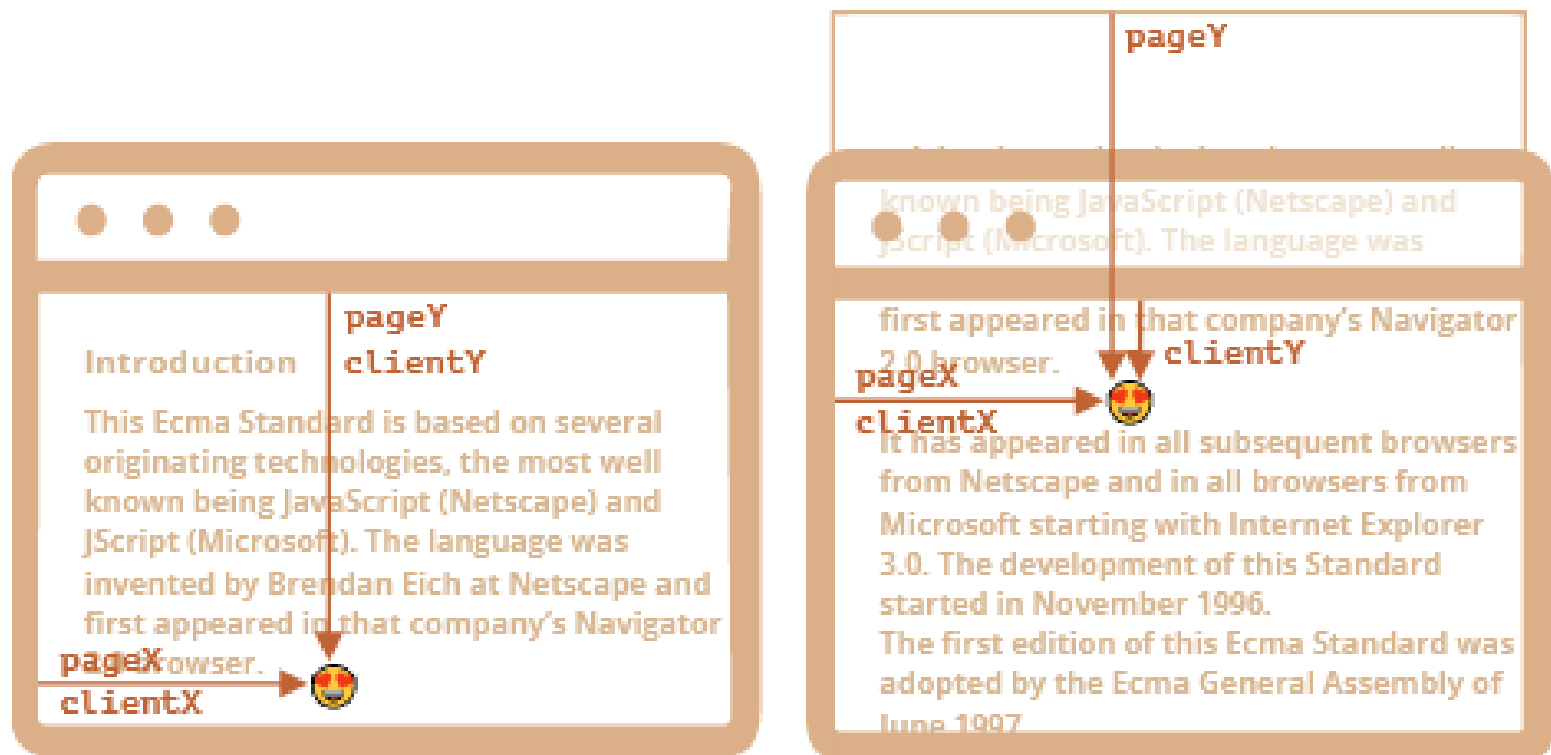
Devuelven un valor booleano que indica si la tecla respectiva ha sido pulsada o no.

Otra información que podemos conseguir de los eventos del ratón son las coordenadas, que pueden ser:

- Relativas a la ventana: **clientX** y **clientY**
- Relativas al documento: **pageX** y **pageY**

Las **relativas a la ventana** indican la distancia a la parte superior izquierda de la ventana de navegación (similar a cuando usamos *position: fixed* en CSS)

Las relativas al documento indican la distancia al principio del documento (en CSS el equivalente es *position: absolute*)



Un efecto indeseable cuando hacemos doble click y click y arrastramos sobre un texto es que dispara el evento, pero también **es seleccionado**.

