

# Bloque 2

## Programación con JavaScript



**BOM y DOM. Eventos.  
Fechas. Expresiones  
regulares.**

Programación con JavaScript  
Cefire 2017/2018

Autor: Arturo Bernal Mayordomo

# Índice

Trabajando con el BOM y el DOM.....	3
Browser Object Model (BOM).....	3
Document Object Model (DOM).....	6
Eventos.....	12
Manejo de Eventos.....	13
Objeto del evento.....	15
Propagación de eventos (bubbling).....	16
Objetos y funciones globales.....	18
Funciones globales.....	18
El objeto Math.....	18
Fechas.....	20
Expresiones regulares.....	21

# Trabajando con el BOM y el DOM

---

Normalmente, cuando programamos en JavaScript, nos encontramos dentro del contexto de un navegador (esto no pasa si desarrollamos con NodeJS por ejemplo). Dentro de este contexto, tenemos algunos objetos predefinidos y funciones que podemos usar para interactuar con dicho navegador y con el documento HTML.

## Browser Object Model (BOM)

### Objeto window

El objeto **window** representa el navegador y es el objeto principal. Todo está contenido dentro de window (variables globales, el objeto document, el objeto location, el objeto navigation, etc.). El objeto window puede ser omitido accediendo a las propiedades directamente.

Comprueba toda la información de estos objetos, métodos y propiedades aquí: [window object](#), [location object](#), [document object](#), [history object](#), [navigator object](#), [screen object](#).

Ejemplos:

```
'use strict';
// Tamaño total de la ventana (excluye la barra superior del navegador)
console.log(window.outerWidth + " - " + window.outerHeight);
window.open("https://www.google.com");

// Propiedades de la pantalla
console.log(window.screen.width + " - " + window.screen.height); // Ancho de pantalla y alto (Resolución)
console.log(window.screen.availWidth + " - " + window.screen.availHeight); // Excluyendo la barra del S.O.

// Propiedades del navegador
console.log(window.navigator.userAgent); // Imprime la información del navegador
window.navigator.geolocation.getCurrentPosition(function(position) {
  console.log("Latitude: " + position.coords.latitude + ", Longitude: " + position.coords.longitude);
});

// En las variables globales podemos omitir el objeto window (está implícito)
console.log(history.length); // Número de páginas del history. Lo mismo que window.history.length
```

### “Timers” (avisadores)

Hay dos tipos de “timers” que podemos crear en JavaScript para ejecutar algún trozo de código en el futuro (especificado en milisegundos), **timeouts** e **intervals**. El primero se ejecuta sólo una vez (debemos volver a crearlo manualmente si queremos que se repita algo en el tiempo), y el segundo se repite cada X milisegundos sin parar (o hasta que sea cancelado).

- **timeout(función, milisegundos)** → Ejecuta una función pasados un número de milisegundos.

```
console.log(new Date().toString()); // Imprime inmediatamente la fecha actual
setTimeout(() => console.log(new Date().toString()), 5000); // Se ejecutará en 5 segundos (5000 ms)
```

- **clearTimeout(timeoutId)** → Cancela un timeout (antes de ser llamado)

```
// setTimeout devuelve un número con el id, y a partir de ahí, podremos cancelarlo
let idTime = setTimeout(() => console.log(new Date().toString()), 5000);
clearTimeout(idTime); // Cancela el timeout (antes de que se ejecute)
```

- **setInterval(funcion, milisegundo)** → La diferencia con timeout es que cuando el tiempo acaba y se ejecuta la función, se resetea y se repite cada X milisegundos automáticamente hasta que nosotros lo cancelemos.

```
let num = 1;
setInterval(() => console.log(num++), 1000); // Imprime un número y lo incrementa cada segundo
```

- **clearInterval(idInterval)** → Cancela un intervalo (no se repetirá más).

```
let num = 1;
let idInterval = setInterval(() => {
  console.log(num++);
  if(num > 10) { // Cuando imprimimos 10, paramos el timer para que no se repita más
    clearInterval(idInterval);
  }
}, 1000);
```

- **setInterval/setTimeout(nombreFuncion, milisegundos, argumentos...)** → Podemos pasarle un nombre función existente. Si se requieren parámetros podemos establecerlos tras los milisegundos.

```
function multiply(num1, num2) {
  console.log(num1 * num2);
}

setTimeout(multiply, 3000, 5, 7); // Después de 3 segundos imprimirá 35 (5*7)
```

## Objeto location

El objeto **location** (no confundir con el objeto navigator.geolocation) contiene información sobre la url actual del navegador. Podemos acceder y modificar dicha url usando este objeto.

```
console.log(location.href); // Imprime la URL actual
console.log(location.host); // Imprime el nombre del servidor (o la IP) como "localhost" 192.168.0.34
console.log(location.port); // Imprime el número del puerto (normalmente 80)
console.log(location.protocol); // Imprime el protocolo usado (http ó https)

location.reload(); // Recarga la página actual
location.assign("https://google.com"); // Carga una nueva página. El parámetro es la nueva URL
location.replace("https://google.com"); // Carga una nueva página sin guardar la actual en el objeto history
```

Para navegar a través de las páginas que hemos visitado en la pestaña actual, podemos usar el objeto **history**. Este objeto tiene métodos bastante útiles:

```
console.log(history.length); // Imprime el número de páginas almacenadas

history.back(); // Vuelve a la página anterior
history.forward(); // Va hacia la siguiente página
history.go(2); // Va dos páginas adelante (-2 iría dos páginas hacia atrás)
```

¿Qué ocurre si no queremos recargar la página actual cuando vamos atrás y adelante en el history?. Por ejemplo, nuestra página es controlada por métodos de JavaScript y queremos deshacer o rehacer cosas cuando un usuario pulsa el botón de atrás o avanzar.

Para conseguir esto, debemos usar **history.pushState()**. Estos métodos usan objetos JSON como primer parámetro con información sobre los cambios, y el título (normalmente es ignorado) como segundo parámetro. Podemos usar **replaceState()** si no queremos que se almacene en el historial.

