Browser Object Model

The Browser Object Model, or **BOM** , is a collection of global objects with various functionalities for interacting with the browser's capabilities from our JavaScript code. This includes things like managing the browser window, the current tab's page history, the current URL, local storage, geolocation, and a series of new APIs that have emerged, such as push notifications, multithreading with Webworkers, or access to the device's camera, among others.

The Window object

The window object represents the browser window and is the main object. The rest of the BOM objects are contained within window (location , document , history , navigator , screen). The window object can be overridden by accessing its properties directly (window.location == location).

 # index.js

```
'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);
});

//  Podemos omitir el objeto window (está implícito)
console.log(history.length); // Páginas visitadas en la pestaña
actual. Lo mismo que window.history.length
```

Interval and Timeout

There are two types of timers we can create in JavaScript to execute a piece of code in the future (specified in milliseconds): **timeout** and **interval** . The first one runs only once (we must recreate it manually if we want it to repeat more times), and the second one repeats every X milliseconds without stopping (or until canceled).

timeout(function, milliseconds) → Executes a function after a number of milliseconds.

 # index.js

```
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) → Cancels a timeout (before being called)

 # index.js

```
// 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(function, millisecond) → The difference with timeout is that when the time runs out and the function is executed, it is reset and repeated every X milliseconds automatically until we cancel it.

 # index.js

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

clearInterval(idIntervalo) → Cancels an interval (it will no longer be repeated).

 # index.js

```
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(functionName, milliseconds, arguments...) → We can pass an existing function name. Additionally, if parameters are required, we can set their values after the milliseconds.

 # index.js

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

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

The location object

The location object contains information about the browser's current URL. We can access and modify this URL from this object.

 # index.js

```
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)
console.log(location.search); // Imprime los parámetros de búsqueda
de la url (Ej: '?p1=1&p2=2')

location.reload(); // Recarga la página actual
location.assign("https://google.com"); // Carga una nueva página
```

```
location.replace("https://google.com"); // Carga una nueva página sin guardar la actual en el objeto history
```

The history object

To navigate through the pages we've visited in the current tab, we can use the **history** object . This object has some very useful methods.

 # index.js

```
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)
```

Dialog boxes

In each browser, we have a set of dialog boxes for interacting with the user. However, these are not customizable, and therefore each browser implements its own dialog boxes in its own way and with its own style. Therefore, it is not recommended to use them in a production application (uniformity). Instead, they are a good option for testing (in production, we should use dialog boxes built with HTML and CSS).

Alert

The alert dialog displays a message (with an OK button) within a window. It blocks the application from running until it is closed.

 # index.js

```
alert("Hola mundo!")
```

Confirm

The confirm dialog is similar, but returns a Boolean. It has two buttons (**Cancel → false** , **OK → true**). The user will choose between those two options.

 # index.js

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

Prompt

The prompt dialog displays an input after the message. We can use it to prompt the user to enter a value, returning a string containing the entered value. If the user clicks the Cancel button or closes the dialog, null will be returned. A default value can be set (second parameter).

 # index.js

```
let name = prompt("What's your name?", "Nobody");

if(name) {
    console.log("Your name is: " + name);
} else {
    console.log("You didn't answer!");
}
```

Document Object Model

The Document Object Model (**DOM**) is an internal tree structure that contains a representation of all HTML nodes, including their attributes (objects). We can navigate through this hierarchy, adding, removing, or modifying nodes and thus the structure of the HTML document dynamically.

The main object of the DOM is `document` , which represents an object that contains all the HTML structure. Each HTML node within the document is an Element object, and each of these elements in turn contains other nodes, attributes, and styling.

Sections of this section

Navigating the DOM

The following methods allow us to access different DOM nodes. Many of them, in addition to being used from **document** , can also be used from **any HTML element** we reference in our code.

- **document.documentElement** → Represents the <html> element
- **document.head** → Represents the element
- **document.body** → Represents the element
- **document.getElementById("id")** → Returns the element with the specified id, or null if it does not exist.
- **document.getElementsByClassName("class")** → Returns an array of elements that have the specified CSS class. Calling this method from a node other than document will search for elements starting from that node.
- **document.getElementsByTagName("HTML tag")** → Returns an array with the elements with the specified HTML tag (For example "p" → paragraphs), which are within the referenced node.
- **element.firstElementChild** or **element.lastElementChild → Returns the first (or last) HTML element contained (child) in the current node. There are also firstChild** and **lastChild** properties . The difference is that the latter can handle any node type, including text and comments.
- **element.children** → Returns a collection of all the HTML elements contained in the referenced node. The collection is not an Array, but **an HTMLCollection** , which is similar but more limited. We can also use **element.childNodes** , but in this case, it will also return the text and comment nodes as part of the collection.
- **element.parentNode** → Returns the parent node of an element.
- **element.nextElementSibling** → Returns the next sibling node. The **previousElementSibling** method returns the previous one. These properties only consider HTML nodes. We also have the **nextSibling** and **previousSibling** properties , which consider text and comments.

 # index.html