Para evitar esto la opción más sencilla es

```
<b onclick="alert('click')" onmousedown="return false">  
Haz doble click  
</b>
```

Observa que esto evita que sea seleccionado si se comienza a hacer click sobre el propio elemento, pero si se hace fuera sigue siendo posible seleccionarlo.

También hay ocasiones que no se desea que el usuario pueda copiar el texto de una página web.

En estos casos podemos usar el evento **copy**

```
<b oncopy="alert('No se puede copiar')">  
Intenta copiar este texto  
</b>
```

mouseover y mouseout

Estos eventos son disparados cuando el cursor del ratón se mueve entre elementos.

- **mouseover** se produce cuando el cursor entra al elemento
- **mouseout** se produce cuando el cursor sale del elemento

Estos eventos son especiales porque tienen otra propiedad llamada **relatedTarget**.

Para **mouseover**:

- **event.target** es el elemento al que entra el cursor
- **event.relatedTarget** es el elemento del que viene el cursor

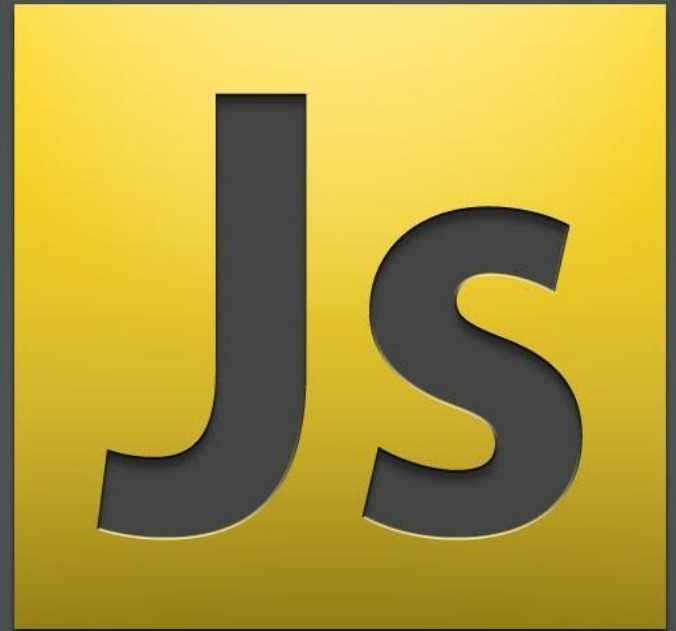
Para **mouseout**:

- **event.target** es el elemento del que salió el cursor
- **event.relatedTarget** es el nuevo elemento al que entró el cursor

NOTA: `relatedTarget` puede ser *null*, p.e. si salimos del navegador, así que hay que tener cuidado con ello

4

EVENTOS DEL TECLADO





















keydown y keyup

Son disparados cuando el usuario pulsa una tecla y cuando la libera respectivamente.

La propiedad **key** permite obtener el carácter pulsado, mientras que la propiedad **code** es el código físico de la tecla.

```
let body = document.querySelector('body');  
  
body.addEventListener( 'keydown', (e) => {  
  console.log(`Key: ${e.key} - Code: ${e.code}`);  
} )
```


Los valores de los códigos los puedes ver en <https://www.w3.org/TR/uievents-code/>

| KeyboardEvent.code | Notes (Non-normative) |
|--------------------|---|
| "Backquote" |  on a US keyboard. This is the  (hankaku/zenkaku/kanji) key on Japanese keyboards |
| "Backslash" | Used for both the US  (on the 101-key layout) and also for the key located between the  and  keys on row C of the 102-, 104- and 106-key layouts. Labelled  on a UK (102) keyboard. |
| "Backspace" |  or  . Labelled  on Apple keyboards. |
| "BracketLeft" |  on a US keyboard. |
| "BracketRight" |  on a US keyboard. |
| "Comma" |  on a US keyboard. |
| "Digit0" |  on a US keyboard. |
| "Digit1" |  on a US keyboard. |
| "Digit2" |  on a US keyboard. |
| "Digit3" |  on a US keyboard. |
| "Digit4" |  on a US keyboard. |
| "Digit5" |  on a US keyboard. |

Algunos ejemplos:

| Tecla | event.key | event.code |
|---------|---------------|------------|
| Z | z (minúscula) | KeyZ |
| Shift+Z | Z (mayúscula) | KeyZ |

| Key | event.key | event.code |
|-----------|-----------|-------------------------|
| F1 | F1 | F1 |
| Backspace | Backspace | Backspace |
| Shift | Shift | ShiftRight or ShiftLeft |

Si el usuario mantiene pulsada la tecla se pueden disparar varios eventos de *keydown*, pero solo uno de *keyup*.

