LENGUAJES DE MARCAS Y SISTEMAS DE GESTIÓN DE LA INFORMACIÓN

3. DOM (Document Object Model)

Profesor: Alejandro Amat Reina

ÍNDICE

1. Introducción

La creación del Document Object Model o DOM es una de las innovaciones que más ha influido en el desarrollo de las páginas web dinámicas y de las aplicaciones web más complejas. DOM o Document Object Model es un conjunto de utilidades específicamente diseñadas para manipular documentos XML. Por extensión, DOM también se puede utilizar para manipular documentos HTML. Técnicamente, DOM es una API de funciones que se pueden utilizar para manipular las páginas HTML de forma rápida y eficiente.

Usando el DOM, la estructura del documento puede ser manipulada mediante programación.

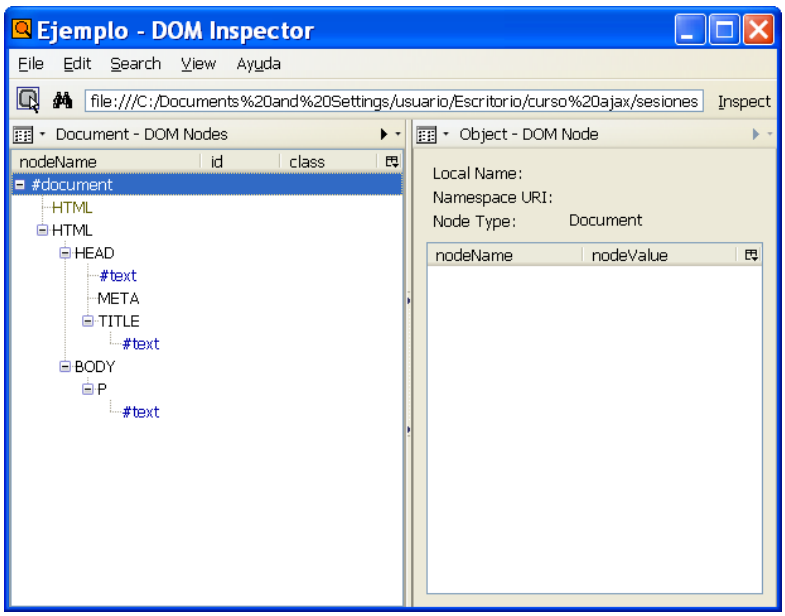
Antes de poder utilizar sus funciones, DOM transforma internamente el archivo XML original en una estructura más fácil de manejar formada por una jerarquía de nodos. De esta forma, DOM transforma el código XML en una serie de nodos interconectados en forma de árbol. El árbol generado no sólo representa los contenidos del archivo original (mediante los nodos del árbol), sino que también representa sus relaciones (mediante las ramas del árbol que conectan los nodos).

Aunque en ocasiones DOM se asocia con la programación web y con JavaScript, la API de DOM es independiente de cualquier lenguaje de programación. De hecho, DOM está disponible en la mayoría de lenguajes de programación comúnmente empleados.

La representación DOM de una página web también se estructura en forma de árbol. Dicho árbol estará compuesto de elementos o nodos, los cuales pueden contener nodos hijos dentro de ellos, y así sucesivamente. JavaScript expone el nodo principal de la página web actual a través de la variable global “document”, la cual sirve como punto de entrada para todas las manipulaciones del DOM que realicemos.

