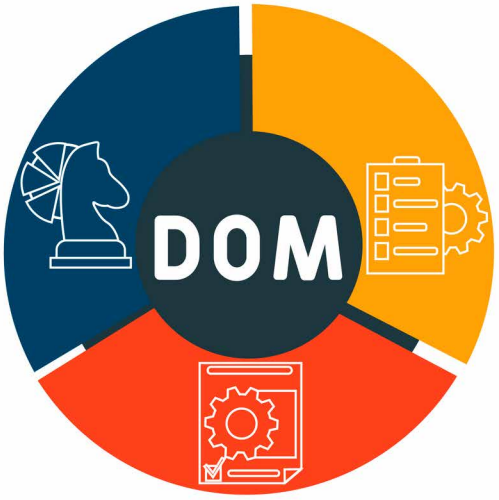
Programación con JavaScript I

DOM

Introducción

El lenguaje JavaScript puede interpretarse desde el lado del cliente o desde el lado del servidor, el lado del cliente existe para convertir documentos HTML estáticos en aplicaciones web interactivas. Por lo tanto, la creación de secuencias de comandos del contenido de las páginas web es realmente el propósito central de JavaScript.



En este tema, aprenderás que cada objeto de ventana en un navegador tiene una propiedad de documento que se refiere a un objeto document. Este objeto representa el contenido de la ventana, sin embargo, el objeto no es independiente, es el objeto central del DOM para representar y manipular el contenido del documento.

Cuando abres una página web en tu navegador, el navegador recupera el texto HTML de la página y lo analiza. El navegador crea un modelo de la estructura del documento y usa este modelo para dibujar la página en la pantalla.

Esta representación del documento es uno de los juguetes que un programa JavaScript tiene disponible en su sandbox. Dicha representación del documento es una estructura de datos que puedes leer o modificar, además actúa como una estructura de datos en vivo: cuando se modifica, la página en la pantalla se actualiza para reflejar los cambios.

Explicación

DOM

Para poder definir DOM, es necesario comprender que uno de los objetos más importantes en la programación de JavaScript del lado del cliente es el objeto document, que representa el documento HTML que se muestra en una ventana del navegador. La Interfaz de programación de aplicaciones (API) para trabajar con documentos HTML, se conoce como **Modelo de Objetos de Documento**, o DOM (por sus siglas en ingles). Por eso el DOM es tan fundamental para la programación de JavaScript del lado del cliente, es la representación del documento HTML como una estructura de nodos y objetos, estos elementos tienen propiedades y métodos. En esencia, DOM permite conectar las páginas web a los scripts.

Los documentos HTML contienen elementos HTML anidados entre sí, formando un árbol. Considera el siguiente documento HTML simple:

```
<html>

<head>
<title>Documento ejemplo</title>
</head>

<body>
<h1>Un Documento HTML</h1>
<p>Este es un <i>simple</i> documento.
</body>


</html>
```

Figura 1. Documento HTML simple.

La etiqueta <html> de nivel superior contiene las etiquetas <head> y <body>; la etiqueta <head> contiene una etiqueta <title>; y la etiqueta <body> contiene etiquetas <h1> y <p>; las etiquetas <title> y <h1> contienen cadenas de texto, y la etiqueta <p> contiene dos cadenas de texto con una etiqueta <i> entre ellas.

Flanagan (2020), explica que la API DOM refleja la estructura de árbol de un documento HTML. Para cada etiqueta HTML en el documento, hay un objeto de Elemento JavaScript correspondiente, y para cada ejecución de texto en el documento, hay un objeto de Texto correspondiente. Las clases Element y Text, así como la clase Document en sí, son todas subclases de la clase **Nodo** (Node) más general. Los objetos **Node** están organizados jerárquicamente en una estructura de árbol que JavaScript puede consultar y recorrer utilizando la API DOM. La representación DOM de este documento es el árbol que se muestra en la figura 2.

Explicación

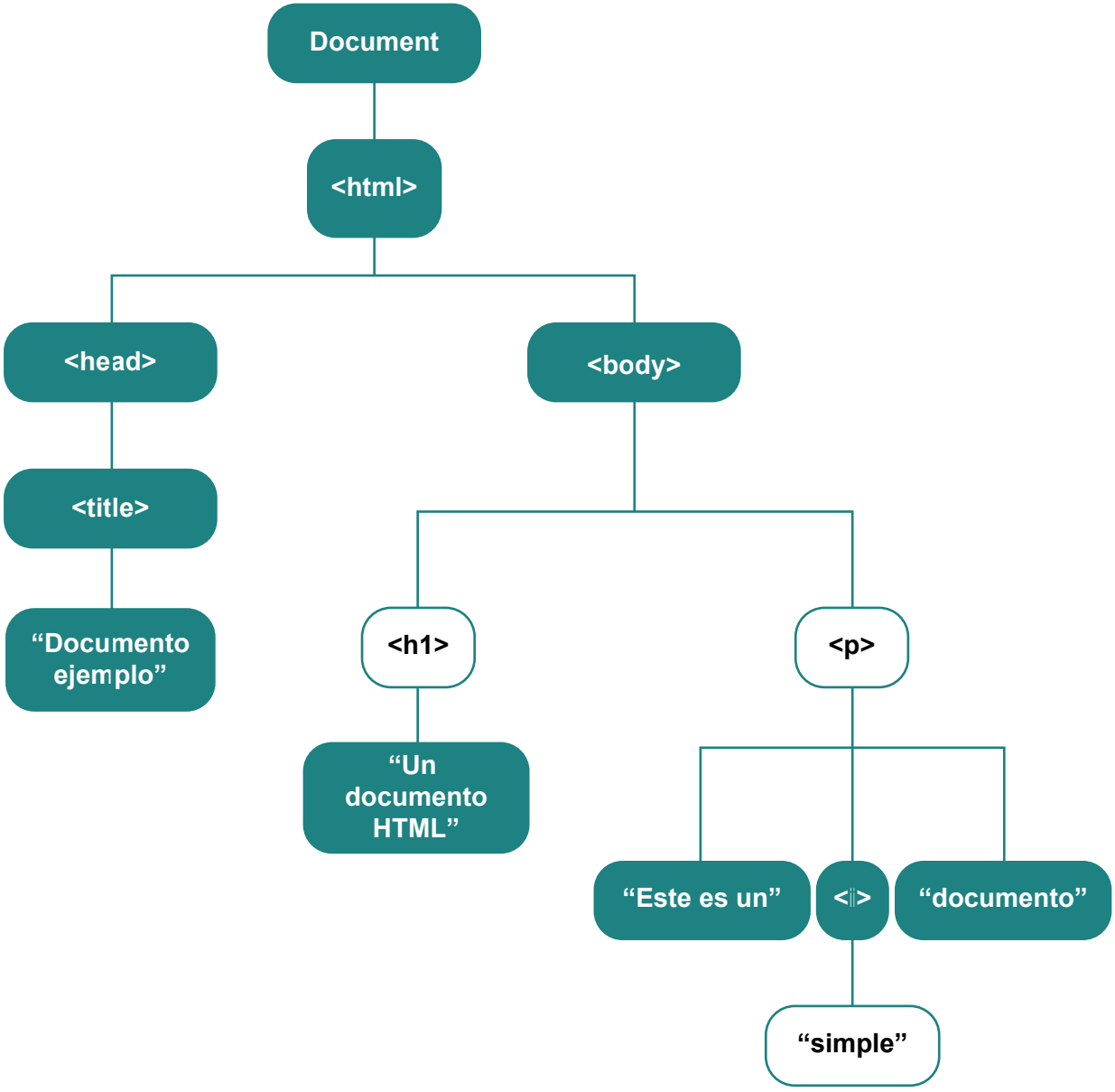


Figura 2. Representación en árbol de un documento HTML.

Si aún no estás familiarizado con las estructuras de árbol en la programación, es útil saber que toman terminología prestada de los árboles genealógicos. En el conjunto de nodos, cualquier número de niveles por debajo de otro nodo es el descendiente de ese nodo, y el padre, el abuelo y todos los demás nodos por encima de un nodo son los antepasados de ese nodo.

Explicación

La API DOM incluye métodos para crear nuevos nodos de element y text, para insertarlos en el documento como elementos hijo de otros objetos de element. También existen métodos para mover elementos dentro del documento y para eliminarlos por completo. Mientras que una aplicación del lado del servidor puede producir una salida en forma de cadenas con el uso de **console.log**(), una aplicación de JavaScript del lado del cliente puede producir una salida HTML formateada construyendo o manipulando el árbol del documento usando la API DOM.

Cada objeto de nodo DOM tiene una propiedad

- Node*Type*, esta contiene un código (número) que identifica el tipo de nodo. Los elementos tienen el código 1, también se define como la propiedad constante Node.ELEMENT_NODE.
- Los nodos de texto, representan una sección de texto en el documento, obtienen el código 3 (Node.TEXT_NODE).
- Los comentarios tienen el código 8 (Nodo.COMMENT_NODE).

Hay una clase de JavaScript correspondiente a cada tipo de etiqueta HTML, y cada aparición de la etiqueta en un documento está representada por una instancia de la clase. La etiqueta <body>, por ejemplo, está representada por una instancia de **HTMLBodyElement**, y una etiqueta <table> está representada por una instancia de **HTMLTableElement**.