Para poder detectar estas situaciones disponemos de la propiedad **repeat**, cuyo valor es *false* en la primera captura del evento y *true* en las que sean repeticiones.

```
let body = document.querySelector('body');

body.addEventListener( 'keydown', (e) => {
  console.log(`Pulsada tecla ${e.key} - Valor de repeat:
${e.repeat}`)
} )
```

| | | | | | | | |
|--|--------------|-----------|-------------|---------|-----------------------|-----|------------|
| Errores | Advertencias | Registros | Información | Depurar | CSS | XHR | Peticiones |
| Pulsada tecla s - Valor de repeat: false | | | | | javascript.js:4:11 | | |
| Pulsada tecla s - Valor de repeat: true | | | | | 40 javascript.js:4:11 | | |
| Pulsada tecla Alt - Valor de repeat: false | | | | | javascript.js:4:11 | | |

5

FORMULARIOS



Es habitual utilizar JavaScript para trabajar con formularios, por lo que disponemos de un gran número de funciones, propiedades y eventos para acceder a ellos y manipularlos.

document.forms

Esta propiedad contiene un HTMLCollection con todos los elementos `<form>` que hay en el documento.

Es una colección nombrada y ordenada, es decir, se puede acceder a cada uno de los formularios mediante su nombre o mediante su índice.

```
<form name="login">
  <input name="username">
  <input name="password">
</form>
```

```
// Estas dos órdenes son equivalentes
let form = document.forms.login;
let form2 = document.forms[0];
console.log(form);
```

Esta colección tiene múltiples propiedades, una de ellas es **elements**, que contiene todos los elementos del formulario.

```
let form = document.forms.login;  
let username = form.elements.username;  
console.log(username);
```

element.form

Desde cualquier elemento del formulario podemos acceder al formulario que lo contiene con la propiedad **form**.

Elementos `<input>` y `<textarea>`

Para acceder al contenido de un elemento de formulario de tipo `<input>` o `<textarea>` debemos utilizar la propiedad **value**.

Elementos `<select>` y `<option>`

Un elemento `<select>` tiene 3 propiedades importantes:

- **`select.options`**: la colección de subelementos de `<option>`
- **`select.value`**: el valor del `<option>` seleccionado actualmente
- **`select.selectedIndex`**: el número del `<option>` seleccionado actualmente.

```
<select id="select" multiple>
  <option value="blues" selected>Blues</option>
  <option value="rock" selected>Rock</option>
  <option value="classic">Classic</option>
</select>
```

```
// las tres líneas hacen lo mismo
select.options[2].selected = true;
select.selectedIndex = 2;
select.value = 'classic';
```

Algo muy habitual es rellenar las opciones del select de forma dinámica desde JavaScript.

Hay dos opciones para hacerlo:

- Crear un elemento de tipo *option*
- *Utilizar la clase Option*

La primera opción consiste en crear un nodo HTML de tipo *option* y agregarlo al *select*.

```
var option = document.createElement("option");  
option.text = "Text";  
option.value = "myvalue";  
var select = document.getElementById("daySelect");  
select.appendChild(option);
```

La segunda opción es más concisa y consiste en crear un objeto de la clase `Option`.

```
option = new Option(text, value, defaultSelected, selected);
```

Los parámetros son:

- **text:** texto que se mostrará
- **value:** el valor
- **defaultSelected:** si es *true* el atributo HTML *selected* se le crea
- **selected:** si es *true* el option se selecciona.

Por ejemplo:

```
daySelect = document.getElementById('daySelect');  
daySelect.options[daySelect.options.length] = new  
Option('Text 1', 'Value1');
```

Enfoque y desenfoque

Un elemento HTML recibe el foco o se enfoca cuando el usuario hace click sobre él o llega a él pulsando Tab en el teclado.