```
<!DOCTYPE>
<html>
    <head>
        <title>JS Example</title>
        <script src="./index.js" defer></script>
    </head>
    <body>
          <ul>
              <li id="firstListElement">Element 1</li>
              <li>Element 2</li>
```

```
            <li>Element 3</li>
        </ul>
    </body>
</html>
```

 # index.js

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

Query Selector

In addition to the methods discussed above for retrieving DOM elements, there are more advanced methods that use CSS selectors to return HTML elements that meet certain criteria. These methods can be called from the document or from any HTML node in the DOM.

- **document.querySelector("selector")** → Returns the first element that matches the selector.
- **document.querySelectorAll("selector")** → Returns a collection of all elements that match the selector.
- **element.closest("selector")** → Navigates up the DOM tree until it finds an ancestor node of the current one that matches the selector.

 # Selector examples

```
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" */
```

Example using **querySelector()** and **querySelectorAll()** :

 # index.html

```
<!DOCTYPE>
<html>
    <head>
        <title>JS Example</title>
        <script src="./index.js" defer></script>
    </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>
```

```
    </body>
</html>
```

index.js

```
<!DOCTYPE>
<html>
    <head>
        <title>JS Example</title>
        <script src="./index.js" defer></script>
    </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>
    </body>
</html>
```

Manipulating the DOM

Let's look at some methods that allow us to remove, add, or replace DOM elements.

- **document.createElement("tag")** → Creates an HTML element. It won't be in the DOM yet until we insert it (using appendChild, for example) into another DOM element.
- **document.createTextNode("text")** → Creates a text node that we can insert inside an HTML element. We could also use innerText instead, but this allows us, for example, to easily add a text node alongside other HTML nodes within the same element.
- **element.append(...childElements)** → Adds one or more elements to the end of the current node. They can also be text nodes, simply by passing a string as a parameter.
- **element.prepend(...childElements)** → Adds one or more elements to the beginning of the current node.
- **element.before(...childElements)** → Adds one or more elements before the current element (siblings).
- **element.after(...childElements)** → Adds one or more elements after the current element (siblings).
- **element.remove()** → The element is removed from the DOM.
- **element.replaceWith(...otherElements)** → Replaces the current element with one or more elements that are in the same position.
- **element.replaceChildren(...otherElements)** → Replaces all children of the current element with the elements passed to it as an argument. If none are passed, the element is left **empty** .

There are other methods such as **appendChild** , **insertBefore** , **removeChild** or **replaceChild** that are not worth explaining because the methods described above are more modern, easier to use, and allow you to do more things with less code.

 # index.html

```
<!DOCTYPE>
<html>
    <head>
        <title>JS Example</title>
        <script src="./index.js" defer></script>
    </head>
    <body>
        <ul>
            <li id="firstListElement">Element 1</li>
            <li>Element 2</li>
```

```
            <li>Element 3</li>
        </ul>
    </body>
</html>
```

index.js

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

Result

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

Attributes

Within HTML elements, and depending on the element type, there are standard attributes such as **name** , **id** , **href** , **src** , etc. These attributes can be accessed directly as properties of the object to read or modify them (example: **element.href = "http://domain.com"**).

Other useful properties and methods for working with attributes are:

- **element.attributes** → Returns the array with the attributes of an element
- **element.className** → Used to access (read or change) the class attribute. Other attributes that can be accessed directly include: element.id, element.title, element.style (CSS properties), etc.
- **element.classList** → Array of the element's CSS classes. Unlike className, which is a string with the classes separated by spaces, this attribute returns them as an array or list. It has very useful methods for consulting and modifying classes such as: **classList.contains("class")** → true if it has the class. **classList.replace("class1","class2")** : Removes the class "class1" and replaces it with class "class2". **classList.add("class1")** : Adds the class "class1" to the element. **classList.remove("class1")** : Removes the class "class1". **classList.toggle("class1")** : If it doesn't have "class1", adds it. Otherwise, removes it.
- **element.hasAttribute("attrName")** → Returns true if the element has an attribute with the specified name.
- **element.getAttribute("attrName")** → Returns the value of the attribute.
- **element.setAttribute("attrName", "newValue")** → Changes the value of the attribute.

 # index.html

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

 # index.js

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

The style attribute

The style attribute allows you to modify the CSS properties associated with an element. The CSS property to be modified must be written in **camel-case** , while in CSS, it is written in snake-case. For example, the background-color (CSS) attribute is accessed through **element.style.backgroundColor** . The value set for a property will be a string containing a valid CSS value for the attribute.

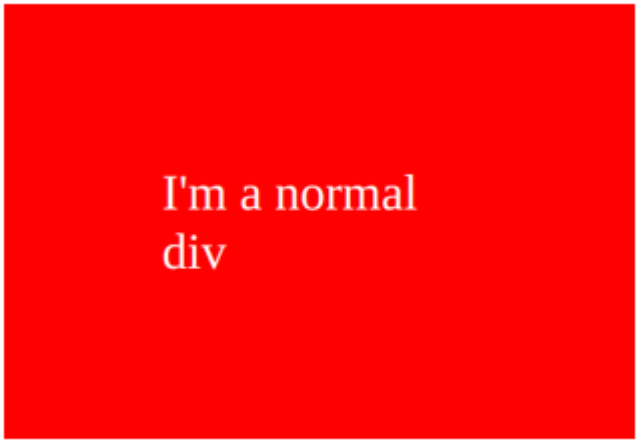
 # index.html

```
<!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>
```

 # new.js

```
<!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>
```



Events

When a user interacts with an application, a series of events occur (keyboard, mouse, etc.) that our code should handle appropriately. There are many events that can be captured and processed (listed here). We'll look at some of the most common ones.

Some types of events

In this section, we'll look at a small subset of the events that can be generated during the execution of a web application. These events can be generated by user interaction (mouse, keyboard, etc.) or when a milestone occurs during program execution (loading, animations, printer, etc.).

You can check out the existing event types on the official MDN website . There, for each type, you'll find one or more links to a list of specific events and their documentation. Below, we'll look at some of these events:

Keyboard events

- **keydown** → The user presses a key. If the key is held down for a while, this event will be generated repeatedly.
- **keyup** → Occurs when the user releases the key
- **keypress** → Similar to keydown. Press and lift action.

Mouse events

- **click** → This event occurs when the user clicks on an element (presses and lifts their finger from the button → mousedown + mouseup). It also fires when a touch event is received.
- **dblclick** → It is launched when the element is double-clicked.
- **mousedown** → Similar to keydown. Press and lift action.
- **mouseup** → This event occurs when the user lifts their finger from the mouse button.
- **mouseenter** → Fires when the mouse pointer enters the element.
- **mouseleave** → Fired when the mouse pointer leaves the element.
- **mousemove** → This event is called repeatedly when a mouse pointer moves while inside an element.

Touch events

- **touchstart** → Launched when a touch is detected on the touchscreen.
- **touchend** → Fired when your finger is lifted from the screen.
- **touchmove** → Fires when a finger is moved across the screen.

- **touchcancel** → This event occurs when a touch event is interrupted.

Form events

- **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 seleccionado 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 lanza cuando un formulario es enviado (el envío puede ser cancelado). Se aplica al elemento <form>.

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

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**).

Si llamamos a una función, podemos pasarle como parámetros la palabra reservada **this** y **event** pasar una referencia al elemento HTML (afectado por el evento) y el objeto con información del evento. Vamos a ver un ejemplo del evento click.

 # index.html

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

 # index.js

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

Podemos añadir una función manejadora de evento desde código a un elemento desde la propiedad correspondiente (onclick, onfocus, ...), o asignarles el valor **null** si queremos dejar de escuchar algún evento. Si utilizamos una funcón anónima en

lugar de una función flecha, podremos acceder al objeto **this**, que representará el elemento que recibe el evento.

 # index.js

```
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, ni tampoco elegir como se propaga el evento (ver más adelante).

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).

 # index.js

```
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.

 # index.js

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

function inputClick2(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
en 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 HTML que recibe el evento.
- **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 enlace, 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 (ver propagación de eventos más adelante).
- **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

Este objeto representa un evento que se ha producido por la acción del usuario sobre el ratón (click, movimiento).

- **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

Este objeto representa un evento que se produce por la interacción del usuario a través del teclado.

- **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)

Often, elements on a website overlap (are contained within) other elements. For example, if we click on a paragraph contained within a element, will the event be triggered on the paragraph, the , or both? Which one is triggered first?

For example, let's see what happens to these two elements (a div element inside another div) when we click on them.

 # index.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>
```

 # index.js

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

```
height: 100px;"></div>
</div>
```

If we click on the red element **#div1** , it will only print "You clicked: div1." However, if we click on the blue element **#div2** (which is inside the red element), it will print both messages (#div2 first).

In short, the event is first received by the element we're acting on, and then propagated to the container elements (including the document). This is called **event bubbling** . We can reverse the order of this process by adding a third parameter to the addEventListener method and setting it to true. This would be **event capture** .

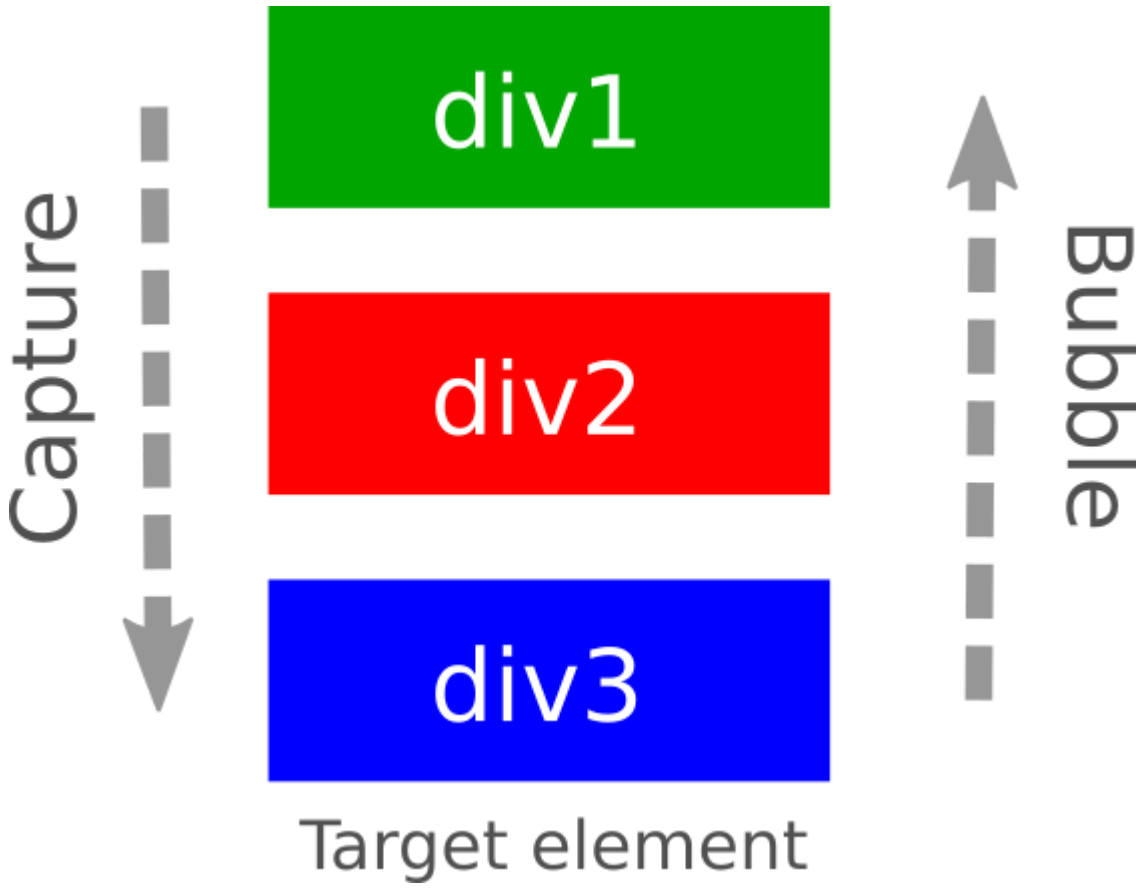
 # index.html

```
<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>
```

 # index.js

```
<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>
```

The eventPhase property of the event will indicate the propagation phase we are in: 1. Capture (parameter to true), 2: Target or element that receives the event, 3: Bubble (default)


div1
div2
div3
Capture
Bubble
Target element

Forms

In this section, we'll look at the different ways to retrieve values from a form. The form submission must be processed from its **submit event, without reloading or redirecting to another page. To do this, it's important to call the event's preventDefault()** method as soon as the form starts. This way, we override the default behavior (reloading the page).

Getting values from fields

The value of a form field is obtained from its **value** property . There are several ways to access a form field. Using the id to access it is unnecessary; you can access it from the form object using the value of the **name** attribute .

 # index.html

```
<form id="formProducto">
    <p><input type="text" name="nombre" id="nombre"
placeholder="Nombre" required></p>
    <p><input type="text" name="descripcion" id="descripcion"
placeholder="Descripcion" required></p>
    <button type="submit">Añadir</button>
</form>
```

 # index.js

```
<form id="formProducto">
    <p><input type="text" name="nombre" id="nombre"
placeholder="Nombre" required></p>
    <p><input type="text" name="descripcion" id="descripcion"
placeholder="Descripcion" required></p>
    <button type="submit">Añadir</button>
</form>
```

Multiple values

When collecting values from controls like **input[type=checkbox]** grouped by the same name, or from multiple-select lists, the value attribute won't work. In this case, accessing the element by name will return a collection of elements (RadioNodeList). If we want to obtain the values, for example, as an array of strings, we'll have to iterate through the collection, filtering out the selected elements (**checked**).

 # index.html

```
<form id="formPersona">
  <p><input type="text" name="nombre" id="nombre"
placeholder="Nombre" required></p>
  <p>
    Aficiones:
    <input type="checkbox" name="hobbies" value="tenis"> Tenis
    <input type="checkbox" name="hobbies" value="comer"> Comer
    <input type="checkbox" name="hobbies" value="viajar"> Viajar
    <input type="checkbox" name="hobbies" value="caminar"> Caminar
  </p>
  <button type="submit">Añadir</button>
</form>
```

 # index.js

```
<form id="formPersona">
  <p><input type="text" name="nombre" id="nombre"
placeholder="Nombre" required></p>
  <p>
    Aficiones:
    <input type="checkbox" name="hobbies" value="tenis"> Tenis
    <input type="checkbox" name="hobbies" value="comer"> Comer
    <input type="checkbox" name="hobbies" value="viajar"> Viajar
    <input type="checkbox" name="hobbies" value="caminar"> Caminar
  </p>
  <button type="submit">Añadir</button>
</form>
```

Preview image

If we have a file type input to select an image, and we want to preview its content in an **<img>** element on the page, we have 2 options.

The first would be to serialize it in **base64** . This is a format that allows any binary content to be represented in text format using a set of 64 characters. The image will take up a little more space, but we can send it to a server, for example, serialized within a JSON object along with the rest of the form fields (if the server requires that format to transfer data).

 # index.html

```
<form id="formPersona">
  <p>Nombre: <input type="text" name="nombre" placeholder="Nombre"
```

```
required></p>
  <p>Avatar: <input type="file" name="avatar" required
accept="image/*"></p>
  <p><img src="" alt="" id="imgPreview"></p>
  <button type="submit">Añadir</button>
</form>
```

 # index.js

```
<form id="formPersona">
  <p>Nombre: <input type="text" name="nombre" placeholder="Nombre"
required></p>
  <p>Avatar: <input type="file" name="avatar" required
accept="image/*"></p>
  <p><img src="" alt="" id="imgPreview"></p>
  <button type="submit">Añadir</button>
</form>
```

The second way is to use **createObjectURL** , and we'll see that in the next section.

Using FormData

If our server supports **multipart/form-data data, we can use the FormData** object to collect all the form information, including multi-choice or file-type fields. It's a Map-like collection that stores values in key/value format.

If we want to access the stored values, we'll use the **get** method , and for multiple values, **getAll** . If we want to preview image files without having to first transform them to Base64 (because we don't need to send a JSON object), we can use URL.createObjectUrl to generate a URL to an internal representation of the file (not the path to the file), which allows us to view the image. However, we must free the memory of this representation once the image is loaded, since otherwise, it would not be freed until the current document is closed.

 # index.html

```
<form id="formPersona">
    <p>Nombre: <input type="text" name="nombre"
placeholder="Nombre" required></p>
    <p>Avatar: <input type="file" name="avatar" required
accept="image/*"></p>
    <p><img src="" alt="" id="imgPreview"></p>
    <p>
```

```
        Aficiones:
        <input type="checkbox" name="hobbies" value="tenis"> Tenis
        <input type="checkbox" name="hobbies" value="comer"> Comer
        <input type="checkbox" name="hobbies" value="viajar"> Viajar
        <input type="checkbox" name="hobbies" value="caminar">
Caminar
    </p>
    <button type="submit">Añadir</button>
</form>
```

 # index.js

```
<form id="formPersona">
    <p>Nombre: <input type="text" name="nombre"
placeholder="Nombre" required></p>
    <p>Avatar: <input type="file" name="avatar" required
accept="image/*"></p>
    <p><img src="" alt="" id="imgPreview"></p>
    <p>
        Aficiones:
        <input type="checkbox" name="hobbies" value="tenis"> Tenis
        <input type="checkbox" name="hobbies" value="comer"> Comer
        <input type="checkbox" name="hobbies" value="viajar"> Viajar
        <input type="checkbox" name="hobbies" value="caminar">
Caminar
    </p>
    <button type="submit">Añadir</button>
</form>
```

Constraint Validation API

The Constraint Validation API **allows** you to manage the validation of an HTML form in conjunction with the HTML validators that can be used in the different controls on the form.

For example, we can check whether a form <form> or any individual form element is valid or not by calling the following methods on the reference to it:

- **checkValidity()** : Returns a boolean indicating whether the form/field is valid or not (checks if it has the :valid or :invalid pseudo-class).
- **reportValidity()** : Does the same thing but also displays the corresponding error messages to the user.

Additionally, we can set custom error messages and display them to the user using the **setCustomValidity("Message") method,** which associates a custom error message with the field.

 # index.html

```
<form id="formPersona">
    <p>Nombre: <input type="text" name="nombre"
placeholder="Nombre" required></p>
    <p>Avatar: <input type="file" id="avatar" name="avatar"
required accept="image/*"></p>
    <button type="submit">Añadir</button>
</form>
```

 # index.js

```
<form id="formPersona">
    <p>Nombre: <input type="text" name="nombre"
placeholder="Nombre" required></p>
    <p>Avatar: <input type="file" id="avatar" name="avatar"
required accept="image/*"></p>
    <button type="submit">Añadir</button>
</form>
```

Important: In a field where we set a custom message when we detect an error, this message must be set to an empty string when **there is no error** . If it is not empty, it is considered an error, and the field/form will be considered invalid.

HTML Templates

The <template> element represents an HTML fragment of a web application. It has two possible uses:

- **An HTML fragment** (DocumentFragment) that **isn't rendered** in the document and that we can clone to obtain a copy of the HTML structure, modify it to our liking, and add it to the DOM. This is the alternative we'll look at in this section.
- Some content is rendered, but with Shadow DOM enabled. Among other things, it will have its own styles and won't be affected by the CSS styles of the global document. This feature is enabled with the **shadowrootmode="open"** attribute .

Using this element allows us to create complex HTML structures without having to manually create each node (**createElement**) and tediously build the structure with append . It also saves us from using **innerHTML** if we're looking to reduce the amount of code, as this poses security (XSS attacks) and performance issues (especially when adding content to a node that already has content).

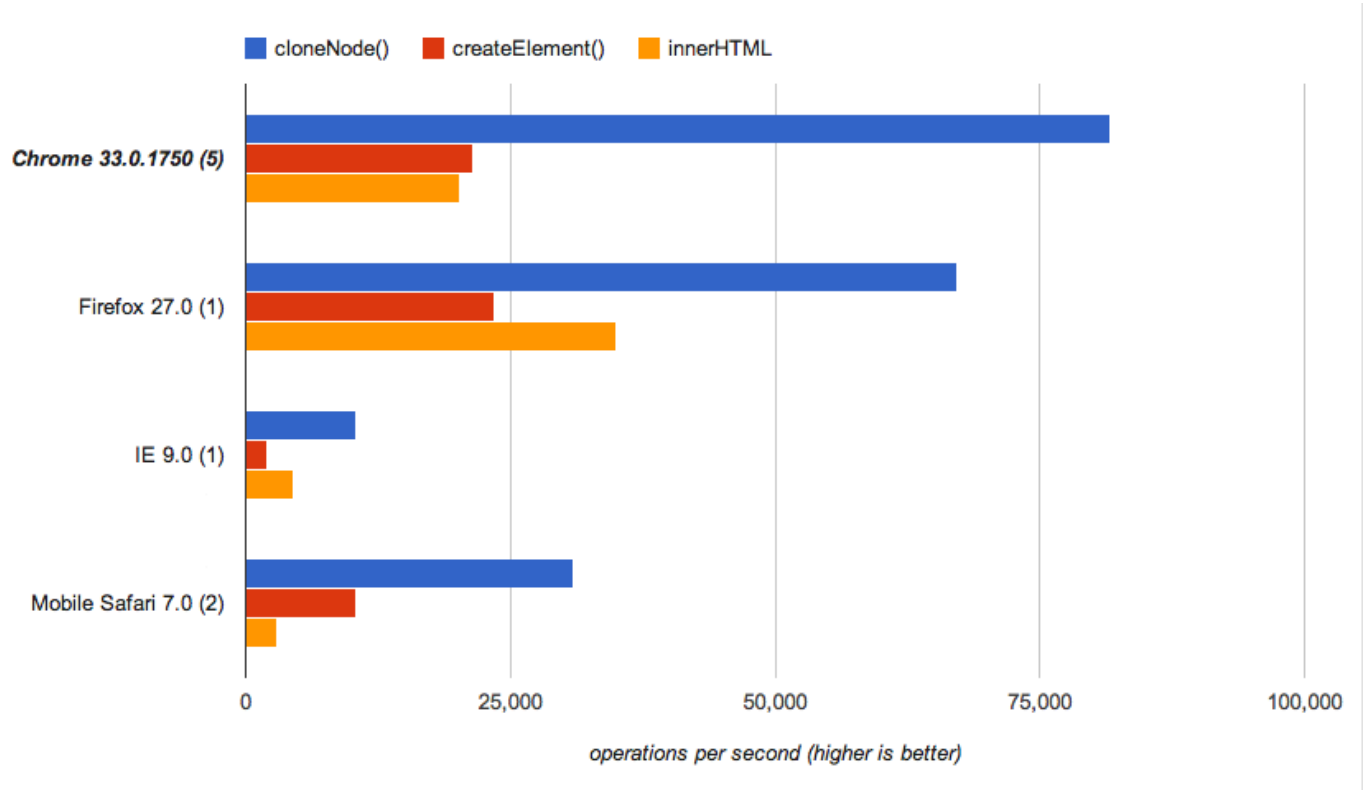
Testing in Chrome 75.0.3770 / Mac OS X 10.13.4		
	Test	**Ops/sec**
A	`container.innerHTML += '<p>Just some <span>text</span> here</p>';`	332 ±2.76% 100% slower
B	`container.innerHTML = '<p>Just some <span>text</span> here</p>';`	47,733 ±1.31% 66% slower
C	`$('#container').append('<p>Just some <span>text</span> here</p>');`	14,926 ±1.32% 89% slower

D	`var p = document.createElement("p");`
	`p.innerHTML = 'Just some <span>text</span> here';`
	`container.appendChild(p);`
		39,265 ±3.35% 72% slower

E	`var p = document.createElement("p");`
	`var s = document.createElement("span");`
	`s.appendChild( document.createTextNode("text ") );`
	`p.appendChild( document.createTextNode("Just some ") );`
	`p.appendChild( s );`
	`p.appendChild( document.createTextNode(" here") );`
	`container.appendChild(p);`
		140,137 ±1.97% fastest

F	`container.insertAdjacentHTML('beforeend', '<p>Just some <span>text</span> here</p>');`	46,428 ±1.75% 67% slower

Using templates and cloning template nodes is the best option in terms of performance, even considering that the performance of operations can vary significantly between browsers and versions.



Below, we'll look at two examples in which we'll add elements to the DOM using createElement vs. <template>. These are two simple examples where we have a form for entering a person's information (name, hobbies, and avatar).

Example: Without using <template>

In this first example, we create the row (<tr>) and then manually create the internal <td> elements, as well as the <img> element for the avatar, and then nest them correctly using the append method.

 # index.html

```
    <form id="formPersona">
      <p><input type="text" name="nombre" placeholder="Nombre"
  required></p>
      <p>
        Aficiones:
        <input type="checkbox" name="hobbies" value="tenis"> Tenis
        <input type="checkbox" name="hobbies" value="comer"> Comer
        <input type="checkbox" name="hobbies" value="viajar"> Viajar
        <input type="checkbox" name="hobbies" value="caminar"> Caminar
      </p>
```

```
    <p>Avatar: <input type="file" name="avatar" required
  accept="image/*"></p>
    <p><img src="" alt="" id="imgPreview"></p>
    <button type="submit">Añadir</button>
  </form>
  <table id="users">
    <thead>
      <tr>
        <td>Avatar</td>
        <td>Nombre</td>
        <td>Aficiones</td>
      </tr>
    </thead>
    <tbody></tbody>
  </table>
```

 # index.js

```
  <form id="formPersona">
    <p><input type="text" name="nombre" placeholder="Nombre"
  required></p>
    <p>
      Aficiones:
      <input type="checkbox" name="hobbies" value="tenis"> Tenis
      <input type="checkbox" name="hobbies" value="comer"> Comer
      <input type="checkbox" name="hobbies" value="viajar"> Viajar
      <input type="checkbox" name="hobbies" value="caminar"> Caminar
    </p>
    <p>Avatar: <input type="file" name="avatar" required
  accept="image/*"></p>
    <p><img src="" alt="" id="imgPreview"></p>
    <button type="submit">Añadir</button>
  </form>
  <table id="users">
    <thead>
      <tr>
        <td>Avatar</td>
        <td>Nombre</td>
        <td>Aficiones</td>
      </tr>
    </thead>
    <tbody></tbody>
  </table>
```

Example: Using <template>

The following example is the same, but with the addition of a <template> element containing the structure we're going to generate. In the code, we clone the template content and modify it (adding text and the image's src attribute), then add it to the table.

 # index.html

```
<body>
  <form id="formPersona">
    <!-- Mismo formulario -->
  </form>
  <table id="users">
    <!-- Misma tabla -->
  </table>

  <template id="userTemplate">
    <tr>
      <td><img src="" alt=""></td>
      <td></td>
      <td></td>
    </tr>
  </template>
</body>
```

 # new.js

```
<body>
  <form id="formPersona">
    <!-- Mismo formulario -->
  </form>
  <table id="users">
    <!-- Misma tabla -->
  </table>

  <template id="userTemplate">
    <tr>
      <td><img src="" alt=""></td>
      <td></td>
      <td></td>
    </tr>
```

```
    </template>
  </body>
```

As you can see, the code is simpler and the structure is much easier to create, since it's HTML code that we add directly to the document. The more complex the structure, the more code we'll save and the better performance we'll achieve.