Los objetos de elementos de JavaScript tienen propiedades que se corresponden con los atributos HTML de las etiquetas. Por ejemplo, las instancias de HTMLImageElement, que representan etiquetas <img>, tienen una propiedad src que corresponde al atributo src de la etiqueta. El valor inicial de la propiedad src es el valor del atributo que aparece en la etiqueta HTML, y establecer esta propiedad con JavaScript cambia el valor del atributo HTML (y hace que el navegador cargue y muestre una nueva imagen). La mayoría de las clases de elementos de JavaScript solo reflejan los atributos de una etiqueta HTML, pero algunas definen métodos adicionales. Las clases **HTMLAudioElement** y HTMLVideoElement, por ejemplo, definen métodos como **play ()** y **pause ()** para controlar la reproducción de archivos de audio y video.

Si bien es cierto que DOM no es un lenguaje de programación, su importancia radica en que, sin él, JavaScript no tiene ningún modelo o noción de las páginas web ni de los elementos con los que usualmente es relacionado. Cada elemento dentro de la página web (documento) es parte del modelo de objeto del documento, así se puede acceder y manipularlos utilizando el DOM y un lenguaje de escritura, como JavaScript para efectos de este curso, sin embargo, DOM fue diseñado para ser independiente de cualquier lenguaje de programación y permite que la presentación estructural del documento sea disponible desde una API y no se requiere nada en especial para poder utilizar el DOM. Los diferentes navegadores ya tienen adaptadas distintas directrices DOM para hacer accesibles las páginas al script.

Explicación

De acuerdo con Kantor (2021), cuando creamos un script dentro de una página web, con la instrucción <script> por ejemplo, inmediatamente está disponible para usarlo con el API, accediendo a los elementos document o window, para manipular el documento mismo o alguna de sus partes. La programación DOM hace algo tan simple como abrir un mensaje de alerta con la función alert() desde el objeto window .

```
<body onload="window.alert('Bienvenido!');">
```

Figura 3. Script dentro de un documento html.

O permite hacer cosas más complejas y sofisticadas para crear un nuevo contenido como el siguiente ejemplo:

```
<html>

<head>
<script>
// ejecuta la función cuando carga la página web
window.onload = function () {
// crea un par de elementos HTML en la página
var cabecera = document.createElement("h1");
var textoCabecera = document.createTextNode("hola mundo");
cabecera.appendChild(textoCabecera);
document.body.appendChild(cabecera);
}
</script>
</head>

<body>
</body>
```

Figura 4. Creación de nuevos objetos HTML.

Selección y modificación nodos

Para seleccionar y modificar los nodos, a menudo resulta útil navegar por estos vínculos entre padres, hijos y hermanos, pero si se quiere encontrar un nodo específico en el documento, llegar a él comenzando en document.body y siguiendo una ruta fija de propiedades es una mala idea. Al hacerlo, se incorporan suposiciones a nuestro programa sobre la estructura precisa del documento, una estructura que quizás se desee cambiar más adelante, otro factor de complicación es que los nodos de texto se crean incluso para el espacio en blanco entre nodos. La etiqueta <body> del de la figura 1 no tiene solo tres elementos secundarios (<h1> y dos elementos <p>), sino que en realidad tiene siete: esos tres, más los espacios antes, después y entre ellos.

Explicación

Entonces, si quieres seleccionar el atributo href de la etiqueta de enlace a un documento, puedes hacerlo, al indicarle que obtenga el primer vínculo del documento, con la instrucción como se ve en la instrucción de la figura 5.

```
…
<p><a href="http://cdctecmilenio.mx">Lernit</a></p>
<script>
        let enlace = document.body.getElementsByTagName("a")[0];
        console.log(enlace.href);
</script>
```

Figura 5. Selección de atributos.

Y la salida de este código regresaría algo como esto:

https://cdctecmilenio.mx/

Todos los nodos de elementos tienen un método **getElementsByTagName**, que recopila todos los elementos con el nombre de etiqueta dado que son descendientes (hijos directos o indirectos) de ese nodo y los devuelve como un objeto similar a una matriz.

Para encontrar un solo nodo específico, puedes asignarle un atributo id y usar **document.getElementById** en su lugar.

```
<p> Mi perro Rufo: </p>
<p> <img id = "rufo" src = "img/rufo.png"> </p>
<script>
let perro = document.getElementById ("rufo");
console.log (perro.src);
</script>
```

Figura 6. Asignación de atributos.

Este fragmento de código nos generaría una salida similar a esta en la consola:

https://cdctecmilenio.mx/img/rufo.png

Un tercer método similar es **getElementsByClassName**, como este busca en el contenido de un nodo de elemento y recupera todos los elementos que tienen la cadena dada en su atributo de clase.