Veamos el árbol que se generaría para la siguiente página de ejemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Ejemplo</title>
</head>
<body>
   <p>Árbol de ejemplo DOM</p>
</body>
</html>
```



Representación del árbol DOM vista con el complemento de Firefox: Mozilla DOM inspector[1].

Como se puede observar, la página HTML se ha transformado en una jerarquía de nodos, en la que el nodo raíz es un nodo de tipo `#document HTML`. A partir de este nodo, existen dos nodos en el mismo nivel formados por las etiquetas `<head> y <body>`. De cada uno de los anteriores surge otro nodo (`<title> y <p>` respectivamente). Por último, de cada nodo anterior surge otro nodo de tipo texto.

Para utilizar DOM es imprescindible que la página web se haya cargado por completo, ya que de otro modo no existe el árbol de nodos y las funciones DOM no pueden funcionar correctamente.

La ventaja de emplear DOM es que permite a los programadores disponer de un control muy preciso sobre la estructura del documento HTML o XML que están manipulando. Las funciones que proporciona DOM permiten añadir, eliminar, modificar y reemplazar cualquier nodo de cualquier documento de forma sencilla.

La especificación del *DOM nivel 1* fue realizada por el W3C en 1998, podemos encontrar una *traducción al castellano de esta especificación* realizada por Juan R. Pozo.

Hace unos años algunos navegadores (especialmente Internet Explorer) no soportaban gran parte del API DOM del W3C, pero, afortunadamente, en la actualidad la mayoría de los navegadores soportan esta API prácticamente por completo, lo que ha simplificado bastante nuestra labor como programadores Javascript.

Si queremos ver el árbol DOM correspondiente a la página que tenemos abierta en nuestro navegador, podemos utilizar el inspector de elementos de las herramientas de desarrollador. En el siguiente enlace se explica el funcionamiento del mismo en Chrome: *https://developers.google.com/web/tools/chrome-devtools/dom*

[1] El complemento mostrado en la imagen es una herramienta bastante antigua que ya no está disponible, ahora disponemos del DOM inspector de las herramientas para desarrolladores tanto en Chrome como en Firefox, pero hemos utilizado esta imagen porque es muy ilustrativa de la estructura de un árbol DOM.

2. Tipos de nodos

Como ya hemos comentado anteriormente, los documentos XML y HTML tratados por DOM se convierten en una jerarquía de nodos. Los nodos que representan los documentos pueden ser de diferentes tipos. A continuación se detallan los tipos más importantes:

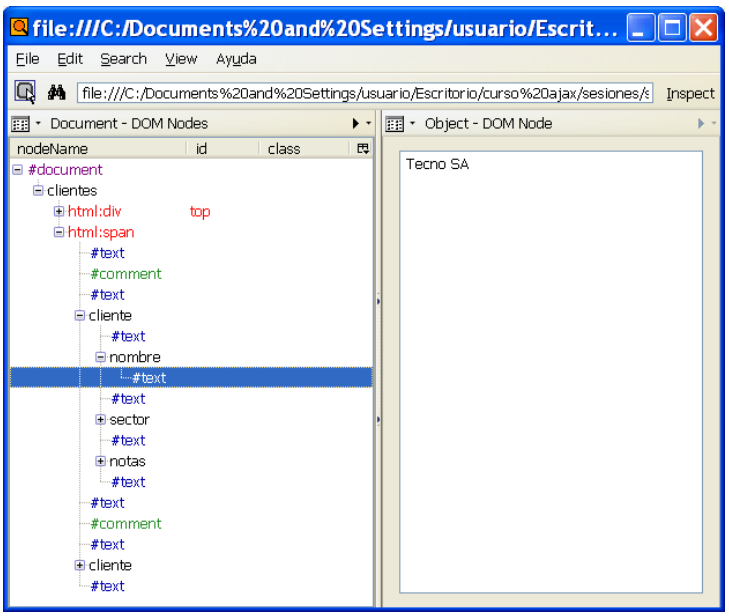
- Document: es el nodo raíz de todos los documentos HTML y XML. Todos los demás nodos derivan de él.
- DocumentType: es el nodo que contiene la representación del DTD empleado en la página (indicado mediante el DOCTYPE).
- Element: representa el contenido definido por un par de etiquetas de apertura y cierre (<etiqueta>...</etiqueta>) o de una etiqueta abreviada que se abre y se cierra a la vez (<etiqueta>). Es el único nodo que puede tener tanto nodos hijos como atributos.
- Attr: representa el par nombre-de-atributo/valor.
- Text: almacena el contenido del texto que se encuentra entre una etiqueta de apertura y una de cierre. También almacena el contenido de una sección de tipo CDATA.
- CDataSection: es el nodo que representa una sección de tipo <![CDATA[ ]]>.
- Comment: representa un comentario de XML.

Se han definido otros tipos de nodos, pero no son empleados habitualmente: DocumentFragment, Entity, EntityReference, ProcessingInstruction y Notation.

El siguiente ejemplo de documento sencillo de XML muestra algunos de los nodos más habituales:

```
<?xml version="1.0"?>
<clientes>
    <!-- El primer cliente -->
    <cliente telefono="687564345" >
        <nombre>Tecno SA</nombre>
        <sector>Tecnología</sector>
        <notas><![CDATA[Llamar la próxima semana]]></notas>
    </cliente>
    <!-- El segundo cliente -->
    <cliente>
        <nombre>Comer SA</nombre>
        <sector>Comercio</sector>
        <notas><![CDATA[Sin cita comercial]]></notas>
    </cliente>
</clientes>
```

Su representación como árbol de nodos DOM es la siguiente:



Como podemos ver todos los nodos cuelgan de un nodo raíz de tipo document. Todas las etiquetas del documento, como por ejemplo clientes, se transformarán en nodos de tipo Element.

Es importante observar que cada etiqueta simple de tipo <etiqueta>texto</etiqueta> se transforma en un par de nodos: el primero de tipo Element que contiene la etiqueta en sí, y el segundo, un nodo hijo de tipo Text que contiene el contenido definido entre la etiqueta de apertura y la de cierre.

3. La interfaz Node

Una vez que DOM ha creado de forma automática el árbol completo de nodos de la página, ya es posible utilizar sus funciones para obtener información sobre los nodos o manipular su contenido.

JavaScript crea el objeto Node para definir las propiedades y métodos necesarios para procesar y manipular los documentos.

En primer lugar, el objeto Node define las siguientes constantes para la identificación de los distintos tipos de nodos:

- `Node.ELEMENT_NODE = 1`
- `Node.ATTRIBUTE_NODE = 2`
- `Node.TEXT_NODE = 3`
- `Node.CDATA_SECTION_NODE = 4`
- `Node.ENTITY_REFERENCE_NODE = 5`
- `Node.ENTITY_NODE = 6`
- `Node.PROCESSING_INSTRUCTION_NODE = 7`
- `Node.COMMENT_NODE = 8`
- `Node.DOCUMENT_NODE = 9`
- `Node.DOCUMENT_TYPE_NODE = 10`

- `Node.DOCUMENT_FRAGMENT_NODE = 11`
- `Node.NOTATION_NODE = 12`

Además de estas constantes, Node proporciona las siguientes propiedades y métodos:

Propiedad/Método	**Valor devuelto**	**Descripción**
`nodeName`	String	El nombre del nodo (no está definido para algunos tipos de nodo)
`nodeValue`	String	El valor del nodo (no está definido para algunos tipos de nodo)
`nodeType`	Number	Una de las 12 constantes definidas anteriormente
`ownerDocument`	Document	Referencia del documento al que pertenece el nodo
`firstChild`	Node	Referencia del primer nodo de la lista childNodes
`lastChild`	Node	Referencia del último nodo de la lista childNodes
`childNodes`	NodeList	Lista de todos los nodos hijo del nodo actual
`previousSibling`	Node	Referencia del nodo hermano anterior o null si este nodo es el primer hermano. **previousElementSibling** obtendrá solo los elementos HTML
`nextSibling`	Node	Referencia del nodo hermano siguiente o null si este nodo es el último hermano. **nextElementSibling** obtendrá solo los elementos HTML
`hasChildNodes()`	Bolean	Devuelve true si el nodo actual tiene uno o más nodos hijo
`attributes`	NamedNodeMap	Se emplea con nodos de tipo Element. Contiene objetos de tipo Attr que definen todos los atributos del elemento
`appendChild(nodo)`	Node	Añade un nuevo nodo al final de la lista childNodes
`removeChild(nodo)`	Node	Elimina un nodo de la lista childNodes
`replaceChild(nuevoNodo, anteriorNodo)`	Node	Reemplaza el nodo anteriorNodo por el nodo nuevoNodo
`insertBefore(nuevoNodo, anteriorNodo)`	Node	Inserta el nodo nuevoNodo antes que la posición del nodo anteriorNodo dentro de la lista childNodes
`innerText`	String	Texto contenido en el elemento
`parentNode`	Node	Devuelve el nodo padre de un elemento
`cloneNode(deep)`	Node	Crea una copia de un nodo, si pasamos el parámetro deep a true copiará el nodo y sus descendientes, si lo pasamos a flase o no lo pasamos, sólo copiará el nodo.

Los métodos y propiedades incluidas en la tabla anterior son específicos de XML, aunque pueden aplicarse a todos los lenguajes basados en XML, como por ejemplo HTML. Para las páginas creadas con HTML, los navegadores hacen como si HTML estuviera basado en XML y lo tratan de la misma forma. No obstante, se han definido algunas extensiones y particularidades específicas para HTML.

Cuando se utiliza DOM en páginas HTML, el nodo raíz de todos los demás se define en el objeto HTMLDocument. Además, se crean objetos de tipo HTMLElement por cada nodo de tipo Element del árbol DOM. El objeto document es equivalente al objeto Document del DOM de los documentos XML, por este motivo, el objeto document también hace referencia al nodo raíz de todas las páginas HTML.

4. Acceso directo a los nodos

Podemos acceder a los nodos del árbol DOM a través del objeto document. Por ejemplo, para obtener el objeto que representa el elemento raíz de la página utilizamos la siguiente instrucción:

```
let objeto_html = document.documentElement;
```

A partir de este nodo podríamos acceder al resto de nodos del árbol utilizando los métodos de acceso a los hijos, hermanos, etc., pero como podéis imaginar esto es muy tedioso y sería imposible trabajar de esta manera. Por este motivo, DOM proporciona una serie de métodos para acceder de forma directa a los nodos deseados sin tener que recorrer todo el árbol de elementos hasta llegar a ellos. Los métodos disponibles son:

- `getElementsByTagName(etiqueta)`: obtiene un objeto de tipo HTMLCollection[2] con todos los elementos de la página HTML cuya etiqueta sea igual al nombre de etiqueta que se le pasa como parámetro.

  En el siguiente ejemplo se mostrará el atributo href de cada uno de los enlaces que contenga el primer párrafo de la página:

```
let parrafos = document.getElementsByTagName('p');
let enlaces = Array.from(parrafos[0].getElementsByTagName('a'));
enlaces.forEach(enlace => console.log(enlace.getAttribute('href')));
```

- `getElementsByName(name)`: obtiene todos los elementos de la página HTML cuyo atributo name coincida con el id que se le pasa como parámetro.
- `getElementsByClassName(className)`: obtiene todos los elementos de la página HTML cuyo atributo class coincida con el className que se le pasa como parámetro.
- `getElementById(id)`: es uno de los métodos más utilizados. Devuelve el elemento HTML cuyo atributo id coincide con el parámetro indicado. Como el atributo id debe ser único para cada elemento de una misma página (de no ser así el validador de HTML daría un error), el método devuelve únicamente el nodo deseado.

[2] HTMLCollection no es un array, pero podemos convertirlo en Array con el método Array.from

Ejemplo1.html

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

Ejemplo1.js

