INDICE: Document Object Model y Eventos

DOM - Document Object Model

El DOM es un conjunto de utilidades específicamente diseñadas para manipular documentos XML, y por tanto documentos HTML.

El DOM transforma el archivo HTML en un árbol de nodos jerárquico, fácilmente manipulable.

Los nodos más importantes son:

- ***Document***: Representa el nodo ráiz. Todo el documento HTML.
- ***Element***: Representa el contenido definido por un par de etiquetas de apertura y cierre, lo que se conoce como un tag HTML. Puede tener como hijos más nodos de tipo Element y también atributos.
- ***Attr***: Representa el atributo de un elemento.
- ***Text***: Almacena el contenido del texto que se encuentra entre una etiqueta HTML de apertura y cierre.

Nodo Document.

La interfaz document representa cualquier página web cargada en el navegador y sirve como punto de entrada al árbol DOM. Es decir:

- Cuando un documento HTML se carga en un navegador web, se convierte en un objeto de documento.
- El objeto de documento es el nodo raíz del documento HTML.
- La interfaz Document representa cualquier página web cargada en el navegador y sirve como punto de entrada al contenido de la página (El árbol DOM).
- El DOM incluye elementos como (<body> y <table>), entre muchos otros, y proporciona funcionalidad que es global al documento, como obtener la URL de la página y crear nuevos elementos en el documento.

Algunas de las propiedades serían:

https://developer.mozilla.org/es/docs/Web/API/Document#properties

```
console.log(document.head);
console.log(document.title);
console.log(document.body);
console.log(document.domain);
```

DOMContentLoaded, defer

- ***DOMContentLoaded***: El navegador HTML está completamente cargado y el árbol DOM está construido, pero es posible que los recursos externos como <img> y hojas de estilo no se hayan cargado aún.

Por ejemplo. A través de un escuchador de eventos podría realizar una acción una vez cargado el DOMContentLoaded

```
document.addEventListener("DOMContentLoaded", () => {
    console.log(document.querySelector("h1"));
});
```

- ***defer***: E l atributo defer indica al navegador que no espere por el script. En su lugar debe seguir procesando el HTML y construir el DOM. El script carga en segundo plano y se ejecuta cuando el DOM está completo.

Por lo tanto, cuando colocamos defer en la ejecución del script estamos obligando a ejecutarse el script cuando el DOM está ya listo, pero antes del evento DOMContentLoaded

Defer no funciona igual con todos los navegadores.

Recorriendo el DOM

Para poder recorrer el DOM, lo podemos hacer a través de un API JavaScript con métodos para acceder y manipular los nodos. Algunas de estas funciones son:

- ***getElementById(id)*** : Devuelve el elemento con un id específico.
- ***getElementsByName(name)*** Devuelve los elementos que un name (nombre) específico.
- ***getElementsByTagName(tagname)*** : Devuelve los elementos con un nombre de tag específico.
- ***getElementsByClassName(classname)*** : Devuelve los elementos con un nombre de clase específico.
- ***getAttribute(attributeName)*** : Devuelve el valor del atributo con nombre attributeName
- ***querySelector(selector)*** : Devuelve un único elemento que corresponda con el selector , ya sea por tag, id, o clase. Si hay varios devuelve el primer elemento que coincida con el grupo especificado.
- ***querySelectorAll(selector)*** : Devuelve un array con los elementos que correspondan con la query introducida en selector.

Manipulando el DOM

De igual manera podemos manipular el DOM con las siguientes funciones:

createElement(name) : Crea un elemento HTML con el nombre que le pasemos en el parámetro name .

createTextNode(text) : Crea un nodo de texto que puede ser añadido a un elemento HTML.

createTextAttribute(attribute) : Crea un atributo que puede ser añadido posteriormente a un elemento HTML.

appendChild(node) : Nos permite hacer hijo un elemento a otro.

insertBefore(new, target) : Permite insertar un elemento o nodo new antes del indicado en target .

removeAttribute(attribute) : Elimina el atributo de nombre attribute del nodo desde el que se le llama.

removeChild(child) : Elimina el nodo hijo que se indica con child

replaceChild(new, old) : Reemplaza el nodo old por el que se indica en el parámetro new .