Explicación

Haverbeke (2018), nos indica que casi todo lo relacionado con la estructura de datos DOM se puede cambiar. La forma del árbol del documento se puede modificar cambiando las relaciones entre padres e hijos. Los nodos tienen un método de eliminación para eliminarlos de su nodo principal actual. Para agregar un nodo hijo a un nodo de elemento, podemos usar appendChild, que lo coloca al final de la lista de hijos, o insertBefore, que inserta el nodo dado como primer argumento antes del nodo dado como segundo argumento.

Un nodo puede existir en el documento en un solo lugar. Por lo tanto, si se inserta el párrafo Tres delante del párrafo Uno, primero se eliminará del final del documento y luego se insertará en la parte delantera, lo que dará como resultado Tres / Uno / Dos. Todas las operaciones que insertan un nodo en algún lugar, como efecto secundario, harán que se elimine de su posición actual (si tiene una).

El método **replaceChild** se utiliza para reemplazar un nodo hijo por otro. Toma como argumentos dos nodos: un nuevo nodo y el nodo a reemplazar. El nodo reemplazado debe ser hijo del elemento al que se llama el método. Ten en cuenta que tanto **replaceChild** como **insertBefore** esperan el nuevo nodo como primer argumento.

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>

<script>
let parrafos = document.body.getElementsByTagName("p");
document.body.insertBefore(parrafos[2], parrafos[0]);
</script>
```

Figura 7. Remplazo de nodos.

Ahora imagina que se quiere escribir un script que reemplace todas las imágenes (etiquetas <img>) en el documento con el texto contenido en sus atributos **alt** (que especifica una representación textual alternativa de la imagen). Esto implica no solo eliminar las imágenes, sino agregar un nuevo nodo de texto para reemplazarlas. Los nodos de texto se crean con el método **document.createTextNode.**

Explicación

```
<script>
function remplaceImagenes() {
    let imagenes = document.body.getElementsByTagName("img");
    for (let i = imagenes.length - 1; i >= 0; i--) {
        let imagen = imagenes[i];
        if (imagen.alt) {
            let texto = document.createTextNode(imagen.alt);
             imagen.parentNode.replaceChild(texto, imagen);
}
    }
}
</script>
```

Figura 8. Remplazo de imágenes por texto.

Dada una cadena, **createTextNode** nos da un nodo de texto que podemos insertar en el documento para que aparezca en la pantalla.

El bucle que recorre las imágenes comienza al final de la lista. Esto es necesario porque la lista de nodos devuelta por un método como **getElementsByTagName** (o una propiedad como **childNodes**) está activa. Es decir, se actualiza a medida que cambia el documento. Si comenzáramos desde el frente, eliminar la primera imagen haría que la lista perdiera su primer elemento, de modo que la segunda vez que se repite el ciclo, donde i es 1, se detendría porque la longitud de la colección ahora también es 1.

Si desea una colección sólida de nodos, en lugar de una en vivo, puede convertir la colección en una matriz real llamando a **Array.from**.

```
let arregloish = {0: "uno", 1: "dos", length: 2};
let arreglo = Array.from(arregloish);
console.log(arreglo.map(s => s.toUpperCase()));
// → Regresara ["UNO", "DOS"]
```

Figura 9. Conversión de colección en matriz.

Explicación

Para crear nodos de elementos, puede utilizar el método **document.createElement.** Este método toma un nombre de etiqueta y devuelve un nuevo nodo vacío del tipo dado.

El siguiente ejemplo define una funcion **elt**, que crea un nodo de elemento y trata el resto de sus argumentos como hijos de ese nodo. Luego, esta función se usa para agregar una atribución a una cotización.

```
<blockquote id="cita">
    Ningún libro puede terminarse jamás. Mientras trabajamos en ello aprendemos
    lo suficiente para encontrarlo inmaduro en el momento en que nos alejamos de eso.
</blockquote>

<script>
    function elt(type, ...hijos) {
let nodo = document.createElement(type);
for (let hijo of hijos) {
    if (typeof hijo != "string") nodo.appendChild(hijo);
    else nodo.appendChild(document.createTextNode(hijo));
}
return nodo;
    }

    document.getElementById("cita").appendChild(
elt("footer", "—",
    elt("strong", "Karl Popper"),
    ", prefacion de la segunda edicion de ",
    elt("em", "La Sociedad abierta y sus enemigos"),
    ", 1950"));
</script>
```

Figura 10. Creación de nodo de elementos.

Ningún libro puede terminarse jamás. Mientras trabajamos en ello aprendemos lo suficiente para encontrarlo inmaduro en el momento en que nos alejamos de eso.
—**Karl Popper**, prefacion de la segunda edicion de *La Sociedad abierta y sus enemigos*, 1950

En lo que respecta a los atributos de algunos elementos, tales como href para **enaces**, se puede acceder a ellos a través de una propiedad que tenga el mismo nombre en el objeto DOM del **elemento**. Esto se puede realizar para los atributos estándar más utilizados.

El lenguaje HTML te permite establecer cualquier atributo que desees en los nodos. Esto puede resultar útil porque facilita almacenar información adicional en un documento. Sin embargo, si crea sus propios nombres de atributos, estos no estarán presentes como propiedades en el nodo del elemento. En su lugar, debe utilizar los métodos **getAttribute** y **setAttribute** para trabajar con ellos.

Explicación