Hay dos eventos relacionados con el foco:

- **element.focus**: cuando el elemento recibe el foco
- **element.blur**: cuando el elemento pierde el foco

Normalmente se utilizan para realizar validaciones sobre los datos introducidos por el usuario y mostrar algún tipo de error.


```
<style>
  .invalid { border-color: red; }
  #error { color: red }
</style>
```

Su correo por favor:

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

```
input.onblur = function() {
  if (!input.value.includes('@')) { // not email
    input.classList.add('invalid');
    error.innerHTML = 'Por favor introduzca un correo válido.'
  }
};

input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // quitar la indicación "error", porque el usuario quiere reintroducir
    algo
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
```

Además, todos los elementos disponen de dos funciones relacionadas con el foco:

- **element.focus()**: hace que el elemento tome el foco.
- **element.blur()**: quita el foco de un elemento

Por defecto, únicamente algunos elementos pueden recibir el foco, como `<button>`, `<input>`, `<select>`, `<a>`, ...

Sin embargo, otros no pueden recibir el foco, como `<div>`, `` o `<table>`

Si queremos hacer que uno de estos elementos sea enfocable únicamente debemos añadirle el atributo HTML **`tabindex="x"`** donde x hace referencia al orden en que se traslada el foco.

El evento change

Se activa cuando un elemento finaliza un cambio.

En el caso de un cuadro de texto se dispara cuando pierde el foco.

En el caso de elementos `<select>` o checkbox se dispara inmediatamente después de cambiar la opción seleccionada.

El evento input

Se dispara cada vez que un valor es modificado por el usuario.

A diferencia de los eventos de teclado, ocurre con cualquier cambio, por ejemplo, copiar y pegar con el ratón o usar reconocimiento de voz para dictar texto.

Eventos cut, copy y paste

Ocurren al cortar/copiar o pegar un valor.

El evento permite acceder al contenido del portapapeles mediante la propiedad **clipboardData**.

El evento submit

Se activa cuando el formulario es enviado, normalmente se utiliza para realizar comprobaciones antes de su envío al servidor.

El método submit()

Con este método podemos realizar el envío del formulario desde JavaScript

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

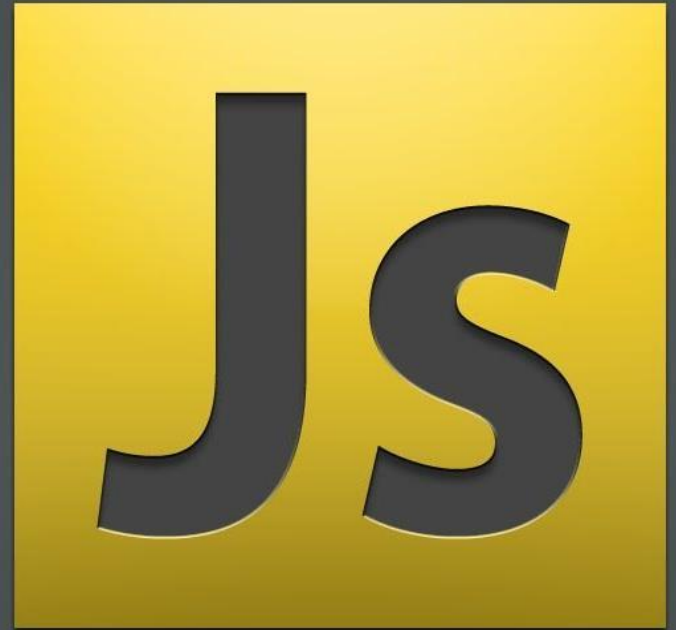
form.innerHTML = '<input name="q" value="test">';

// el formulario debe estar en el document para poder
// enviarlo
document.body.append(form);