Ejemplo:

```
<ul id="mi-lista"></ul>

<script>
  // elemento donde vamos a incorporar los <li>
  const lista = document.getElementById("mi-lista");

  // Creamos el <li> con createElement
  const li = document.createElement("li");

  // Agregamos texto al <li>
  li.textContent = "Mi <li> dinámico";

  // Finalmente incorporamos al <ul>
  lista.appendChild(li);
</script>
```

Otra opción (NO RECOMENDADO)

```
const lista = document.getElementById("mi-lista");

const arrayItem = ["dato 1", "dato 2", "dato 3"];

arrayItem.forEach((item) => {
  lista.innerHTML += `
            <li>${item}</li>
          `;
});
```

Así podemos generar lo que se denomina REFLOW : https://developer.mozilla.org/en-US/docs/Glossary/reflow

Ocurre cuando un navegador debe pintar parte o la totalidad de una página web nuevamente y en repetidas ocasiones.

Para solucionar este problema existe los Fragment.

Propiedades de los nodos del DOM

Todos los nodos tienen algunas propiedades que pueden ser muy útiles para las necesidades de nuestros desarrollos:

attributes : Nos devuelve un objeto con todos los atributos que posee un nodo.

className : Permite setear o devolver el nombre de la clase (para CSS) que tenga el nodo si la tiene.

id : Igual que className pero para el atributo id

innerHTML : Devuelve o permite insertar código HTML (incluyendo tags y texto)

dentro de un nodo.

nodeName : Devuelve o nombre del nodo, si es un devolverá DIV .

nodeValue : Devuelve el valor del nodo. Si es de tipo element devolverá null . Pero por ejemplo si es un nodo de tipo texto, devolverá ese valor.

style : Permite insertar código CSS para editar el estilo.

tagName : Devuelve el nombre de la etiqueta HTML correspondiente al nodo. Similar a nodeName, pero solo en nodos de tipo tag HTML.

title : Devuelve o permite modificar el valor del atributo title de un nodo.

childNodes : Devuelve un array con los nodos hijos del nodo desde el que se llama.

firstChild : Devuelve el primer hijo.

lastChild : Devuelve el último hijo.

previousSibling : Devuelve el anterior "hermano" o nodo al mismo nivel.

nextSibling : Devuelve el siguiente "hermano" o nodo al mismo nivel.

ownerDocument : Devuelve el nodo raiz donde se encuenta el nodo desde el que se llama.

parentNode : Devuelve el nodo padre del nodo que se llama.

Fragment.

Se utiliza como una versión ligera de Document, que almacena un segmento de una estructura de documento compuesta de nodos como si fuera un documento normal.

En un Fragment guardamos todos los nodos HTML que posteriormente pintaremos en el DOM y así se evita el Reflow.

Por ejemplo:

```
<script>
  const lista = document.getElementById("mi-lista");

  const arrayItem = ["dato 1", "dato 2", "dato 3"];

  const fragment = document.createDocumentFragment();

  arrayItem.forEach((item) => {
    const li = document.createElement("li");
    li.textContent = item;
    fragment.appendChild(li);
  });

  lista.appendChild(fragment);
</script>
```

Métodos útiles serían firstChild e insertBefore() por ejemplo.

```
const nodoPrimero = fragment.firstChild; // nos devuelve el primer elemento.

// La sintaxis de insertBefore sería: parentNOde.insertBefore(newNode, referenceNode)

fragment.insetBefore(li, nodoPrimero);
```

Template HTML.

Para solucionar los problemas de reflow y no hacer muy tedioso la creación de nodos en javascript disponemos de los template.

El elemento HTML <template> es un mecanismo para mantener el contenido HTML del lado del cliente que no se renderiza cuando se carga una página, pero que posteriormente puede ser instanciado durante el tiempo de ejecución empleando JavaScript.

