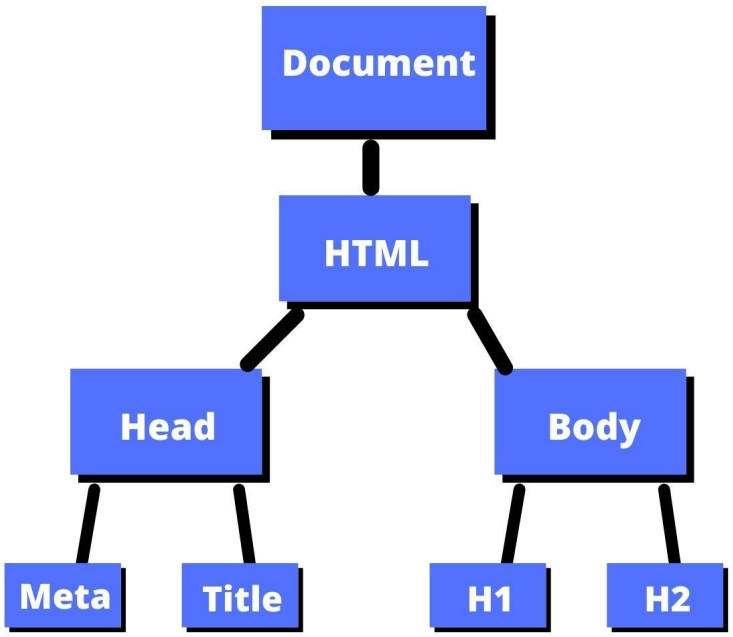
DESARROLLO EN ENTORNO CLIENTE/SERVIDOR

UD6: MODELO DE OBJETOS DEL DOCUMENTO

JavaScript



UD6 – Modelo de Objetos del Documento

1. Evaluación

El presente documento, junto con sus actividades, cubren los siguientes criterios de evaluación:

RESULTADOS DE APRENDIZAJE	CRITERIOS DE EVALUACIÓN
RA0612.3. Escribe código, identificando y aplicando las funcionalidades aportadas por los objetos predefinidos del lenguaje.	c) Se han escrito sentencias que utilicen los objetos predefinidos del lenguaje para cambiar el aspecto del navegador y el documento que contiene. d) Se han generado textos y etiquetas como resultado de la ejecución de código en el navegador. e) Se han escrito sentencias que utilicen los objetos predefinidos del lenguaje para interactuar con el usuario. h) Se ha depurado y documentado el código.
RA0612.6. Desarrolla aplicaciones web analizando y aplicando las características del modelo de objetos del documento.	a) Se ha reconocido el modelo de objetos del documento de una página web. b) Se han identificado los objetos del modelo, sus propiedades y métodos. c) Se ha creado y verificado un código que acceda a la estructura del documento. d) Se han creado nuevos elementos de la estructura y modificado elementos ya existentes. f) Se han identificado las diferencias que presenta el modelo en diferentes navegadores. g) Se han programado aplicaciones web de forma que funcionen en navegadores con diferentes implementaciones del modelo.

2. Documento Objecto Model (DOM)

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

El objeto principal del DOM es document. Este es un objeto global del lenguaje.vCada nodo HTML contenido dentro del documento es un objeto element, y estos elementos contienen otros nodos, atributos y estilo.

3. Navegando a través del DOM

A continuación se detallan las principales funciones y propiedades del objeto document para acceder a las distintas partes del documento:

- **document.documentElement** → Devuelve el elemento <html>
- **document.head** → Devuelve el elemento <head>

- **document.body** → Devuelve el elemento <body>
- **document.getElementById("id")** → Devuelve el elemento que tiene el id especifcado, o null si no existe.
- **document.getElementsByClassName("class")** → Devuelve una colección (que podemos convertir en un array de elementos) que tengan la clase especificada. Al llamar a este método desde un nodo (en lugar de document), buscará los elementos a partir de dicho nodo.
- **document.getElementsByTagName("HTML tag")** → Devuelve una colección con los elementos con la etiqueta HTML especificada. Por ejemplo "p" (párrafos).
- **element.childNodes** → Devuelve un array con los descendientes (hijos) del nodo. Esto incluye los nodos de tipo texto y comentarios.
- **element.chidren** → Igual que arriba pero excluye los comentarios y las etiquetas de texto (sólo nodos HTML). Normalmente es el recomendado.
- **element.parentNode** → Devuelve el nodo padre de un elemento.
- **element.nextSibling** → Devuelve el siguiente nodo del mismo nivel (el hermano). El método previousSibling hace justo lo opuesto. Es recomendable usar nextElementSibling o previousElementSibling si queremos obtener sólo los elementos HTML.

Una vez seleccionado un elemento, es usual cambiar el contenido del mismo, existen tres propiedades que sirven a tal efecto: innerText, textContent, innerHTML.

InnerText y **textContent** acceden al contenido de un elemento como texto, aunque hay algunas diferencias:

1. Mientras textContent lee el contenido de todos los elementos, incluyendo los elementos <script> (en-US) y <style>, innerText, no.
2. innerText también tiene en cuenta el estilo y no retornará el texto de elementos escondidos, mientras que textContent sí lo hará.

3. Como innerText tiene en cuenta el estilo CSS, escribirlo disparará un reflow, mientras que textContent no lo hará.

Diferencias con innerHTML

innerHTML retorna el HTML como su nombre indica. Con bastante frecuencia, para leer o escribir texto en un elemento, la gente usa innerHTML. textContent debería usarse en su lugar. Ya que el texto no es procesado es más probable que tenga mejor rendimiento. Además, esto evita un vector de ataques XSS (ataques XSS o Cross-Site Scripting: https://www.welivesecurity.com/la-es/2021/09/28/que-es-ataque-xss-cross-site-scripting/).

Ejemplos:

```
<!DOCTYPE html>
<html lang="es">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Uso del DOM</title>
</head>
<body>
      <div id="sele">
            Texto de <b>prueba</b>
            <script>
                  //comentario
            </script>
      </div>
    <script src="dom.js"></script>
</body>
</html>
```

Y el siguiente código JavaScript:

```
'use strict'
let miDiv=document.getElementById("sele");
console.log("innerText: "+miDiv.innerText);
console.log("innerHTML: "+miDiv.innerHTML);
console.log("textContent: "+miDiv.textContent);
```

Mostraría lo siguiente por consola:

```
innerText: Texto de prueba
innerHTML:
            Texto de <b>prueba</b>
            <script>//comentario</script>
textContent:
            Texto de prueba
            //comentario
```

4. Selector Query

Una de las principales características que JQuery introdujo cuando se lanzó (en 2006) fue la posibilidad de acceder a los elementos HTML basándose en selectores CSS (clase, id, atributos, ….). JavaScript ha implementado esta característica de forma nativa (selector query) sin la necesidad de usar jQuery:

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

Ejemplos de selectores CSS que podemos usar para encontrar elementos:

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

Ejemplo usando querySelector() y querySelectorAll():

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

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

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

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

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

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

5. Manipulando el DOM

No sólo vamos a querer seleccionar elementos ya existentes, si no también crear nuevos o eliminar otros ya existentes.

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

nuevo nodo.

6. Atributos

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

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

Ejemplo:

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

En el fichero example.js tendríamos:

```
let link = document.getElementById("toGoogle");
link.className = "specialLink"; // Equivale a: link.setAttribute("class", "specialLink");
link.setAttribute("href", "https://twitter.com");
link.textContent = "Twitter";

if(!link.hasAttribute("title")) { // Si no tenía el atributo title, establecemos uno
        link.title = "Ahora voy aTwitter!";
}
/* Imprime: <a id="toGoogle" href="https://twitter.com" class="specialLink" title="Ahora voy a Twitter!">Twitter</a> */
```

```
console.log(link);
```

7. El atributo style

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

File: example1.html

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

File: example1.js

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