form.submit();
```


6

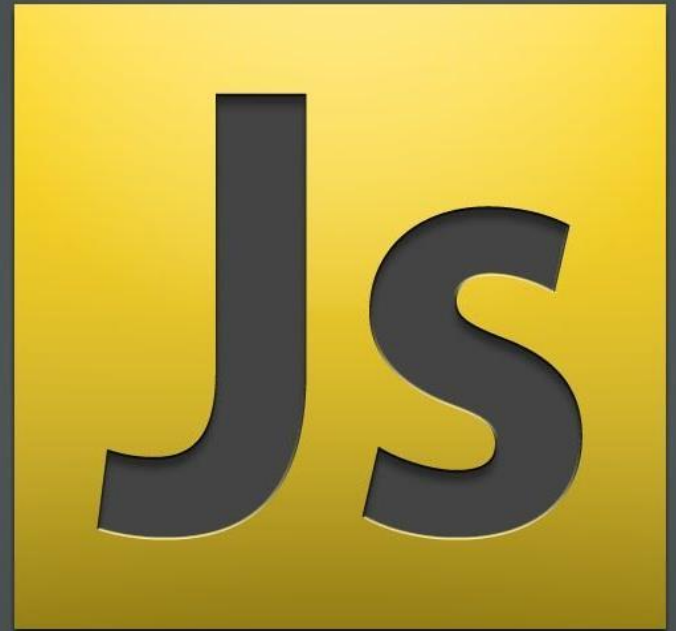
VALIDACIÓN DE FORMULARIOS



https://developer.mozilla.org/es/docs/Learn/Forms/Form_validation

7

EXPRESIONES REGULARES



Las **expresiones regulares** son un mecanismo que permiten identificar patrones de texto en cadenas.

Hay dos sintaxis para crear un objeto de expresión regular:

1.- Utilizando la clase **RegExp**

```
regexp = new RegExp("patrón", "flags");
```

2.- Usando la notación de barras “/”

```
regexp = /pattern/flags;
```

Hay varios **flags**, pero los que vamos a utilizar son:

- **i**: la expresión regular no distingue entre mayúsculas y minúsculas
- **g**: la búsqueda encuentra todas las coincidencias, si no está solo la primera
- **s**: permite que el punto (.) coincida con el carácter de nueva línea (\n)

Hay tres funciones con las que utilizaremos expresiones regulares:

- `str.match()`
- `str.replace()`
- `regexp.text()`

`str.match(regex)`

Busca coincidencias de la expresión regular en la cadena y las devuelve en un **array**.

Se pueden dar tres casos:

- Si utilizamos el *flag* **g** devuelve todas las coincidencias
- Si no tenemos dicho *flag* devuelve un array cuyo primer elemento es la primera ocurrencia de la expresión regular, además de algunas propiedades como la cadena original.
- Si no hay ninguna coincidencia devuelve ***null***.

```
let str = "We will, we will rock you";  
console.log( str.match(/we/i) );
```

```
▼ Array [ "We" ]                                javascript.js:3:9  
  0: "We"  
  groups: undefined  
  index: 0  
  input: "We will, we will rock you"  
  length: 1  
  ▶ <prototype>: Array []
```

```
let str = "We will, we will rock you";  
console.log( str.match(/we/gi) );
```

```
▼ Array [ "We", "we" ]                          javascript.js:3:9  
  0: "We"  
  1: "we"  
  length: 2  
  ▶ <prototype>: Array []
```


`string.replace(regex, replacement)`

Busca las coincidencias de la expresión regular y las reemplaza por la cadena *replacement* (solo la primera coincidencia si no se indica el *flag g*).

Si queremos hacer referencia al patrón que se ajustó a la expresión regular se pueden utilizar los caracteres especiales **\$&**.

```
let str="Me gusta HTML";
console.log( str.replace(/HTML/, "JavaScript") );
// Me gusta JavaScript

console.log( str.replace(/HTML/, "$& y JavaScript") );
// Me gusta HTML y JavaScript
```

`regex.test(str)`

Analiza la cadena *str* en busca de una coincidencia del patrón, si encuentra alguna devuelve *true* y *false* si no encontrara ninguna.

```
>> /victor/.test('Victor J. González')
```

```
← false
```

```
>> /victor/i.test('Victor J. González')
```

```
← true
```

El **patrón** más sencillo que puede tener una expresión regular es una **cadena literal**, es decir, indicamos exactamente qué caracteres debe contener la expresión regular.

```
>> '2º DAW'.match(/DAW/)
< ▶ Array [ "DAW" ]
```

Pero hay muchas más probabilidades a la hora de crear una expresión regular.

Clases de caracteres

Las expresiones regulares que hemos visto hasta ahora buscan coincidencias exactas, pero podemos utilizar **clases de caracteres** para indicar que un carácter sea de un tipo determinado.

Las más utilizadas son:

- **\d**: un dígito
- **\s**: un carácter en blanco (espacio, tab, nueva línea, ...)
- **\w**: un carácter de palabra (letra del alfabeto latino, dígito o guion bajo)

Las clases de caracteres se pueden combinar con caracteres normales.