Así por ejemplo colocamos la etiqueta template en HTML y posteriormente en JavaScript podemos instanciar elementos haciendo uso de cloneNode como se puede ver en este ejemplo:

```
<ul id="lista-dinamica"></ul>
<template>
  <li class="list">
    <b>nombre: </b> <span class="text-danger">descripción...</span>
  </li>
</template>
```

```
const lista = document.getElementById("lista-dinamica");
const arrayItem = ["item 1", "item 2", "item 3"];

const fragment = document.createDocumentFragment();
const template = document.querySelector("#template-li").content;

arrayItem.forEach((item) => {
```

```
  template.querySelector("span").textContent = item;
  const clone = template.cloneNode(true);
  // const clone = document.importNode(template, true);
  fragment.appendChild(clone);
});

lista.appendChild(fragment);
```

Más información en:

https://developer.mozilla.org/es/docs/Web/HTML/Elemento/template

Modificar atributos y clases.

Para acceder a los **atributos** utilizamos:

Element.getAttribute(“nombreAbributo”) // Obtenemos el valor de un atributo

Element.setAttribute(“nombreAtributo”,valor) //Modificamos el valor de un atributo

Para las **clases** usamos los métodos:

Element.classList.add(“clase1”,”clase2”) //Añado dos clases de nombre clase1 y clase2

Element.classList.remove(“clase1”,”clase2”) // Elimino la clase1 y clase2

Element.classList.toggle(“clase1”) //Si no tiene la clase la pone, y si la tiene la quieta.

Element.classList.contains(“clase1”) // Devuelve True o False según tenga o no dicha clase el elemento.

Element.classList.replace(“oldClase”, “nuevaClase”) // Sustituye una clase por otra.

EVENTOS:

Un evento es cualquier cosa que sucede en nuestro documento. Ejemplos:

El contenido se ha leído.

El contenido se ha cargado.

El usuario mueve el ratón.

El usuario pulsa una tecla.

La ventana se ha cerrado.

Hace unos pocos años los eventos se trataban así:

```
<p onclick=”saludo()”> Ejemplo de párrafo</p>  // NO USARLO
```

Esta forma no se debe de usar porque mezcla lenguajes, incrustando javascript en html que hace difícil el mantenimiento del código y su depuración.

La forma correcta sería:

```
Element.addEventListener(“evento”, callback)

//callback función anónima que se dispara cuando se lanza el evento.
```

```
/*
Otros eventos de ratón
- mouseenter - Cuando entramos en la zona que tiene el evento.
- mouseleave - Cuando salimos de la zona que tiene el evento.
- mousedown - Cuando pulsamos el botón izquierdo del ratón.
- mouseup - Cuando soltamos el botón izquierdo del ratón.
- mousemove - Cuando movemos el ratón.
*/

/*
Otros eventos de teclado en
- keydown -cuando pulsamos una tecla.
- keyup - cuando soltamos una tecla.
```

```
- keypress - cuando pulsamo una tecla y no la soltamos.
```

Algunos ejemplos:

```
const boton = document.getElementById("boton");
const caja = document.getElementById("caja");

boton.addEventListener("click", (e) => {
  console.log("click");
});
boton.addEventListener("dblclick", (e) => {
  console.log("doble click");
});

boton.addEventListener("mouseenter", (e) => {
  caja.classList.replace("red", "blue");
});
```

La (e) hace referencia a event.

Event es un objeto y sólo vive cuando hay un evento.

Event.target es el lugar o punto donde se ha producido un evento.

Si por ejemplo pulso en un botón, en el click event.target me daría muchísima información referente al botón.

Nota:

Mucho cuidado porque cuando pasamos una función a un evento, no se pueden pasar parámetros a no ser que lo envolvamos en una función flecha. Por ejemplo:

```
function saludar(nombre =”desconocido”){
      Alert(`Hola ${nombre}  `);
}
 Botonlanzar.addEventListener(“click”,saludar);
// al pulsar en el click no aparecería por pantalla Hola desconocido,
// aparecería Hola [object MouseEvent]
```

El motivo es que cuando pasamos en un evenListener una función sólo podríamos pasar como parámetros el event o evento.