En nuestra página web la propiedad **window.onpopstate** deberá ser asignada a una función. Esta función será llamada cada vez que avancemos de página o retrocedamos en el objeto history. Recibirá un parámetro que representa el evento de navegación, este tiene una propiedad llamada **state** que contiene los datos JSON que se asignaron al estado actual. Podemos acceder a ese estado a través del **history.state** (La primera página tendrá como estado null).

File: `example1.html`

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <script src="example1.js"></script>
    <button onclick="goBack()">Página anterior</button>
    <button onclick="goNext()">Página siguiente</button>
  </body>
</html>
```

File: `example1.js`

```
'use strict';

// Evento para capturar la navegación por el history
window.onpopstate = function (event) {
  if(event.state) {
    console.log("Estoy en la página " + event.state.page);
  } else { // event.state == null si es la primera página
    console.log("Estoy en la primera página");
  }
};

let page = 1;

function goBack() {
  history.back();
}

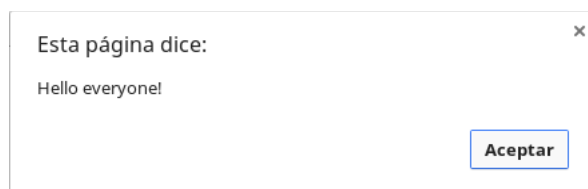
function goNext() {
  // history.state == null si es la primera página (la siguiente es la página 2)
  let pageNum = history.state?history.state.page + 1:2; // Siguiente página = Página actual + 1.
  history.pushState({page: pageNum}, "");
  console.log("Estoy en la página " + pageNum);
}
```

## Diálogos

En cada navegador, tenemos un conjunto de diálogos para interactuar con el usuario. Sin embargo, estos no son personalizables y por tanto cada navegador implementa el suyo a su manera. Por ello no es recomendable usarlos en una aplicación en producción (uniformidad). En cambio, son una buena opción para hacer pruebas (en producción deberíamos usar diálogos contruidos con HTML y CSS).

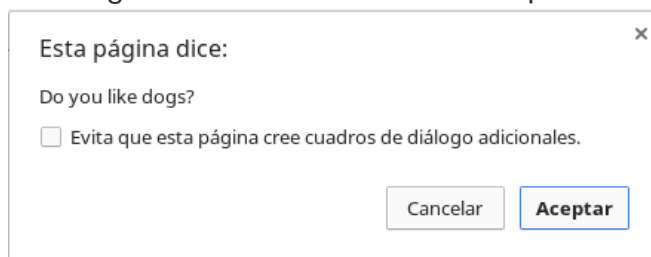
El diálogo **alert**, muestra un mensaje (con un botón de Aceptar) dentro de una ventana. Bloquea la ejecución de la aplicación hasta que se cierra.

```
alert("Hello everyone!");
```



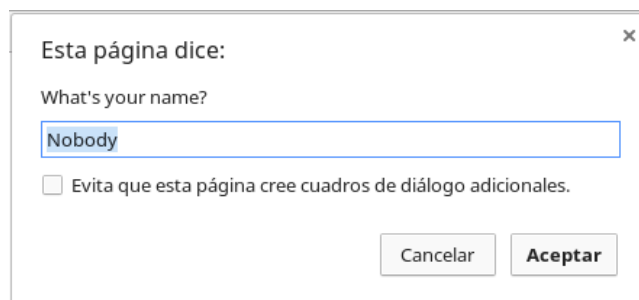
El diálogo **confirm** es similar, pero te devuelve un booleano. Tiene dos botones (cancelar → false, Aceptar → true). El usuario elegirá entre una de esas dos opciones.

```
if(confirm("Do you like dogs?")) {  
  console.log("You are a good person");  
} else {  
  console.log("You have no soul");  
}
```



El diálogo **prompt** muestra un input después del mensaje. Lo podemos usar para que el usuario introduzca algún valor, devolviendo un **string** con el valor introducido. Si el usuario pulsa el botón de **Cancelar** o **cierra** el diálogo devolverá **null**. Se puede establecer un valor por defecto (segundo parámetro).

```
let name = prompt("What's your name?", "Nobody");  
  
if(name !== "null") {  
  console.log("Your name is: " + name);  
} else {  
  console.log("You didn't answer!");  
}
```



## Document Object Model (DOM)

El Document Object Model es una estructura en árbol que contiene una representación de todos los nodos del HTML incluyendo sus atributos. En este árbol, todo se representa como nodo y podemos añadir, eliminar o modificarlos.

```

<html>
  ▶ <head>...</head>
  ▼ <body>
    ▼ <div>
      <p>Hello</p>
      ▼ <p>
        "Go to "
        <a href="https://google.es">Google</a>
      </p>
    </div>
    <script src="example1.js"></script>
  </body>
</html>

```

El objeto principal del DOM es [document](#). Este es un objeto global del lenguaje. Cada nodo HTML contenido dentro del documento es un objeto [element](#), y estos elementos contienen otros nodos, [atributos](#) y [estilo](#).

Manipular el DOM usando sólo JavaScript es algo más complicado que con la ayuda de librerías como JQuery o frameworks como Angular. Veremos algunos métodos básicos y propiedades de los elementos del DOM en JavaScript. No vamos a estudiarlo en profundidad para abarcar más conceptos durante el curso, y porque realmente usando jQuery, Angular, etc. es más fácil y práctico.

### Navegando a través del DOM

- **document.documentElement** → Devuelve el elemento <html>
- **document.head** → Devuelve el elemento <head>
- **document.body** → Devuelve el elemento <body>
- **document.getElementById("id")** → Devuelve el elemento que tiene el id especificado, o **null** si no existe.
- **document.getElementsByTagName("class")** → Devuelve un array de elementos que tengan la clase especificada. Al llamar a este método desde un nodo (en lugar de document), buscará los elementos a partir de dicho nodo.
- **document.getElementsByTagName("HTML tag")** → Devuelve un array con los elementos con la etiqueta HTML especificada. Por ejemplo "p" (párrafos).
- **element.childNodes** → Devuelve un array con los descendientes (hijos) del nodo. Esto incluye los nodos de tipo texto y comentarios.
- **element.chidren** → Igual que arriba pero excluye los comentarios y las etiquetas de texto (sólo nodos HTML). Normalmente es el recomendado.
- **element.parentNode** → Devuelve el nodo padre de un elemento.
- **element.nextSibling** → Devuelve el siguiente nodo del mismo nivel (el

hermano). El método **previousSibling** hace justo lo opuesto. Es recomendable usar **nextElementSibling** o **previousElementSibling** si queremos obtener sólo los elementos HTML.

File: `example1.html`