```
<script>
let parrafos = document.body.getElementsByTagName("p");
for (let parrafo of Array.from(parrafos)) {
if (parrafo.getAttribute("data-classified") == "secreto") {
parrafo.remove();
}
}
</script>
```

Figura 11. Uso del método getAttribute.

Se recomienda prefijar los nombres de dichos atributos inventados con datos para asegurarse de que no entren en conflicto con ningún otro atributo.

Existe un atributo de uso común, **class,** que es una palabra clave en el lenguaje JavaScript. Por razones históricas, algunas implementaciones antiguas de JavaScript no podían manejar nombres de propiedad que coincidieran con palabras clave, la propiedad utilizada para acceder a este atributo se llama **className.** También puedes acceder a él con su nombre real, "class", utilizando los métodos **getAttribute** y **setAttribute.**

Es posible que hayas notado que los diferentes tipos de elementos se presentan de manera distinta. Algunos, como los párrafos (<p>) o los títulos (<h1>), ocupan todo el ancho del documento y se representan en líneas separadas. Estos se denominan elementos de bloque. Otros, como los enlaces (<a>) o el elemento , se representan en la misma línea con el texto que los rodea. Estos elementos se denominan **elementos en línea.**

Para cualquier documento dado, los navegadores pueden calcular un diseño, lo que le da a cada elemento un tamaño y una posición en función de su tipo y contenido. Este diseño ***(layout)*** se utiliza para dibujar realmente el documento.

Se puede acceder al tamaño y la posición de un elemento desde JavaScript. Las propiedades **offsetWidth** y **offsetHeight** le dan el espacio que ocupa el elemento en pixeles. Un píxel es la unidad de medida básica en el navegador, tradicionalmente, corresponde al punto más pequeño que puede dibujar la pantalla, pero en las pantallas modernas, que pueden dibujar puntos muy pequeños, puede que ya no sea el caso, y un píxel del navegador puede abarcar varios puntos de visualización.

Explicación

De manera similar, **clientWidth** y **clientHeight** le dan el tamaño del espacio dentro del elemento, ignorando el ancho del borde.

```
<p style="border: 3px solid blue">
Estoy encajonado
</p>

<script>
let paragraph = document.body.getElementsByTagName("p")[0];
console.log("clientHeight:", paragraph.clientHeight);
console.log("offsetHeight:", paragraph.offsetHeight);
</script>
```

Figura 12. Obtención de características por nombre de etiqueta.

La forma más eficaz de encontrar la posición precisa de un elemento en la pantalla es el método **getBoundingClientRect**. Devuelve un objeto con propiedades superior, inferior, izquierda y derecha, lo que indica las posiciones de los pixeles de los lados del elemento en relación con la parte superior izquierda de la pantalla. Si los deseas en relación con todo el documento, debes agregar la posición de desplazamiento actual, que puedes encontrar en los enlaces pageXOffset y **pageYOffset**.

Diseñar un documento puede suponer mucho trabajo. En aras de la velocidad, los motores de los navegadores no rediseñan inmediatamente un documento cada vez que lo cambias, sino que esperan tanto tiempo como puedan. Cuando un programa JavaScript que cambió el documento termina de ejecutarse, el navegador tendrá que calcular un nuevo diseño para dibujar el documento modificado en la pantalla. Cuando un programa pregunta por la posición o el tamaño de algo, leyendo propiedades como **offsetHeight** o llamando a **getBoundingClientRect**, proporcionar información correcta también va a requerir calcular un diseño.

Un programa que alterna repetidamente entre leer la información de diseño del DOM y cambiar el DOM obliga a que se realicen muchos cálculos de diseño y, en consecuencia, se ejecutará muy lentamente. El siguiente código es un ejemplo de esto, contiene dos programas diferentes que construyen una línea de X caracteres de 2.000 piixeles de ancho y miden el tiempo que tarda cada uno.

Explicación