Para arreglar esto usaremos las funciones flechas y por tanto lo anterior se arreglaría:

```
function saludar(nombre =”desconocido”){
```

```
      Alert(`Hola ${nombre}  `);
}
 Botonlanzar.addEventListener("click", () => saludar());
// Ahora ya sí aparece Hola desconocido
```

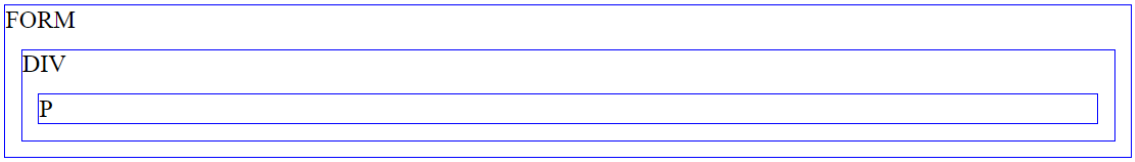
Flujos de eventos (burbuja y captura)

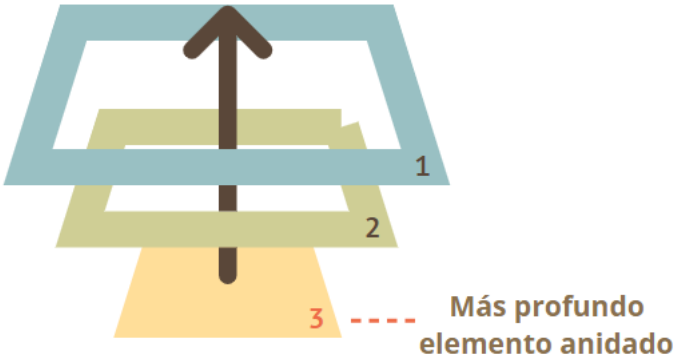
Cuando un evento se origina, automáticamente tiene una propagación a lo largo del DOM. Generalmente va desde el elemento más interno hasta el más externo(Document)

Propagación:

El principio de propagación es simple.

Cuando un evento ocurre en un elemento, este primero ejecuta los manejadores que tiene asignados, luego los manejadores de su padre, y así hasta otros ancestros.





Un clic en el elemento del interior <p> primero ejecuta el evento click :

- En ese <p>.
- Luego en el de arriba.
- Luego en el <form> de más arriba.
- Y así sucesivamente hasta el objeto document.

Así si hacemos clic en <p>, entonces veremos 3 alertas: p → div → form.

Este proceso se conoce como "**propagación**" porque los eventos "se propagan" desde el elemento más al interior, a través de los padres, como una burbuja en el agua.

event.target

El elemento anidado más profundo que causó el evento es llamado elemento objetivo, accesible como **event.target**

Nota la diferencia de this (=event.currentTarget):

- **event.target** – es el elemento "objetivo" que inició el evento, no cambia a través de todo el proceso de propagación.
- **this** – es el elemento "actual", el que tiene un manejador ejecutándose en el momento.
- Por ejemplo, si tenemos un solo manejador form.onclick(o con addListener) , este puede atrapar todos los clicks dentro del formulario. No importa dónde el clic se hizo, se propaga hasta el <form> y ejecuta el manejador.
- En el manejador form.onclick:
    - this (=event.currentTarget) es el elemento <form>, porque el manejador se ejecutó en él.
    - event.target es el elemento actual dentro del formulario al que se le hizo clic.

Parar la Propagación:

Una propagación de evento empieza desde el elemento objetivo hacia arriba. Normalmente este continúa hasta <html> y luego hacia el objeto document, algunos eventos incluso alcanzan window, llamando a todos los manejadores en el camino.

Pero cualquier manejador podría decidir que el evento se ha procesado por completo y detener su propagación.

El método para esto es **event.stopPropagation().**

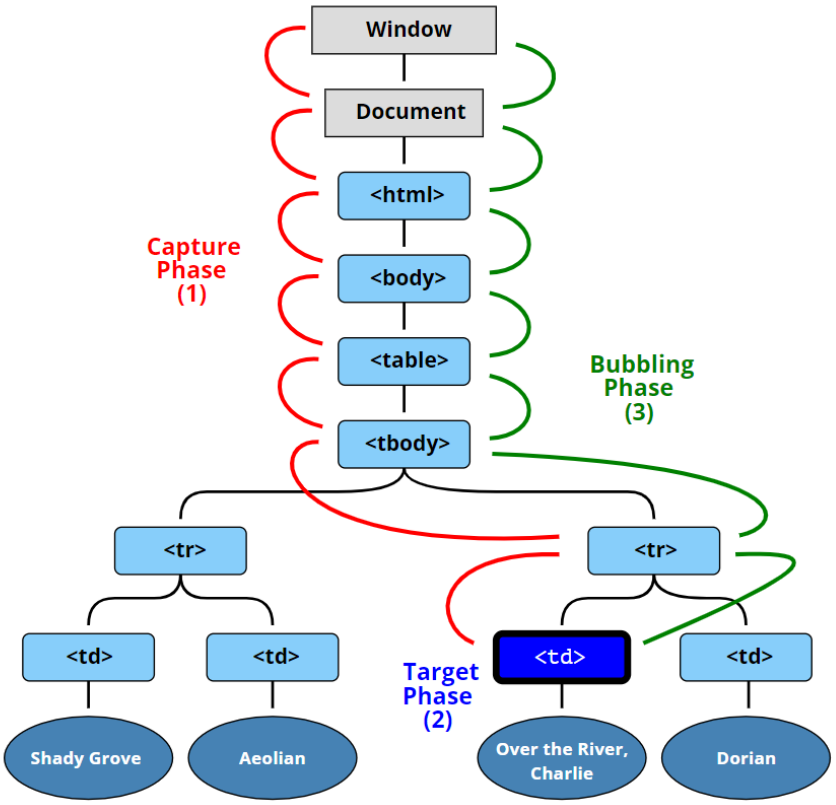
Captura:

Hay otra fase en el procesamiento de eventos llamada "captura". Es raro usarla en código real, pero a veces puede ser útil.

El estándar de eventos del DOM describe 3 fases de la propagación de eventos:

1. Fase de captura – el evento desciende al elemento.
2. Fase de objetivo – el evento alcanza al elemento.

3. Fase de propagación – el evento se propaga hacia arriba del elemento.



Se explica así: por un clic en <td> el evento va primero a través de la cadena de ancestros hacia el elemento (fase de captura), luego alcanza el objetivo y se desencadena ahí (fase de objetivo), y por último va hacia arriba (fase de propagación), ejecutando los manejadores en su camino.

Para atrapar un evento en la fase de captura, necesitamos preparar la opción capture como true en el manejador:

```
elem.addEventListener(..., {capture: true})
// o, solo "true" es una forma más corta de {capture: true}
elem.addEventListener(..., true)
```

Hay dos posibles valores para la opción **capture**:

- Si es **false** (por defecto), entonces el manejador es preparado para la fase de propagación.
- Si es **true**, entonces el manejador es preparado para la fase de captura.

Resumen:

Cuando ocurre un evento, el elemento más anidado dónde ocurrió se reconoce como el "elemento objetivo" (**event.target**).

- Luego el evento se mueve hacia abajo desde el documento raíz hacia event.target, llamando a los manejadores en el camino asignados con addEventListener(..., true) (true es una abreviación para {capture: true}).
- Luego los manejadores son llamados en el elemento objetivo mismo.
- Luego el evento se propaga desde event.target hacia la raíz, llamando a los manejadores que se asignaron usando on<event>, atributos HTML y addEventListener sin el 3er argumento o con el 3er argumento false/{capture:false}.

Cada manejador puede acceder a las propiedades del objeto event:

- event.target – el elemento más profundo que originó el evento.
- event.currentTarget (=this) – el elemento actual que maneja el evento (el que tiene al manejador en él)
- event.eventPhase – la fase actual (captura=1, objetivo=2, propagación=3).

Cualquier manejador de evento puede detener el evento al llamar **event.stopPropagation(),** pero no es recomendado porque no podemos realmente asegurar que no lo necesitaremos más adelante, quizá para completar diferentes cosas.

Más información en:

https://es.javascript.info/bubbling-and-capturing

Delegación de Eventos:

La captura y la propagación nos permiten implementar uno de los más poderosos patrones de manejo de eventos llamado delegación de eventos.

Consiste en asignar el evento al elemento más alto (por ejemplo, el document) y con selectores condicionales activar el evento sólo a los elementos que necesitemos, evitando así la propagación.

Así con `evento.target.matches("nodo a buscar") // en CSS`