```
'25 de julio de 2020'.match(/\d\d\d\d/)
```

```
► Array [ "2020" ]
```

```
'25 de julio de 2020'.match(/ \w\w /g)
```

```
► Array [ " de ", " de " ]
```

Tambín disponemos de clases inversas:

- **\D**: cualquier caŕcter excepto un d́gito
- **\S**: cualquier caŕcter salvo espacios
- **\W**: cualquier caŕcter que no sea de palabra

Si queremos un patrón al que se ajuste **cualquier carácter** debemos utilizar el punto (.)

El carácter punto en una expresión regular significa un único carácter, sea cual sea.

```
'Día 25 de julio de 2020'.match(/ .. /g)
```

```
► Array [ " 25 ", " de " ]
```

Anclas: inicio ^ y final \$

El patrón ^ significa principio del texto, mientras que \$ significa final del texto.

```
let str="Me gusta HTML";  
console.log(str.match(/HTML/g));    // [ 'HTML' ]  
console.log(str.match(/HTML$/g));   // [ 'HTML' ]  
console.log(str.match(/^HTML/g));    // null
```

Si queremos una coincidencia exacta del patrón podemos usar ambos para delimitar la expresión regular.

```
let str="Me gusta HTML";  
console.log(str.match(/^HTML$/g));  // null  
console.log('HTML'.match(/^HTML$/g)); // ["HTML"]
```


Conjuntos y rangos [...]

Los corchetes significan un carácter de entre los que estén contenidos entre los corchetes. A esto se le llama **conjunto**.

```
console.log("DAW DAM".match( /DA[MW]/g )); // ["DAW", "DAM"]
```

También se pueden utilizar **rangos**, que se indican con un guion entre dos caracteres. Eso significa cualquier carácter comprendido entre los dos indicados.

```
matricula = /[0-9][0-9][0-9][0-9][A-Z][A-Z][A-Z]/g;
```

Excluyendo rangos

Si queremos utilizar conjuntos o rangos, pero necesitamos indicar *cualquier carácter menos uno de los indicados* debemos utilizar el carácter ^

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // @ y .
```

Cuantificadores +, *, ? y {n}

Los **cuantificadores** me permiten indicar que un patrón se repetirá varias veces.

Si queremos indicar que un patrón se repetirá un número fijo de veces utilizamos las llaves con el número de veces que se repetirá en su interior.

Por ejemplo, la expresión regular para identificar una matrícula que vimos antes quedaría de la forma:

```
matricula = /[0-9]{4}[A-Z]{3}/g;
```

Entre las llaves podemos indicar un número mínimo y un número máximo de veces que se repetirá, separando ambos valores con una coma.

Podemos omitir el valor de máximo para indicar que por lo menos se repita X veces.

```
console.log('1234ABC'.match( /[0-9]{2,}/g )); // ['1234']
```

Hay varios caracteres especiales para los cuantificadores ḿs comunes:

- `+`: significa uno o ḿs. Equivale a $\{1, \}$
- `?`: significa cero o uno. Equivale a $\{0, 1\}$
- `*`: significa cero o ḿs. Equivale a $\{0, \}$

Es importante entender bien cómo gestiona el motor de expresiones regulares los cuantificadores.

Observa el siguiente ejemplo donde se pretenden obtener todas las palabras rodeadas por comillas simples:

```
`Aquí se imparte 'DAW' y 'DAM'`.match(/'.'+/g)  
▶ Array [ "'DAW' y 'DAM'" ]
```

Como puedes apreciar, no devuelve dos cadenas como podríamos esperar, sino que devuelve una única cadena.

Esto se debe a que el motor de expresiones regulares utiliza **búsqueda codiciosa**, que básicamente consiste en que buscará la cadena más larga que se ajuste a la expresión regular.

Normalmente la búsqueda codiciosa (que es la búsqueda por defecto), pero hay ocasiones (como en el ejemplo anterior) en que no es lo más adecuado.

En esos casos podemos usar la **búsqueda perezosa**, que, resumiendo, consiste en que los cuantificadores se ajustarán a la cadena más corta encontrada.

Se indica que debe utilizar la **búsqueda perezosa** en un cuantificador **utilizando el símbolo ? detrás del cuantificador**.

```
`Aquí se imparte 'DAW' y 'DAM'`.match(/'.+?'/g)
```

```
► Array [ "'DAW'", "'DAM'" ]
```

Una parte de un patrón se puede rodear entre paréntesis, a esto se le llama **grupo de captura** y tiene dos utilidades:

- Si se coloca un cuantificador después de los paréntesis se aplicará a todo su contenido.
- Permite obtener una parte de la coincidencia como un elemento separado en la matriz de resultados

Cuantificadores tras paréntesis

```
"aaaabbabbba".match(/ab+/g)
```

```
▶ Array [ "abb", "abbb" ]
```

```
"aaaabbabbba".match(/(ab)+/g)
```

```
▶ Array [ "ab", "ab" ]
```

Obtener partes de la expresi3n regular

Si no utilizamos el flag `g` en la expresi3n regular y hay paréntesis dentro de la misma, la funci3n `match()` devolverá un array donde:

- El primer elemento será la primera coincidencia con la expresi3n regular
- Los siguientes elementos del array coincidirán con cada una de las partes de la expresi3n regular rodeada entre paréntesis.

```
'usuario@mail.com'.match(/\w+@\w+\.\w+/)
```

```
► Array [ "usuario@mail.com" ]
```

```
'usuario@mail.com'.match(/(\w+)@(\w+)\.(\w+)/)
```

```
► Array(4) [ "usuario@mail.com", "usuario", "mail", "com" ]
```

También se pueden **anidar los paréntesis**

```
'usuario@mail.com'.match(/(\w+)@((\w+)\.(\w+)))/)
```

```
► Array(5) [ "usuario@mail.com", "usuario", "mail.com", "mail", "com" ]
```

Si tenemos varios grupos puede ser difícil recordar el número de cada uno, pero podemos utilizar **grupos con nombre**.

Se hace poniendo la cadena `?<name>` justo después del paréntesis de apertura.

Podemos acceder ahora a estos grupos a través del objeto devuelto en la propiedad *groups*.

```
'victor@mail.com'.match(/(?<usuario>\w+)@(?<dominio>\w+\.\w+)/).groups  
► Object { usuario: "victor", dominio: "mail.com" }
```

Mismo ejemplo pero utilizando la desestructuración de objetos.

```
let {usuario, dominio} = 'victor@mail.com'.match(/(?<usuario>\w+)@(?<dominio>\w+\.\w+)/).groups
```

```
undefined
```

```
usuario
```

```
"victor"
```

```
dominio
```

```
"mail.com"
```

Otra posibilidad de los grupos es la utilización de **grupos de captura en reemplazo** con la función `replace()`.

En la cadena de reemplazo podemos hacer referencia a algún grupo utilizando a notación `$N` donde `N` es el número de grupo.

```
let str = "John Bull";  
let regexp = /(\w+) (\w+)/;  
  
alert( str.replace(regexp, '$2, $1') ); // Bull, John
```


Otra utilidad de los grupos es para hacer una **referencia inversa** en la expresión regular, es decir, para hacer referencia en la propia expresión regular a una coincidencia anterior.

Ejemplo, quiero localizar todas las palabras rodeadas de comillas, ya sean dobles o simples.

Podríamos usar la expresión `/[' "] .* [' "] /g` pero puede no dar los resultados esperados.

```
let a = `He said: "She's the one!".`.match(/['"](.*)['"]/g)
```

```
undefined
```

```
a
```

```
► Array [ "\"She'" ]
```

La solución a este problema está en el uso de referencias inversas:

```
`He said: "She's the one!".`.match(/(['"])(.*?)\1/g)  
► Array [ "\"She's the one!\\"" ]
```

También se podría utilizar **referencia inversa por nombre**.

```
let str = `He said: "She's the one!".`;
let regexp = /(?<quote>['"])(.*?)\k<quote>/g;
alert( str.match(regexp) ); // "She's the one!"
```

La **alternancia** se representa con el símbolo barra (|) y sirve para elegir entre dos partes de la expresión regular.

```
let regexp = /html|php|css|java(script)?/gi;  
  
let str = "Primera aparición de HTML, luego CSS, luego JavaScript";  
  
alert( str.match(regexp) ); // 'HTML', 'CSS', 'JavaScript'
```