```
<p><span id="uno"></span></p>
<p><span id="dos"></span></p>


<script>
function tiempo(nombre, action) {
let inicio = Date.now(); // Tiempo actual en milisegundos
action();
console.log(nombre, "tomó", Date.now() - inicio, "ms");
}


tiempo("ingenuo", () => {
let objetivo = document.getElementById("uno");
while (objetivo.offsetWidth < 2000) {
objetivo.appendChild(document.createTextNode("/"));
}
});
// → Regresa: ingenuo tomó 48 ms


tiempo("inteligente", () => {
let objetivo = document.getElementById("dos");
objetivo.appendChild(document.createTextNode("/////"));
let tot = Math.ceil(2000 / (objetivo.offsetWidth / 5));
objetivo.firstChild.nodeValue = "/".repeat(tot);
});
// → Regresa: inteligente tomó 1 ms
</script>
```

Figura 13. Diseño dinámico.

Como se ha visto, los diferentes elementos HTML se dibujan de manera diferente. Algunos se muestran como bloques, otros en línea, algunos agregan estilo: <**strong**> hace que su contenido sea en **negrita** y <**a**> lo hace azul y lo subraya.

La forma en que una etiqueta <img> muestra una imagen o una etiqueta <a> hace que se siga un enlace cuando se hace clic en él, está fuertemente ligada al tipo de elemento. Pero podemos cambiar el estilo asociado con un elemento, como el color del texto o el subrayado. Aquí hay un ejemplo que usa la propiedad de estilo:

```
<p> <a href="."> Enlace normal </a> </p>
<p> <a href = "." style = "color: green"> Enlace verde </a> </p>
```

Figura 14. Cambio de estilo.

Explicación

El código anterior nos generaría algo como lo siguiente:

Enlace normal

Enlace verde

Un atributo de estilo puede contener una o más declaraciones, que son una propiedad (como el color) seguida de dos puntos y un valor (como el verde). Cuando hay más de una declaración, deben estar separadas por punto y coma, como en "color: rojo; borde: ninguno".

El estilo puede influir en muchos aspectos del documento, por ejemplo, la propiedad de visualización controla si un elemento se muestra como un bloque o un elemento en línea.

```
Este texto se muestra <strong> en línea </strong>,
<strong style = "display: block"> como un bloque </strong>, y
<strong style = "display: none"> para nada. </strong>.
```

Figura 15. Propiedad de visualización.

La salida al ejemplo anterior seria la siguiente:

Este texto se muestra **en línea,**
como un bloque
,y.

En este ejemplo la etiqueta **block** se escribirá en su propia línea, ya que los elementos del tipo **block** no se muestran en la misma línea con el texto que se encuentra fuera de la etiqueta, independientemente como esté escrito en el código html. La última etiqueta no se muestra en absoluto: **display: none** evita que un elemento aparezca en la pantalla. Esta es una forma de ocultar elementos. A menudo es preferible eliminarlos del documento por completo, ya que facilita volver a revelarlos más tarde.

Explicación

El código JavaScript puede manipular directamente el estilo de un elemento a través de la propiedad de **style** del elemento. Esta propiedad contiene un objeto que tiene todas las posibles propiedades de estilo. Los valores de estas propiedades son cadenas, en las que podemos escribir para cambiar un aspecto particular del estilo del elemento.

```
<p id="parrafo" style="color: purple">
Texto lindo
</p>

<script>
    let parrafo = document.getElementById("parrafo");
    console.log(parrafo.style.color);
    parrafo.style.color = "magenta";
</script>
```

Figura 16. Manipulación de estilo de visualización.

Con este código se obtendría este resultado:

Texto lindo

Y en la consola

purple

Algunos nombres de propiedades de estilo contienen guiones, como font-family. Debido a que es difícil trabajar con tales nombres de propiedad en JavaScript (tendría que decir style ["font-family"]), ya que ad en el objeto de estilo los nombres de propiedades tienen sus guiones eliminados y las letras después de ellos en mayúscula (style.fontFamily).

El sistema de estilo para HTML se llama CSS (hojas de estilo en cascada). Una hoja de estilo es un conjunto de reglas sobre cómo aplicar estilo a los elementos de un documento, se puede dar dentro de una etiqueta <style>.

Explicación

```
strong {
font-style: italic;
color: gray;
}
</style>
<p>Ahora <strong>el texto en negrillas</strong> está en cursivas y gris.</p>
```

Figura 17. Declaración de estilo.

Y el texto ahora se ve así:

Ahora *el texto en negrillas* está en cursivas y gris.

Las hojas de estilo en cascada se llaman así debido a que varias de estas reglas se combinan para producir el estilo final de un elemento. En el ejemplo, el estilo predeterminado para las etiquetas <**strong**>, que les da **font-weight: bold**, está superpuesto por la regla en la etiqueta <**style**>, que agrega estilo de fuente y color.

Cuando varias reglas definen un valor para la misma propiedad, la regla leída más recientemente obtiene una mayor prioridad y gana. Entonces, si la regla en la etiqueta <style> incluye font-weight: normal, contradiciendo la regla predeterminada de font-weight, el texto sería normal, no negrita. El atributo de estilo aplicado directamente al nodo tiene la mayor prioridad y siempre gana.

Es posible apuntar a cosas distintas en los nombres de etiquetas en las reglas CSS. Una regla para una clase .**abc** se aplica a todos los elementos con "**abc**" en su atributo class. Una regla para #xyz se aplica al elemento con un atributo id con valor "**xyz**" (que debe ser único dentro del documento).