```
// Devuelve <li>
let firstLi = document.getElementById("firstListElement");
// Imprime "LI"
console.log(firstLi.nodeName);
// Imprime 1. (elemento -> 1, atributo -> 2, texto -> 3, comentario del nodo -> 8)
console.log(firstLi.nodeType);
// Imprime "Element 1". El primer (y único) hijo es un nodo de texto
console.log(firstLi.firstChild.nodeValue);
// Imprime "Element 1". Otra forma de obtener el contenido (texto)
console.log(firstLi.textContent);
// Itera a través de todos los elementos de la lista
let liElem = firstLi;
while(liElem !== null) {
    // Imprime el texto de dentro del elemento <li>
    console.log(liElem.innerText);
    // Va al siguiente elemento de la lista <li>
    liElem = liElem.nextElementSibling;
}
// Obtiene el elemento <ul>. Similar a parentNode.
let ulElem = firstLi.parentElement;
/* Imprime el código HTML de dentro del elemento <ul>:
<li id="firstListElement">Element 1</li>
<li>Element 2</li>
<li>Element 3</li> */
console.log(ulElem.innerHTML);
```

5. Atributos

Además del tipo de etiqueta HTML y su contenido de texto, DOM permite el acceso directo a todos los atributos de cada etiqueta.

- `element.hasAttribute(nombre)`, devuelve cierto si el elemento tiene un atributo con el nombre especificado
- `element.getAttribute(nombre)`, es equivalente a `attributes.getNamedItem(nombre)`
- `element.setAttribute(nombre, valor)` equivalente a `attributes.getNamedItem(nombre).value = valor`
- `element.removeAttribute(nombre)`, equivalente a `attributes.removeNamedItem(nombre)`

5.1. Atributos HTML en DOM

Los métodos presentados anteriormente para el acceso a los atributos de los elementos, son genéricos de XML. La versión de DOM específica para HTML incluye algunas propiedades y métodos aún más directos y sencillos para el acceso a los atributos de los elementos HTML y a sus propiedades CSS.

La principal ventaja del DOM para HTML es que todos los atributos de todos los elementos HTML se transforman en propiedades de los nodos. De esta forma, es posible acceder de forma directa a cualquier atributo de HTML. Además, algunas versiones de Internet Explorer no implementan correctamente el método setAttribute(), lo que provoca que, en ocasiones, los cambios realizados no se reflejan en la página HTML.

La única excepción que existe en esta forma de obtener el valor de los atributos HTML es el atributo `class`. Como la palabra `class` está reservada por JavaScript para su uso futuro, no es posible utilizarla para acceder al valor del atributo `class` de HTML. La solución consiste en acceder a ese atributo mediante el nombre alternativo `className`.

Ejemplo2.html

```
<!DOCTYPE>
<html>
<head>
    <title>JS Example</title>
</head>
<body>
    <p>
        <a id="toGoogle" href="https://google.es" class="normalLink">
            Google
        </a>
    </p>
<script src="ejemplo2.js"></script>
</body>
</html>
```

Ejemplo2.js

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

5.2. Propiedades CSS en DOM

El acceso a las propiedades CSS no es tan directo y sencillo como el acceso a los atributos HTML. En primer lugar, los estilos CSS se pueden aplicar de varias formas diferentes sobre un mismo elemento HTML. Si se establecen las propiedades CSS mediante el atributo style de

HTML, el acceso a las propiedades CSS se realiza a través de la propiedad style del nodo que representa a ese elemento.

Para acceder al valor de una propiedad CSS, se obtiene la referencia del nodo, se accede a su propiedad style y a continuación se indica el nombre de la propiedad CSS cuyo valor se quiere obtener. En el caso de las propiedades CSS con nombre compuesto (`font-weight`, `border-top-style`, `list-style-type`, etc.), para acceder a su valor es necesario modificar su nombre original eliminando los guiones medios y escribiendo en mayúsculas la primera letra de cada palabra que no sea la primera. Veamos los siguientes ejemplos:

```
let parrafo = document.getElementById('parrafo');
let negrita = parrafo.style.fontWeight;
let color = parrafo.style.color;
let estiloBordeSuperior = parrafo.style.borderTopStyle;
```

Sin embargo, esta propiedad sólo permite acceder al valor de las propiedades CSS establecidas directamente sobre el elemento HTML. En otras palabras, la propiedad style del nodo sólo contiene el valor de las propiedades CSS establecidas mediante el atributo style de HTML.

Por otra parte, los estilos CSS normalmente se aplican mediante reglas CSS incluidas en archivos externos. Si se utiliza la propiedad style de DOM para acceder al valor de una propiedad CSS establecida mediante una regla externa, el navegador no obtiene el valor correcto.

Para obtener el estilo exacto que un elemento tiene aplicado tendríamos que utilizar el método getComputedStyle. Puedes consultar cómo se usa en el siguiente enlace:

https://developer.mozilla.org/es/docs/Web/API/Window/getComputedStyle

Además de obtener el valor de las propiedades CSS, también es posible modificar su valor mediante javaScript. Una vez obtenida la referencia del nodo, se puede modificar el valor de cualquier propiedad CSS accediendo a ella mediante la propiedad style:

```
let parrafo = document.getElementById('parrafo');
parrafo.style.margin = '10px';
parrafo.style.color = '#CCC';
```

6. Crear, modificar y eliminar nodos

Hasta ahora, todos los métodos DOM presentados se limitan a acceder a los nodos y sus propiedades. A continuación, se muestran los métodos necesarios para la creación, modificación y eliminación de nodos dentro del árbol de nodos DOM.

Los métodos DOM disponibles para la creación de nuevos nodos son los siguientes:

Método	Descripción
`createAttribute(nombre)`	Crea un nodo de tipo atributo con el nombre indicado
`createCDataSection(texto)`	Crea una sección `CDATA` con un nodo hijo de tipo texto que contiene el valor indicado
`createComment(texto)`	Crea un nodo de tipo comentario que contiene el valor indicado
`createDocumentFragment()`	Crea un nodo de tipo `DocumentFragment`
`createElement(nombre_etiqueta)`	Crea un elemento del tipo indicado en el parámetro nombre_etiqueta
`createEntityReference(nombre)`	Crea un nodo de tipo `EntityReference`
`createProcessingInstruction(objetivo, datos)`	Crea un nodo de tipo `ProcessingInstruction`
`createTextNode(texto)`	Crea un nodo de tipo texto con el valor indicado como parámetro

Veamos un ejemplo de generación de contenido dinámico a través del DOM. Partiremos de una página Web vacía que sólo contendrá un elemento div y añadiremos un párrafo de texto dentro de este elemento. La página html será la siguiente:

```
<!DOCTYPE>
<html>
<head>
    <title>Ejemplo de generación de Nodos</title>
</head>
<body>
<div id="contenido"></div>
<script src="ejemplo1.js"></script>
</body>
</html>
```

Para añadir el contenido usaremos el siguiente código Javascript:

```
// Obtenemos el elemento correspondiente a la etiqueta div mediante su id
let contenido = document.getElementById('contenido');
// Creamos un nuevo nodo de tipo Element para el párrafo de texto
let p = document.createElement('p');
// Creamos un nuevo nodo de tipo Text para el contenido del párrafo
let texto = document.createTextNode('párrafo creado dinámicamente con el DOM');
// asociamos el nodo Text con el nodo Element
p.appendChild(texto);
// asociamos el nodo Element al nodo Element de la etiqueta div
contenido.appendChild(p);
```

También se pueden utilizar funciones DOM para eliminar cualquier nodo existente originalmente en la página y cualquier nodo creado mediante los métodos DOM. Para eliminar un nodo lo haremos a través de su elemento padre que podemos obtener mediante la propiedad parentNode, o bien, llamando directamente al método remove del elemento a eliminar. El siguiente código Javascript eliminaría el párrafo creado en el ejemplo anterior:

```
// Obtenemos el elemento correspondiente a la etiqueta div mediante su id
let contenido = document.getElementById('contenido');
// Obtenemos el primer párrafo dentro del elemento contenido
let p = contenido.getElementsByTagName('p')[0];
// Eliminamos el párrafo a través del elemento padre
p.parentNode.removeChild(p);
```

Otra posibilidad es eliminar el párrafo directamente:

```
// Obtenemos el elemento correspondiente a la etiqueta div mediante su id
let contenido = document.getElementById('contenido');
// Obtenemos el primer párrafo dentro del elemento contenido
let p = contenido.getElementsByTagName('p')[0];
// Eliminamos el párrafo a través del elemento padre
p.remove();
```

Además de crear y eliminar nodos, las funciones DOM también permiten reemplazar un nodo por otro. Utilizando la misma página HTML de ejemplo, se va a sustituir el párrafo original por otro párrafo con un contenido diferente.

```
// Obtenemos el elemento correspondiente a la etiqueta div mediante su id
let contenido = document.getElementById('contenido');
// Creamos un nuevo nodo de tipo Element para el párrafo de texto
let nuevoP = document.createElement('p');
// Creamos un nuevo nodo de tipo Text para el contenido del párrafo
let texto = document.createTextNode('párrafo que sustituye al anterior');
// asociamos el nodo Text con el nodo Element
nuevoP.appendChild(texto);
// Obtenemos el párrafo creado anteriormente
let anteriorP = contenido.getElementsByTagName('p')[0];
// Reemplazamos el párrafo anterior por el nuevo
anteriorP.parentNode.replaceChild(nuevoP, anteriorP);
```

Hemos visto que mediante el método `appendChild` podemos insertar un elemento hijo a un elemento ya existente, este nuevo elemento se insertará al final de la lista de hijos que contenga. Si queremos insertar un elemento antes de otro existente podemos utilizar el método `insertBefore()`. Veamos como haríamos esto con un ejemplo:

```
// Obtenemos el elemento correspondiente a la etiqueta div mediante su id
let contenido = document.getElementById('contenido');
// Creamos un nuevo nodo de tipo Element para el párrafo de texto
let nuevoP = document.createElement('p');
// Creamos un nuevo nodo de tipo Text para el contenido del párrafo
let texto = document.createTextNode('párrafo insertado delante del anterior');
// asociamos el nodo Text con el nodo Element
nuevoP.appendChild(texto);
// Obtenemos el párrafo creado anteriormente
let anteriorP = contenido.getElementsByTagName('p')[0];
// Insertamos el nueva párrafo antes del anterior
anteriorP.parentNode.insertBefore(nuevoP, anteriorP);
```

Una vez más, es importante recordar que las modificaciones en el árbol de nodos DOM sólo se pueden realizar cuando toda la página web se ha cargado en el navegador. El motivo es que los navegadores construyen el árbol de nodos DOM una vez que se ha cargado completamente la

página web. Cuando una página no ha terminado de cargarse, su árbol no está construido y por tanto no se pueden utilizar las funciones DOM.

Si una página realiza modificaciones automáticas (sin intervención del usuario) es importante utilizar el evento onload para llamar a las funciones de JavaScript, tal y como se verá más adelante en el tema de los eventos.

6.1. Un atajo. La propiedad innerHTML

Los ejemplos anteriores son adecuados cuando estamos creando un documento HTML que sigue una estructura muy regular que puede ser codificada como un algoritmo, pero todos los navegadores admiten también una propiedad denominada innerHTML que permite asignar contenido arbitrario a un elemento de forma muy sencilla. innerHTML es una cadena, que permite representar los hijos de un nodo como etiquetas HTML. Por ejemplo, podemos reescribir el primer ejemplo del capítulo anterior de la siguiente forma:

```
// Obtenemos el elemento correspondiente a la etiqueta div mediante su id
let contenido = document.getElementById('contenido');
// Le añadimos al elemento contenido el código HTML que nos interesa
contenido.innerHTML += '<p>Párrafo de contenido creado con innerHTML</p>';
```

Es importante observar que le hemos añadido el código HTML a la propiedad innerHTML mediante el aperador '+=', esto es así para mantener los contenidos existentes. Si hubiéramos utilizado el operador '=' habríamos sustituido todo el contenido existente por el nuestro.

7. 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,….). Esto, entre otras cosas, hizo que esta librería tuviera una gran aceptación en el mundo de los desarrolladores web en la parte de cliente. Posteriormente, se introdujeron nuevos métodos en el DOM para implementar esta característica de forma nativa (selector query) sin la necesidad de usar jQuery.

Hay dos métodos que podemos usar para obtener los elementos del DOM basándonos en selectores CSS:

- 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 parrafos <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="./example1.js"></script>
</body>
</html>
```

```
console.log(document.querySelector("#div1 a").title); // Imprime "hello world"
// ERROR: No hay un hijo inmediato dentro de <div id="div1"> el cual sea un enlace <a>
console.log(document.querySelector("#div1 > a").title);
// Imprime "hello world"
console.log(document.querySelector("#div1 > p > a").title);
// Imprime "bye world"
console.log(document.querySelector(".normalLink[title^='bye']").title);
// Imprime "hello again"
console.log(document.querySelector(".normalLink[title^='bye'] + a").title);
let elems = document.querySelectorAll(".normalLink");
// Imprime "hello world" y "bye world"
elems.forEach(function(elem) {
    console.log(elem.title);
});
// Atributo title empieza por "hello..."
let elems2 = document.querySelectorAll("a[title^='hello']");
// Imprime "hello world" y "hello again"
elems2.forEach(function(elem) {
    console.log(elem.title);
});
// Hermanos de <a title="hello world">
let elems2 = document.querySelectorAll("a[title='hello world'] ~ a");
// Imprime "bye world" y "hello again"
elems2.forEach(function(elem) {
    console.log(elem.title);
});
```

8. Motivación

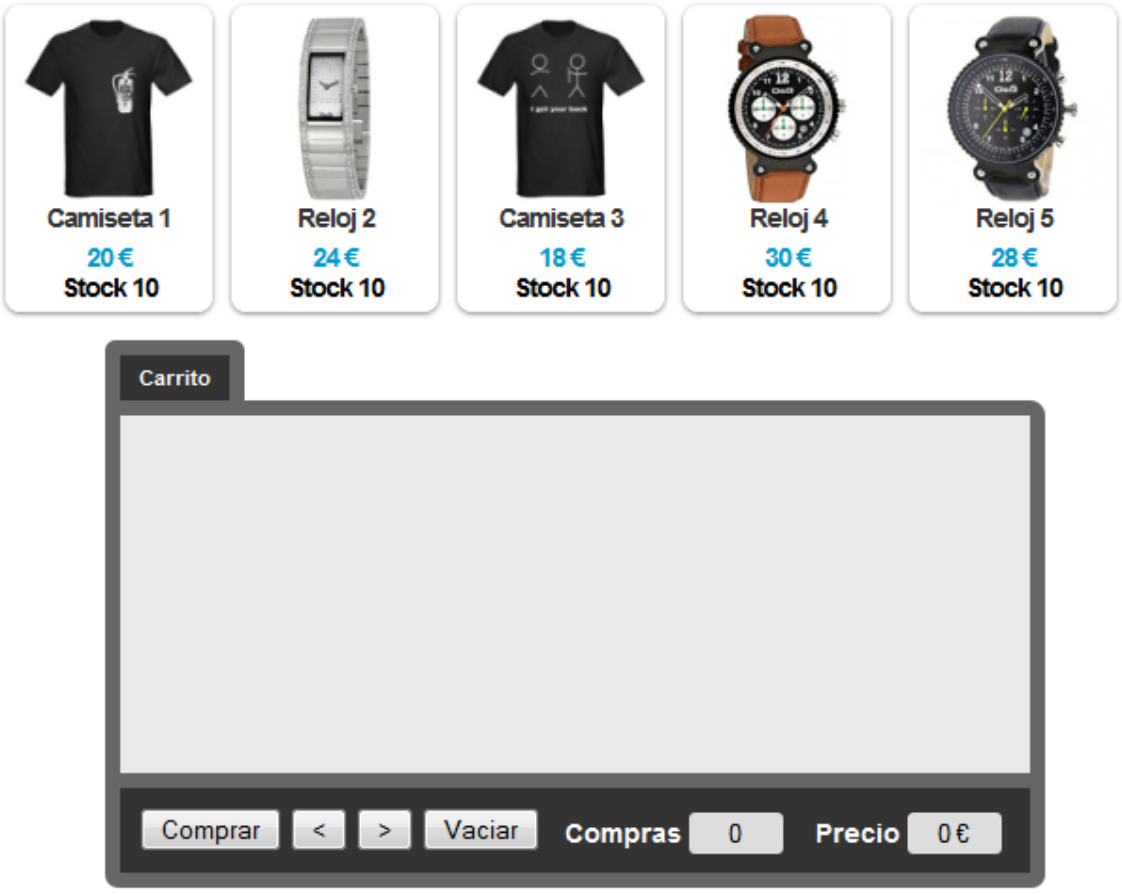
Gracias al DOM tenemos una forma estándar de cambiar los contenidos de nuestra página de forma dinámica, a través del lenguaje Javascript. Evidentemente, esto no nos sirve de nada si no somos capaces de interactuar con el usuario, es decir, que estos cambios dinámicos de la página se produzcan como respuesta a interactuaciones que realice el usuario con la misma.