```
<!DOCTYPE>

<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <ul>
      <li id="firstListElement">Element 1</li>
      <li>Element 2</li>
      <li>Element 3</li>
    </ul>
    <script src="/example1.js"></script>
  </body>
</html>
```

File: `example1.js`

```
let firstLi = document.getElementById("firstListElement"); // Devuelve <li>

console.log(firstLi.nodeName); // Imprime "LI"
console.log(firstLi.nodeType); // Imprime 1. (elemento -> 1, atributo -> 2, texto -> 3, comentario -> 8)
console.log(firstLi.firstChild.nodeValue); // Imprime "Element 1". El primer (y único) hijo es un nodo de texto
console.log(firstLi.textContent); // Imprime "Element 1". Otra forma de obtener el contenido (texto)

// Itera a través de todos los elementos de la lista
let liElem = firstLi;
while(liElem !== null) {
  console.log(liElem.innerText); // Imprime el texto de dentro del elemento <li>
  liElem = liElem.nextElementSibling; // Va al siguiente elemento de la lista <li>
}

let ulElem = firstLi.parentElement; // Obtiene el elemento <ul>. Similar a parentNode.
/* Imprime el código HTML de dentro del elemento <ul>:
  <li id="firstListElement">Element 1</li>
  <li>Element 2</li>
  <li>Element 3</li> */
console.log(ulElem.innerHTML);
```

## Selector Query

Una de las principales características que JQuery introdujo cuando se lanzó (en 2006) fue la posibilidad de acceder a los elementos HTML basándose en selectores CSS (clase, id, atributos,...). Desde hace años, los navegadores han implementado esta característica de forma nativa (selector query) sin la necesidad de usar jquery.

- **document.querySelector("selector")** → Devuelve el primer elemento que coincide con el selector
- **document.querySelectorAll("selector")** → Devuelve un array con todos los elementos que coinciden con el selector



## Ejemplos de selectores CSS que podemos usar para encontrar elementos:

a → Elementos con la etiqueta HTML <a>  
.class → Elementos con la clase "class"  
#id → Elementos con el id "id"  
.class1.class2 → Elementos que tienen ambas clases, "class1" y "class2"  
.class1,.class2 → Elementos que contienen o la clase "class1", o "class2"  
.class1 p → Elementos <p> dentro de elementos con la clase "class1"  
.class1 > p → Elementos <p> que son hijos inmediatos con la clase "class1"  
#id + p → Elemento <p> que va después (siguiente hermano) de un elemento que tiene el id "id"  
#id ~ p → Elementos que son párrafos <p> y hermanos de un elemento con el id "id"  
.class[attrib] → Elementos con la clase "class" y un atributo llamado "attrib"  
.class[attrib="value"] → Elementos con la clase "class" y un atributo "attrib" con el valor "value"  
.class[attrib^="value"] → Elementos con la clase "class" y cuyo atributo "attrib" comienza con "value"  
.class[attrib\*="value"] → Elementos con la clase "class" cuyo atributo "attrib" en su valor contiene "value"  
.class[attrib\$="value"] → Elementos con la clase "class" y cuyo atributo "attrib" acaba con "value"

## Ejemplo usando querySelector() y querySelectorAll():

File: example1.html

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <div id="div1">
      <p>
        <a class="normalLink" href="hello.html" title="hello world">Hello World</a>
        <a class="normalLink" href="bye.html" title="bye world">Bye World</a>
        <a class="specialLink" href="helloagain.html" title="hello again">Hello Again World</a>
      </p>
    </div>
    <script src="./example1.js"></script>
  </body>
</html>
```

File: example1.js

```
console.log(document.querySelector("#div1 a").title); // Imprime "hello world"
console.log(document.querySelector("#div1 > a").title); // ERROR: No hay un hijo inmediato dentro de <div id="div1"> el cual sea un enlace <a>
console.log(document.querySelector("#div1 > p > a").title); // Imprime "hello world"
console.log(document.querySelector(".normalLink[title^='bye']").title); // Imprime "bye world"
console.log(document.querySelector(".normalLink[title^='bye'] + a").title); // Imprime "hello again"

let elems = document.querySelectorAll(".normalLink");
elems.forEach(function(elem) { // Imprime "hello world" y "bye world"
  console.log(elem.title);
});

let elems2 = document.querySelectorAll("a[title^='hello']"); // Atributo title empieza por "hello..."
elems2.forEach(function(elem) { // Imprime "hello world" y "hello again"
  console.log(elem.title);
});

let elems2 = document.querySelectorAll("a[title='hello world'] ~ a"); // Hermanos de <a title="hello world">
elems2.forEach(function(elem) { // Imprime "bye world" y "hello again"
  console.log(elem.title);
});
```

## Manipulando el DOM

- **document.createElement("tag")** → Crea un elemento HTML. Todavía no estará en el DOM, hasta que lo insertemos (usando **appendChild**, por ejemplo) dentro de otro elemento del DOM.
- **document.createTextNode("text")** → Crea un nodo de texto que podemos introducir dentro de un elemento. Equivale a **element.innerText = "texto"**.
- **element.appendChild(childElement)** → Añade un **nuevo** elemento hijo al final del elemento padre.
- **element.insertBefore(newChildElement, childElem)** → Inserta un nuevo elemento hijo **antes** del elemento hijo recibido como segundo parámetro.
- **element.removeChild(childElement)** → Elimina el nodo hijo que recibe por parámetro.
- **element.replaceChild(newChildElem, oldChildElem)** → Reemplaza un nodo hijo con un nuevo nodo.