```
.sutil {
color: gray;
font-size: 80%;
}


#header {
background: blue;
color: white;
}


/* los elementos p con id principal y con clases a y b */
p#principal.a.b {
margin-bottom: 20px;
}
```

Figura 18. Declaración de estilo.

Explicación

La regla de precedencia que favorece a la regla definida más recientemente se aplica solo cuando las reglas tienen la misma especificidad. La especificidad de una regla es una medida de la precisión con la que describe los elementos coincidentes, determinada por el número y tipo (etiqueta, clase o ID) de los aspectos del elemento que requiere. Por ejemplo, una regla que tiene como objetivo p.a es más específica que las reglas que tienen como objetivo p o simplemente a y, por lo tanto, tendría prioridad sobre ellas.

La notación p> a {...} aplica los estilos dados a todas las etiquetas <a> que son hijos directos de las etiquetas <p>. De manera similar, p a {...} se aplica a todas las etiquetas <a> dentro de las etiquetas <p>, ya sean secundarias directas o indirectas.

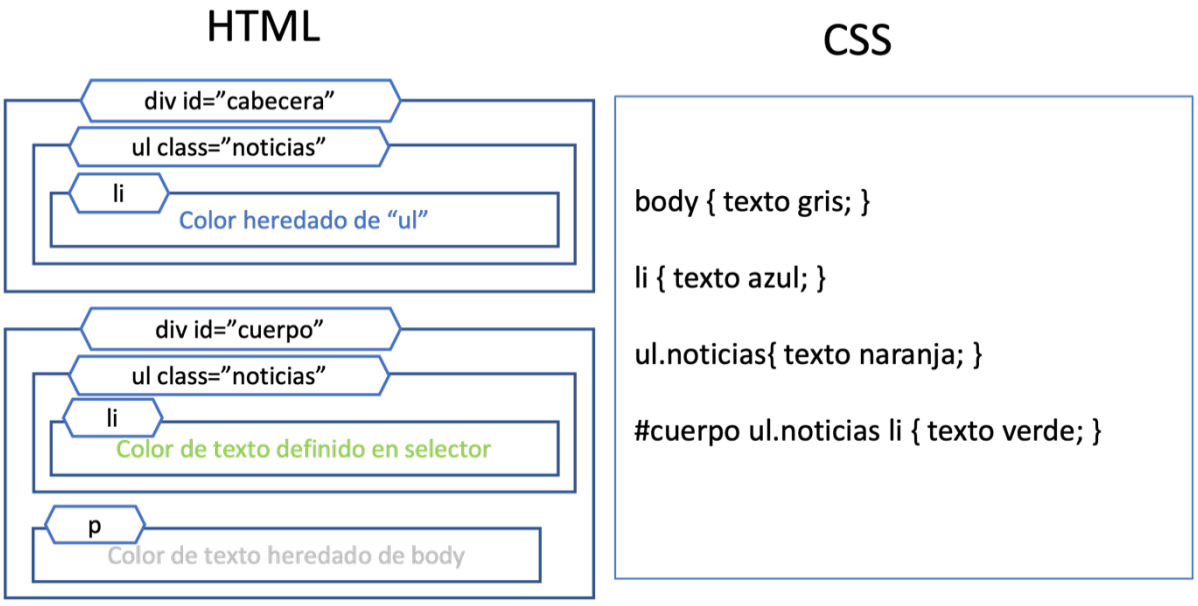


Figura 19. Reglas de precedencia CSS.

Comprender las hojas de estilo es útil cuando se programa en el navegador. La razón principal por la que se introduce la sintaxis del selector, y la notación utilizada en las hojas de estilo para determinar a qué elementos se aplica un conjunto de estilos, es que podemos usar este mismo mini lenguaje como una forma efectiva de encontrar elementos DOM.

Explicación

```
<p> Y si vas a perseguir
    <span class="animal"> conejos </span>
</p>
<p> Y sabes que te vas a caer </p>
<p> Diles un <span class="character"> fumando narguile
    <span class="animal"> oruga </span> </span> </p>
<p> Te ha llamado </p>

<script>
    function cuenta(selector) {
      return document.querySelectorAll(selector).length;
    }
    console.log(cuenta("p")); // Todos los elementos <p>
    // → 4
    console.log(cuenta(".animal")); // Clase Animal
    // → 2
    console.log(cuenta("p .animal")); // Animal dentro de <p>
    // → 2
    console.log(cuenta("p> .animal")); // Hijo directo de <p>
    // → 1
</script>
```

Figura 20. Búsqueda de elementos.

El método **querySelectorAll,** que se define tanto en el objeto del documento como en los nodos del elemento, toma una cadena de selección y devuelve una **NodeList** que contiene todos los elementos que coincide.

A diferencia de métodos como **getElementsByTagName**, el objeto devuelto por **querySelectorAll** no está activo. No cambiará cuando cambie el documento. Sin embargo, todavía no es una matriz real, por lo que aún debe llamar a Array.from si desea tratarla como tal.

El método **querySelector** (sin la parte All) funciona de manera similar. Este es útil si desea un elemento único específico, ya que devolverá solo el primer elemento coincidente o nulo cuando ningún elemento coincida.

Cierre

En este tema pudiste aprender que los programas JavaScript pueden inspeccionar e interferir con el documento que muestra el navegador a través de una estructura de datos llamada DOM. Esta estructura de datos representa el modelo del navegador del documento y con JavaScript se puede modificar para cambiar el documento visible.

Otro punto importante que revisaste es que el DOM está organizado como un árbol, en el que los elementos se ordenan jerárquicamente según la estructura del documento. Los objetos que representan elementos tienen propiedades como parentNode y childNodes, que se pueden usar para navegar por este árbol.

También aprendiste que la forma en que se muestra un documento puede verse influenciada por el estilo, tanto adjuntando estilos a los nodos directamente como definiendo reglas que coincidan con ciertos nodos. Hay muchas propiedades de estilo diferentes, como color o visualización. El código JavaScript puede manipular el estilo de un elemento directamente a través de su propiedad de estilo.

En este tema pudiste ver las siguientes características de la API DOM:

- Cómo consultar o seleccionar elementos individuales de un documento.
- Cómo atravesar un documento y cómo encontrar a los antepasados, hermanos y descendientes de cualquier elemento del documento.
- Cómo consultar y establecer los atributos de los elementos del documento.
- Cómo consultar, configurar y modificar el contenido de un documento.
- Cómo modificar la estructura de un documento creando, insertando y eliminando nodos.

¿Cómo afecta el uso de DOM el funcionamiento de las páginas Web? ¿Crees que DOM es la principal fortaleza de JavaScript?

Referencias bibliográficas

- Flanagan, D. (2020). Javascript: The Definitive Guide (7ma ed.). Estados Unidos: O'Reilly.
- Haverbeke, M. (2018). *Eloquent JavaScript* (3rd ed.). Estados Unidos: No Starch Press. · · https://eloquentjs-es.thedojo.mx/
- Kantor, I. (2021). *The JavaScript Languaje.* Recuperado de https://javascript.info/

Para saber más

Lecturas

- W3Schools. (2021). *JavaScript HTML DOM*. Recuperado de wshttps://www.w3schools.com/js/js_htmldom.asp
- MDN contributors (2021). *Document Object Model (DOM)*. Recuperado de https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Videos

- Códigos de programación – MR. (2020, 15 de julio). *Curso de JS: 11. Manipulación el DOM* parte 1 [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=RcQVT1smEFg
- Códigos de programación – MR. (2020, 21 de julio). *Curso de JS: 12. Manipulación el DOM* parte 2 [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=mXGxfXnljb0
- DesarrolloWeb.com. (2017, 8 de mayo). *Manipulación de la página: Javascript DOM* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=6MKJzOrmpQo

Checkpoints

Asegúrate de:

- Comprender correctamente el concepto de DOM para que apliques diferentes formas de darles vida a las páginas web.
- Comprender correctamente los conceptos de CSS dentro y fuera del DOM para que · puedas comenzar a darles una apariencia más profesional a tus desarrollos.
- Crear nuevos ejemplos del uso de DOM para que adquieras práctica y agilidad al momento de desarrollar.

Requerimientos técnicos

- Computadora con acceso a Internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

- Haber realizado la actividad guiada del tema 6, de preferencia durante la sesión.
- Leer detenidamente y comprender el material explicado en el tema.
- Practicar todos los ejemplos que se describen previamente en el tema.
- Revisar cada uno de los recursos adicionales que se proponen en este tema.

Tecmilenio no guarda relación alguna con las marcas mencionadas como ejemplo. Las marcas son propiedad de sus titulares conforme a la legislación aplicable, se utilizan con fines académicos y didácticos, por lo que no existen fines de lucro, relación publicitaria o de patrocinio.