Para poder realizar este tipo de funcionalidades, veremos en el siguiente tema el BOM y el modelo de eventos del navegador.

A partir de ahora, que ya tenemos una base de Javascript, realizaremos una pequeña aplicación Web. Los ejercicios realizados en un tema ampliarán los del tema anterior y servirán de base para realizar los del tema siguiente.

La aplicación Web que vamos a desarrollar es un carro de la compra, para lo cual, se proporciona un documento html (carro.html), una hoja de estilos (css/carro.css), un archivo JavaScript (script/carro.js) y una carpeta de imágenes (img). Estos ficheros se pueden encontrar dentro del archivo carrito.zip.

La apariencia que tiene la página es la siguiente:



Para poder realizar los ejercicios necesitaremos conocer la estructura interna del documento `carro.html`.

La página está dividida en dos secciones:

- La sección de los artículos, que se encuentra en el `<div>` con id `item_container`.
- La sección del carrito, que se encuentra en el `<div>` con id `cart_container`.

Dentro de la sección de artículos cada artículo se encuentra dentro de un `<div>` de la clase `ítem` cuyo id será el código del artículo. Este `<div>` tendrá dentro los siguientes elementos:

- La imagen del artículo.
- El nombre del artículo dentro de un `<label>` de la clase `title`.
- El precio del artículo dentro de un `<label>` de la clase `price`.
- El stock disponible dentro de un `<label>` de la clase `stock`.

El código html y el aspecto que tomarán los artículos será el siguiente:


```
<div class="item" id="i1">
      <img src="img/reloj1.jpg" alt="descripción i1">
      <label class="title">Reloj 1</label>
      <label class="price">20 €</label>
      <label class="stock">Stock 10</label>
</div>
```

Finalmente, antes de cerrar el `<div>` `item_container` tenemos otro `<div>` de la clase `clear`, cuya misión es terminar con los elementos flotantes del `<div>`.

Por otra parte, el carrito contiene tres `<div>`:

- El título del carrito (`cart_title`).
- El contenedor de artículos comprados (`cart_toolbar`).
- La barra de navegación (`navigate`).

Podemos ayudarnos del inspector de elementos de las herramientas de desarrollador para ir viendo cómo se muestran los distintos elementos.

Los estilos necesarios se encuentran en la hoja de estilos `carro.css`.

Inicialmente, no tenemos implementada ninguna funcionalidad, ya que el fichero `carro.js` está vacío, pero a medida que vayamos completando los temas, iremos implementando las funcionalidades necesarias para que la aplicación funcione correctamente.

9. Ejercicios

9.1. Ejercicio 1

Descarga el archivo carrito.zip del aula virtual. En él encontrarás los recursos necesarios para realizar los ejercicios.

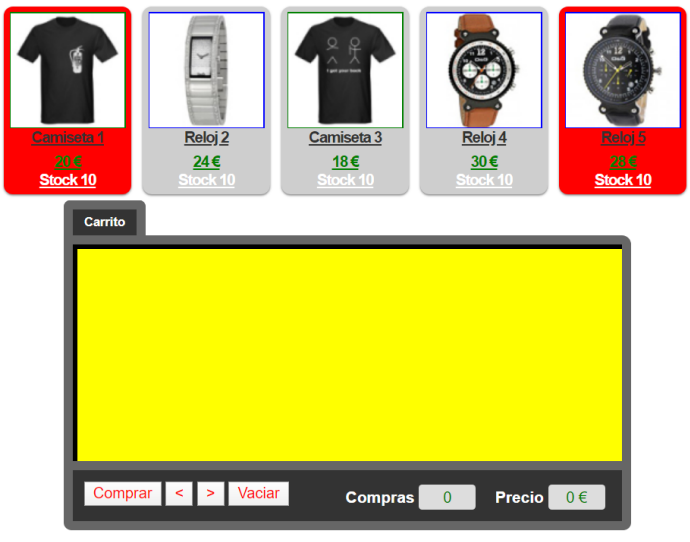
Añade al archivo carro.js un IIFE, dentro del cual, realizarás el resto de ejercicios que se piden. Crea las funciones que estimes necesarias.