Ejemplo de manipulación del DOM (mismo HTML que antes, con una lista):

```
let ul = document.getElementsByTagName("ul")[0]; // Obtiene la primera lista (ul)
let li3 = ul.children[2]; // Tercer elemento de la lista (li)
let newLi3 = document.createElement("li"); // Crea un nuevo elemento de lista
newLi3.innerText = "Now I'm the third element"; // Y le asigna un texto

ul.insertBefore(newLi3, li3); // Ahora li3 es el cuarto elemento de la lista (newLi3 se inserta antes)
li3.innerText = "I'm the fourth element..."; // Cambiamos el texto para reflejar que es el cuarto elemento
```

El HTML resultante será:

```
<ul>
  <li id="firstListElement">Element 1</li>
  <li>Element 2</li>
  <li>Now I'm the third element</li>
  <li>I'm the fourth element...</li>
</ul>
```

## Atributos

Dentro de los elementos HTML hay atributos como name, id, href, src, etc. Cada atributo tiene nombre (**name**) y valor (**value**), y este puede ser leído o modificado.

- **element.attributes** → Devuelve el array con los atributos de un elemento
- **element.className** → Se usa para acceder (leer o cambiar) al atributo **class**. Otros atributos a los que se puede acceder directamente son: **element.id**, **element.title**, **element.style** (propiedades CSS), ... .
- **element.hasAttribute("attrName")** → Devuelve cierto si el elemento tiene un atributo con el nombre especificado

- `element.getAttribute("attrName")` → Devuelve el valor del atributo
- `element.setAttribute("attrName", "newValue")` → Cambia el valor

File: `example1.html`

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <p><a id="toGoogle" href="https://google.es" class="normalLink">Google</a></p>
    <script src="/example1.js"></script>
  </body>
</html>
```

File: `example1.js`

```
let link = document.getElementById("toGoogle");
link.className = "specialLink"; // Equivale a: link.setAttribute("class", "specialLink");
link.setAttribute("href", "https://twitter.com");
link.textContent = "Twitter";
if(!link.hasAttribute("title")) { // Si no tenía el atributo title, establecemos uno
  link.title = "Ahora voy a Twitter!";
}

/* Imprime: <a id="toGoogle" href="https://twitter.com" class="specialLink"
  title="Ahora voy a Twitter!">Twitter</a> */
console.log(link);
```

## El atributo style

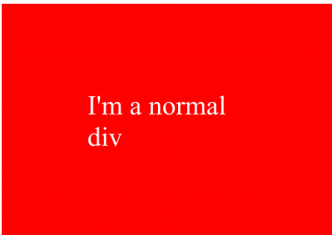
El atributo `style` permite modificar las propiedades CSS. La propiedad CSS a modificar deben escribirse con el formato **camelCase**, mientras que en CSS se emplea en el formato **snake-case**. Por ejemplo, al atributo **background-color** (CSS), se accede a partir de **element.style.backgroundColor**. El valor establecido a una propiedad será un string que contendrá un valor CSS válido para el atributo.

File: `example1.html`

```
<!DOCTYPE>
<html>
  <head>
    <title>JS Example</title>
  </head>
  <body>
    <div id="normalDiv">I'm a normal div</div>
    <script src="/example1.js"></script>
  </body>
</html>
```

File: `example1.js`

```
let div = document.getElementById("normalDiv");
div.style.boxSizing = "border-box";
div.style.maxWidth = "200px";
div.style.padding = "50px";
div.style.color = "white";
div.style.backgroundColor = "red";
```



I'm a normal  
div

# Eventos

---

Cuando un usuario interactúa con una aplicación, se producen una serie de eventos (de teclado, ratón, etc.) que nuestro código debería manejar de forma adecuada. Hay muchos eventos que pueden ser capturados y procesados (listado [aquí](#)). Veremos algunos de los más comunes.

## Eventos de la página

Estos eventos son producidos en el documento HTML. Normalmente afectan al elemento **body**.

- **load** → Este evento se lanza cuando el documento HTML ha terminado de cargarse. Es útil para realizar acciones que requieran que el DOM haya sido completamente cargado (como consultar o modificar el DOM).
- **unload** → Ocurre cuando el documento es destruido, por ejemplo, después de cerrar la pestaña donde la página estaba cargada.
- **beforeunload** → Ocurre justo antes de cerrar la página. Por defecto, un mensaje pregunta al usuario si quiere realmente salir de la página, pero hay otras acciones que pueden ser ejecutadas.
- **resize** → Este evento se lanza cuando el tamaño del documento cambia (normalmente se usa si la ventana se redimensiona)

## Eventos del teclado

- **keydown** → El usuario presiona una tecla. Si la tecla se mantiene pulsada durante un tiempo, este evento se generará de forma repetida.
- **keyup** → Se lanza cuando el usuario deja de presionar la tecla
- **keypress** → Más o menos lo mismo que **keydown**. Acción de pulsar y levantar.

## Eventos del ratón

- **click** → Este evento ocurre cuando el usuario pulsa un elemento (presiona y levanta el dedo del botón → **mousedown** + **mouseup**). También normalmente se lanza cuando un evento táctil de toque (tap) es recibido.
- **dblclick** → Se lanza cuando se hace un doble click sobre el elemento
- **mousedown** → Este evento ocurre cuando el usuario presiona un botón del ratón
- **mouseup** → Este evento ocurre cuando el usuario levanta el dedo del botón del ratón

- **mouseenter** → Se lanza cuando el puntero del ratón entra en un elemento
- **mouseleave** → Se lanza cuando el puntero del ratón sale de un elemento
- **mousemove** → Este evento se llama repetidamente cuando el puntero de un ratón se mueve mientras está dentro de un elemento

### Eventos Touch

- **touchstart** → Se lanza cuando se detecta un toque en la pantalla táctil
- **touchend** → Se lanza cuando se deja de pulsar la pantalla táctil
- **touchmove** → Se lanza cuando un dedo es desplazado a través de la pantalla
- **touchcancel** → Este evento ocurre cuando se interrumpe un evento táctil.

### Eventos de formulario

- **focus** → Este evento se ejecuta cuando un elemento (no sólo un elemento de un formulario) tiene el foco (es seleccionado o está activo).
- **blur** → Se ejecuta cuando un elemento pierde el foco.
- **change** → Se ejecuta cuando el contenido, selección o estado del checkbox de un elemento cambia (sólo <input>, <select>, y <textarea>)
- **input** → Este evento se produce cuando el valor de un elemento <input> o <textarea> cambia.
- **select** → Este evento se lanza cuando el usuario selecciona un texto de un <input> o <textarea>.
- **submit** → Se ejecuta cuando un formulario es enviado (el envío puede ser cancelado).

## Manejo de Eventos

Hay muchas formas de asignar un código o función a un determinado evento. Vamos a ver las dos formas posibles (para ayudar a entender código hecho por otros), pero el recomendado (y la forma válida para este curso) es usar **event listeners**.

### Manejo de eventos clásico (menos recomendado)

Lo primero de todo, podemos poner código JavaScript (o llamar a una función) en un atributo de un elemento HTML. Estos atributos se nombran como los eventos, pero con el prefijo 'on' (click → onclick). Vamos a ver un ejemplo del evento click.

```
<!DOCTYPE>
<html>
  <head>
```

```

<title>Ejemplo JS</title>
</head>
<body>
  <div id="div1">
    <p>
      <input type="text" onclick="alert('¡Me has pulsado!')" />
    </p>
  </div>
  <script src="/ejemplo1.js"></script>
</body>
</html>

```

Si llamáramos a una función, podemos pasarle parámetros (si es necesario. La palabra reservada **this** y **event** se pueden usar para pasar el elemento HTML (afectado por el evento) y / o el objeto con información del evento a la función.

Archivo: ejemplo1.html

```

<input type="text" id="input1" onclick="inputClick(this, event)" />

```

Archivo: ejemplo1.js

```

function inputClick(element, event) {
  // Mostrará "Un evento click ha sido detectado en #input1"
  alert("Un evento" + event.type + " ha sido detectado en #" + element.id);
}

```

Podemos añadir un manejador (función) de evento desde código, accediendo a la propiedad correspondiente (onclick, onfocus, ...), o asignarles un valor nulo si queremos dejar de escuchar algún evento.

```

let input = document.getElementById("input1");
input.onclick = function(event) {
  // Dentro de esta función, 'this' se refiere al elemento
  alert("Un evento" + event.type + " ha sido detectado en " + this.id);
}

```

## Event listeners (recomendado)

El método para manejar eventos explicado arriba tiene algunas desventajas, por ejemplo, no se pueden gestionar varias funciones manejadoras para un mismo evento.

Para añadir un *event listener*, usamos el método **addEventListener** sobre el elemento. Este método recibe al menos dos parámetros. El nombre del evento (una cadena) y un manejador (función anónima o nombre de una función existente).

```

let input = document.getElementById("input1");
input.addEventListener('click', function(event) {
  alert("Un evento" + event.type + " ha sido detectado en " + this.id);
});

```

Podemos añadir tantos manejadores como queramos. Sin embargo, si queremos eliminar un manejador, debemos indicar qué función estamos eliminando.

```

let inputClick = function(event) {
  console.log("Un evento" + event.type + " ha sido detectado en " + this.id);
};

```

```

let inputClick2 = function(event) {
  console.log("Yo soy otro manejador para el evento click!");
};

let input = document.getElementById("input1");
// Añadimos ambos manejadores. Al hacer clic, se ejecutarían ambos por orden.
input.addEventListener('click', inputClick);
input.addEventListener('click', inputClick2);

// Así es cómo se elimina el manejador de un evento
input.removeEventListener('click', inputClick);
input.removeEventListener('click', inputClick2);

```

## Objeto del evento

El objeto del evento es creado por JavaScript y pasado al manejador como parámetro. Este objeto tiene algunas propiedades generales (independientemente del tipo de evento) y otras propiedades específicas (por ejemplo, un evento del ratón tiene las coordenadas del puntero, etc.).

Estas son algunas propiedades generales que tienen todos los eventos:

- **target** → El elemento que lanza el evento (si fue pulsado, etc...).
- **type** → El nombre del evento: 'click', 'keypress', ...
- **cancelable** → Devuelve true o false. Si el evento se puede cancelar significa que llamando a **event.preventDefault()** se puede anular la acción por defecto (El envío de un formulario, el click de un link, etc...).
- **bubbles** → Devuelve cierto o falso dependiendo de si el evento se está propagando (Lo veremos más adelante).
- **preventDefault()** → Este método previene el comportamiento por defecto (cargar una página cuando se pulsa un enlace, el envío de un formulario, etc.)
- **stopPropagation()** → Previene la propagación del evento.
- **stopImmediatePropagation()** → Si el evento tiene más de un manejador, se llama a este método para prevenir la ejecución del resto de manejadores.

Dependiendo del tipo de evento, el objeto tendrá diferentes propiedades:

### MouseEvent

- **button** → Devuelve el botón del ratón que lo ha pulsado (0: botón izquierdo, 1: la rueda del ratón, 2: botón derecho).
- **clientX, clientY** → Coordenadas relativas del ratón en la ventana del navegador cuando el evento fue lanzado.
- **pageX, pageY** → Coordenadas relativas del documento HTML, si se ha realizado algún tipo de desplazamiento (scroll), este será añadido (usando

clientX y clientY no se añade).

- **screenX, screenY** → Coordenadas absolutas del ratón en la pantalla.
- **detail** → Indica cuántas veces el botón del ratón ha sido pulsado (un click, doble, o triple click).

### KeyboardEvent

- **key** → Devuelve el nombre de la tecla pulsada.
- **keyCode** → Devuelve el código del carácter [Unicode](#) en el evento **keypress**, **keyup** o **keydown**.
- **altKey, ctrlKey, shiftKey, metaKey** → Devuelven si las teclas “alt”, “control”, “shift” o “meta” han sido pulsadas durante el evento (Bastante útil para las combinaciones de teclas como ctrl+c). El objeto **MouseEvent** también tiene estas propiedades.

## Propagación de eventos (bubbling)

Muchas veces, hay elementos en una web que se solapan con otros elementos (están contenidos dentro). Por ejemplo, si pulsamos sobre un párrafo que está contenido en un elemento <div>, ¿el evento se ejecutará en el párrafo, en el <div> o en ambos? ¿Cuál se ejecuta primero?

Por ejemplo, vamos a ver qué ocurre con estos dos elementos (un elemento <div> dentro de otro <div>) cuando hacemos clic en ellos.

Archivo: ejemplo1.html

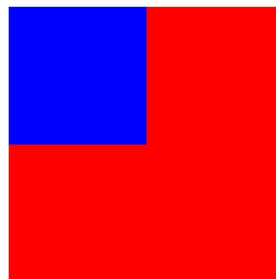
```
<div id="div1" style="background-color: red; width: 200px; height: 200px;">
  <div id="div2" style="background-color: blue; width: 100px; height: 100px;"></div>
</div>
```

Archivo: ejemplo1.js

```
let divClick = function(event) {
  console.log("Has pulsado: " + this.id);
};

let div1 = document.getElementById("div1");
let div2 = document.getElementById("div2");

div1.addEventListener('click', divClick);
div2.addEventListener('click', divClick);
```



Si hacemos click sobre el elemento rojo <div id="div1">, se imprimirá solo “Has pulsado: div1”. Sin embargo, si pulsamos sobre el elemento azul <div> (el cual está dentro del elemento rojo) imprimirá ambos mensajes (#div2 primero). En conclusión, por defecto el elemento el cual está al frente (normalmente un elemento hijo) recibe el evento **primero** y entonces pasa a ejecutar los manejadores que contiene.

Normalmente, la propagación de eventos va de padres a hijos, pero podemos



cambiar este proceso añadiendo un tercer parámetro al método `addEventListener` y establecerlo a **true**.

Vamos a ver otro ejemplo:

Archivo: `ejemplo1.html`

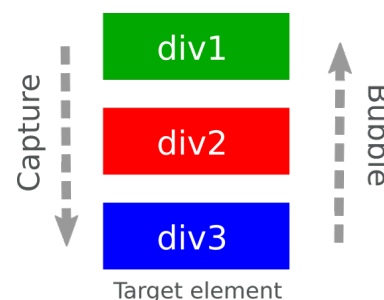
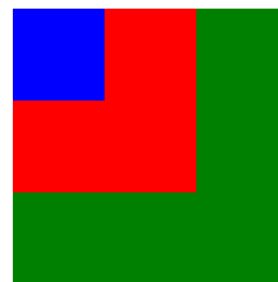
```
<div id="div1" style="background-color: green; width: 150px; height: 150px;">
  <div id="div2" style="background-color: red; width: 100px; height: 100px;">
    <div id="div3" style="background-color: blue; width: 50px; height: 50px;"></div>
  </div>
</div>
```

Archivo: `ejemplo1.js`

```
let divClick = function(event) {
  // eventPhase: 1 -> capture, 2 -> target (objetivo), 3 -> bubble
  console.log("Has pulsado: " + this.id + ". Fase: " + event.eventPhase);
};

let div1 = document.getElementById("div1");
let div2 = document.getElementById("div2");
let div3 = document.getElementById("div3");

div1.addEventListener('click', divClick);
div2.addEventListener('click', divClick);
div3.addEventListener('click', divClick);
```



Por defecto, cuando pulsamos el `<div>` azul (`div3`), imprime:

```
Has pulsado: div3. Fase: 2 → Target element
Has pulsado: div2. Fase: 3 → Bubbling
Has pulsado: div1. Fase: 3 → Bubbling
```

Si se establece un tercer parámetro a **true**, se imprimirá:

```
Has pulsado: div1. Fase: 1 → Propagation
Has pulsado: div2. Fase: 1 → Propagation
Has pulsado: div3. Fase: 2 → Target element
```

Podemos llamar al método **stopPropagation** en el evento, no continuará la propagación (si es en la fase de captura) o en (la fase de propagación o bubbling).

```
let divClick = function(event) {
  // eventPhase: 1 -> capture, 2 -> target (clicked), 3 -> bubble
  console.log("Has pulsado: " + this.id + ". Fase: " + event.eventPhase);
  event.stopPropagation();
};
```

Ahora, cuando hacemos click el elemento imprimirá solo un mensaje, "Has pulsado: div3. Fase: 2", si el tercer argumento no se ha establecido (o se ha puesto a false), o "Has pulsado: div1. Fase: 1" si el tercer argumento se ha marcado a true (el elemento padre previene a los hijos de recibirlo en este caso).

# Objetos y funciones globales

---

JavaScript tiene algunas funciones y objetos globales, las cuales pueden ser accedidas desde cualquier sitio. Estas funciones y objetos son bastante útiles para trabajar con números, cadenas, etc.

## Funciones globales

- **parseInt(value)** → Transforma cualquier valor en un entero. Devuelve el valor entero, o NaN si no puede ser convertido.
- **parseFloat(value)** → Igual que parseInt, pero devuelve un decimal.
- **isNaN(value)** → Devuelve true si el valor es NaN.
- **isFinite(value)** → Devuelve true si el valor es un número finito o false si es infinito.
- **Number(value)** → Transforma un valor en un número (o NaN).
- **String(value)** → Convierte un valor en un string (en objetos llama a toString()).
- **encodeURIComponent(string), decodeURI(string)** → Transforma una cadena en una URL codificada, codificando caracteres especiales a excepción de: , / ? : @ & = + \$ #. Usa decodeURI para transformarlo a string otra vez.
  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: "http://domain.com?val=1%20%203&val2=r+y%256"
- **encodeURIComponentComponent(string), decodeURIComponent(string)** → Estas funciones también codifican y decodifican los caracteres especiales que encodeURIComponent no hace. Se deben usar para codificar elementos de una url como valores de parámetros (no la url entera).
  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: "http%3A%2F%2Fdomain.com%3Fval%3D1%20%203%26val2%3Dr%2By%256"

## El objeto Math

El objeto Math nos proporciona algunas constantes o métodos bastante útiles.

- **Constants** → E (Número de Euler), PI, LN2 (algoritmo natural en base 2), LN10, LOG2E (base-2 logaritmo de E), LOG10E, SQRT1\_2 (raíz cuadrada de 1/2), SQRT2.
- **round(x)** → Redondea x al entero más cercano.

- **floor(x)** → Redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- **ceil(x)** → Redondea x hacia arriba (5.01 → 6)
- **min(x1,x2,...)** → Devuelve el número más bajo de los argumentos que se le pasan.
- **max(x1,x2,...)** → Devuelve el número más alto de los argumentos que se le pasan.
- **pow(x, y)** → Devuelve  $x^y$  (x elevado a y).
- **abs(x)** → Devuelve el valor absoluto de x.
- **random()** → Devuelve un número decimal aleatorio entre 0 y 1 (no incluidos).
- **cos(x)** → Devuelve el coseno de x (en radianes).
- **sin(x)** → Devuelve el seno de x.
- **tan(x)** → Devuelve la tangente de x.
- **sqrt(x)** → Devuelve la raíz cuadrada de x

```
console.log("Raíz cuadrada de 9: " + Math.sqrt(9));
console.log("El valor de PI es: " + Math.PI);
console.log(Math.round(4.546342));
// Número aleatorio entre 1 y 10
console.log(Math.floor(Math.random() * 10) + 1);
```

# Fechas

En Javascript tenemos la clase `Date`, que encapsula información sobre fechas y métodos para operar, permitiéndonos almacenar la fecha y hora local (timezone).

```
let date = new Date(); // Crea objeto Date almacena la fecha actual
console.log(typeof date); // Imprime object
console.log(date instanceof Date); // Imprime true
console.log(date); // Imprime (en el momento de ejecución) Fri Jun 24 2016 12:27:32 GMT+0200 (CEST)
```

Podemos enviarle al constructor el número de milisegundos desde el 1/1/1970 a las 00:00:00 GMT (Llamado **Epoch** o **UNIX time**). Si pasamos más de un número, (sólo el primero y el segundo son obligatorios), el orden debería ser: 1<sup>to</sup> → año, 2<sup>o</sup> → mes (0..11), 3<sup>o</sup> → día, 4<sup>o</sup> → hora, 5<sup>o</sup> → minuto, 6<sup>o</sup> → segundo. Otra opción es pasar un string que contenga la fecha en un formato válido.

```
let date = new Date(1363754739620); // Nueva fecha 20/03/2013 05:45:39 (milisegundos desde Epoch)
let date2 = new Date(2015, 5, 17, 12, 30, 50); // 17/06/2015 12:30:50 (Mes empieza en 0 -> Ene, ... 11 -> Dic)
let date3 = new Date("2015-03-25"); // Formato de fecha largo sin la hora YYYY-MM-DD (00:00:00)
let date4 = new Date("2015-03-25T12:00:00"); // Formato fecha largo con la fecha
let date5 = new Date("03/25/2015"); // Formato corto MM/DD/YYYY
let date6 = new Date("25 Mar 2015"); // Formato corto con el mes en texto (March también sería válido).
let date7 = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (CET)"); // Formato completo con el timezone
```

Si, en lugar de un objeto Date, queremos directamente obtener los milisegundos que han pasado desde el 1/1/1970 (Epoch), lo que tenemos que hacer es usar los métodos `Date.parse(string)` y `Date.UTC(año, mes, día, hora, minuto, segundos)`. También podemos usar `Date.now()`, para la fecha y hora actual en milisegundos.

```
let nowMs = Date.now(); // Momento actual en ms
let dateMs = Date.parse("25 Mar 2015"); // 25 Marzo 2015 en ms
let dateMs2 = Date.UTC(2015, 2, 25); // 25 Marzo 2015 en ms
```

La clase Date tiene **setters** y **getters** para las propiedades: **FullYear**, **month** (0-11), **date** (día), **hours**, **minutes**, **seconds**, y **milliseconds**. Si pasamos un valor negativo, por ejemplo, mes = -1, se establece el último mes (dic.) del año anterior.

```
// Crea un objeto fecha de hace 2 horas
let twoHoursAgo = new Date(Date.now() - (1000*60*60*2)); // (Ahora - 2 horas) en ms
// Ahora hacemos lo mismo, pero usando el método setHours
let now = new Date();
now.setHours(now.getHours() - 2);
```

Cuando queremos imprimir la fecha, tenemos métodos que nos la devuelven en diferentes formatos:

```
let now = new Date();

console.log(now.toString());
console.log(now.toISOString()); // Imprime 2016-06-26T18:00:31.246Z
console.log(now.toUTCString()); // Imprime Sun, 26 Jun 2016 18:02:48 GMT
console.log(now.toDateString()); // Imprime Sun Jun 26 2016
console.log(now.toLocaleDateString()); // Imprime 26/6/2016
console.log(now.toTimeString()); // Imprime 20:00:31 GMT+0200 (CEST)
console.log(now.toLocaleTimeString()); // Imprime 20:00:31
```

# Expresiones regulares

---

Las expresiones regulares nos permiten buscar patrones en un string, por tanto buscar el trozo que coincida o cumpla dicha expresión. En JavaScript, podemos crear una expresión regular instanciando un objeto de la clase `RegExp` o escribiendo directamente la expresión entre dos barras `'/'`.

En este apartado veremos conceptos básicos sobre las expresiones regulares, podemos aprender más sobre ellas en el siguiente [enlace](#). También, hay páginas web como [esta](#) que nos permiten probar expresiones regulares con cualquier texto.

Una expresión también puede tener uno o más modificadores como `'g'` → Búsqueda global (Busca todas las coincidencias y no sólo la primera), `'i'` → Case-insensitive (No distingue entre mayúsculas y minúsculas), o `'m'` → Búsqueda en más de una línea (Sólo tiene sentido en cadenas con saltos de línea). En Javascript puedes crear un objeto de expresión regular estos modificadores de dos formas:

```
let reg = new RegExp("[0-9]{2}", "gi");
let reg2 = /[0-9]{2}/gi;
console.log(reg2 instanceof RegExp); // Imprime true
```

Estas dos formas son equivalentes, por tanto podéis elegir cual usar.

## Expresiones regulares básicas

La forma más básica de una expresión regular es incluir sólo caracteres alfanuméricos (o cualquier otro que no sea un carácter especial). Esta expresión será buscada en el string, y devolverá `true` si encuentra ese *substring*.

```
let str = "Hello, I'm using regular expressions";
let reg = /reg/;
console.log(reg.test(str)); // Imprime true
```

En este caso, `"reg"` se encuentra en `"Hello, I'm using regular expressions"`, por lo tanto, el método `test` devuelve `true`.

## Corchetes (opción entre caracteres)

Entre corchetes podemos incluir varios caracteres (o un rango), y se comprobará si el carácter en esa posición de la cadena coincide con alguno de esos.

`[abc]` → Algún carácter que sea `'a'`, `'b'`, ó `'c'`

`[a-z]` → Algún carácter entre `'a'` y `'z'` (en minúsculas)

`[0-9]` → Algún carácter entre 0 y 9 (carácter numérico)

`[^ab0-9]` → Algún caracter **excepto** `(^)` `'a'`, `'b'` ó número.

## Pipe → | (opción entre subexpresiones)

(exp1|exp2) → La coincidencia en la cadena será con la primera expresión o con la segunda. Podemos añadir tantos pipes como queramos.

## Meta-caracteres

. (punto) → Un único carácter (cualquier carácter excepto salto de línea)

\w → Una palabra o carácter alfanumérico. \W → Lo opuesto a \w

\d → un número o dígito. \D → Lo opuesto a \d

\s → Carácter de espacio. \S → Lo opuesto a \s

\b → Delimitador de palabra. Coincide el principio o final de palabra (no un carácter). Por ejemplo, /\bcase\b/, coincide con “case” pero no “uppercase” o “casein”.

\n → Nueva línea

\t → Carácter de tabulación

## Cuantificadores

+ → El carácter que le precede (o la expresión dentro de un paréntesis) se repite 1 o más veces.

\* → Cero o más ocurrencias. Lo contrario a +, no tiene porqué aparecer

? → Cero o una ocurrencia. Se podría interpretar como que lo anterior es opcional.

{N} → Debe aparecer N veces seguidas.

{N,M} → De N a M veces seguidas.

{N,} → Al menos N veces seguidas.

^ → Principio de cadena.

\$ → Final de la cadena.

Podemos encontrar más sobre las expresiones regulares de JavaScript [aquí](#).

## Ejemplos

/^[0-9]{8}[a-z]\$/i → Comprueba si un string tiene un DNI. Esta expresión coincidirá con una cadena que comience (^) con 8 números, seguidos con una letra. No puede haber nada después (\$ → final de cadena). El modificador ‘i’ hace que no se distinga entre mayúsculas y minúsculas.

/^\d{2}\d{2}\d{2}\d{4}\$/ → Comprueba si una cadena contiene una fecha en

el formato **DD/MM/YY** ó **DD/MM/YYYY**. El metacaracter **\d** es equivalente a usar **[0-9]**, y el año no puede tener tres números, o son 2 o 4.

**/b[aeiou]\w\*\b/i** → Comprueba si hay alguna palabra en el string que comienza por una vocal seguida de cero o más caracteres alfanuméricos (números, letras o '\_').

## Métodos para las expresiones regulares en JavaScript

A partir de un objeto **RegExp**, podemos usar dos métodos para comprobar si una cadena cumple una expresión regular. Los dos métodos son **test** y **exec**.

El método **test()** recibe una cadena e intenta de encontrar una la coincidencia con la expresión regular. Si el modificador global '**g**' se especifica, cada vez que se llame al método se ejecutará desde la posición en la que se encontró la última coincidencia, por tanto podemos saber cuántas veces se encuentran coincidencias con esa expresión. Debemos recordar que si usamos el modificador '**i**' no diferencia entre mayúsculas y minúsculas.

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hay solo dos coincidencias
```

```
let reg2 = /am/gi; // "Am" será lo que busque ahora
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true. Ahora tenemos 3 coincidencias con este nuevo patrón
```

Si queremos más detalles sobre las coincidencias, podríamos usar el método **exec()**. Este método devuelve un objeto con los detalles cuando encuentra alguna coincidencia. Estos detalles incluyen el índice en el que empieza la coincidencia y también el string entero.

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null. No hay más coincidencias
```

Una mejor forma de usarlo sería:

```
let str = "I am amazed in America";
let reg = /am/gi;
let match;
while(match = reg.exec(str)) {
  console.log("Patrón encontrado!: " + match[0] + ", en la posición: " + match.index);
}
/* Esto imprimirá:
* Patrón encontrado!: am, en la posición: 2
* Patrón encontrado!: am, en la posición: 5
* Patrón encontrado!: Am, en la posición: 15 */
```

De forma similar, hay métodos de la clase **String** (sobre la cadena esta vez) que podemos usar, estos admiten expresiones regulares como parámetros (vamos, a la

inversa). Estos métodos son **match** (Funciona de forma similar a `exec`) y **replace**.

El método **match** devuelve un array con todas las coincidencias encontradas en la cadena si ponemos el modificador global → **g**, si no ponemos el modificador global, se comporta igual que **exec()**.

```
let str = "I am amazed in America";  
console.log(str.match(/am/gi)); // Imprime ["am", "am", "Am"]
```

El método **replace** devuelve una nueva cadena con las coincidencias de la expresión regular reemplazadas por la cadena que le pasamos como segundo parámetro (si el modificador global no se especifica, sólo se modifica la primera coincidencia). Podemos enviar una función anónima que procese cada coincidencia encontrada y nos devuelva la cadena con los reemplazos correspondientes.

```
let str = "I am amazed in America";  
console.log(str.replace(/am/gi, "xx")); // Imprime "I xx xazed in xxerica"  
  
console.log(str.replace(/am/gi, function(match) {  
  return "-" + match.toUpperCase() + "-";  
})); // Imprime "I -AM- -AM-azed in -AM-erica"
```