Recorre los distintos elementos de la página carro.html con el inspector de elementos (esto te ayudará a ir familiarizándote con la estructura de la página).

Debes utilizar los métodos de selección de elementos vistos en el tema para resolver los siguientes ejercicios:

- Cambia a gris (#cecece) el color de fondo (background-color) de los artículos (items). Debes utilizar el método `getElementsByClassName`.
- Pon un borde (border) sólido, de color negro y 4 píxeles de grosor (4px solid black) al elemento con id cart_items. Debes utilizar el método `getElementById`.
- Pon un borde (border) sólido, de color azul y 1 píxel de grosor (1px solid blue) a todas las imágenes de la página. Debes utilizar el método `getElementsByTagName`.
- Subraya (text-decoration:underline) los <label> que sean hijos directos de un elemento de la clase item.
- Pon la fuente de color rojo (color:red) a todos los botones (<button>) que estén dentro del cart_container.
- Pon la fuente de color blanco (color:white) a todos los <label> que estén adyacentes a otro <label> y estén dentro de un elemento de la clase item.
- Pon el fondo de color amarillo a todos los que estén vacíos.
- Pon el fondo de color rojo al primer y último elemento de la clase item.
- Pon el borde de color verde (border-color:green) a las imágenes de camisetas.
- Pon la fuente de color verde (color:green) a todos los <input> de la página
- **COMPLICADO**: Pon la fuente de color verde (color:green) a todos los elementos que contengan en el texto el símbolo '€'.

Tras resolver todos los ejercicios, la página Web debe verse de la siguiente forma:



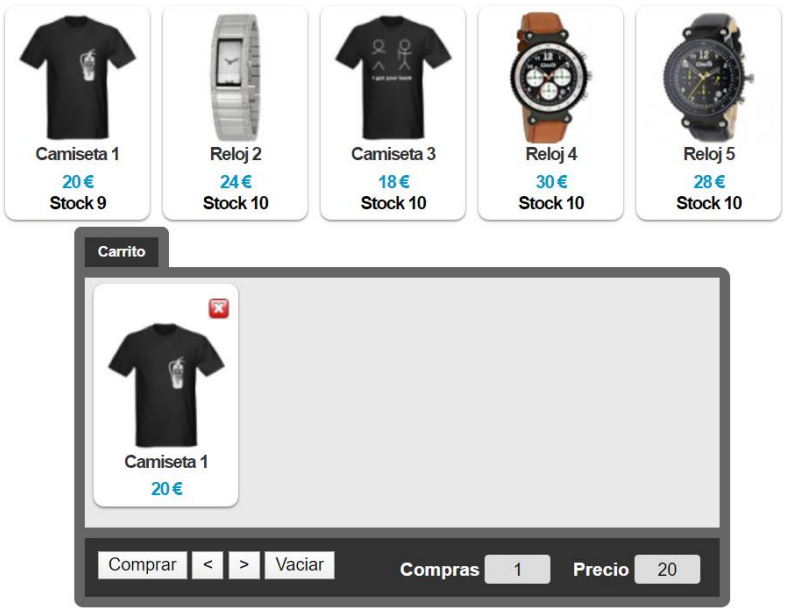
9.2. Ejercicio 2

Vuelve a descomprimir el archivo carrito.zip en una nueva carpeta. En esta nueva versión del carrito sin modificar, resuelve los siguientes ejercicios:

- Al cargar la página debe aparecer el primer producto de la lista de artículos dentro del carrito, simulando que se ha comprado un artículo camiseta 1. Para ello, sigue los siguientes pasos:
  - Añade al archivo carro.js un IIFE, dentro del cual, realizarás el resto del ejercicio.
  - Utilizando las funciones del DOM, haz una copia del artículo. Como esta copia la tienes que introducir en el carrito, para evitar tener un id duplicado añádele al atributo id una “c” delante. Construye el id de la siguiente forma: “c” + id. No puedes usar innerHTML.
  - Oculta el elemento de la clase stock (para hacer esto, puedes añadirle el estilo display: none).
  - Cambia la propiedad css cursor del elemento y de todos sus hijos al valor default.
  - Añade al principio del artículo un enlace (lo utilizaremos para eliminar el artículo del carrito). El enlace creado debe tener este código:

    ```
    <a href="" class="delete"></a>
    ```
  - Añade la copia creada al principio del contenedor de artículos comprados del carrito (elemento con id cart_items).
  - Resta 1 al stock del primer artículo de la lista.
  - Resta 1 al número de artículos disponibles (stock) del primer artículo. Si después de disminuir el stock no quedan más artículos disponibles, le añadiremos al elemento de la clase stock del artículo la clase agotado. Esto hará que el stock aparezca tachado. Comprueba esta funcionalidad poniendo 1 al stock del primer artículo antes de cargar la página.
  - Incrementa en 1 el número de artículos comprados. Para acceder al valor que contiene un input, puedes utilizar la propiedad value (es de lectura/escritura).
  - Actualiza el precio total de la compra sumándole el precio del artículo.

Tras resolver los ejercicios la página se verá de la siguiente forma:



Y si solo queda un artículo Camiseta 1, quedará así: