

6

Utilización del Modelo de Objetos del Documento *(DOM-Document Object Model)*

OBJETIVOS DEL CAPÍTULO

- ✓ Reconocer el modelo de objetos del documento de una páginas Web, identificando sus objetos, propiedades y métodos.
- ✓ Generar y verificar código que acceda a la estructura del documento y crear nuevos elementos de la estructura.
- ✓ Asociar acciones a los eventos del modelo.
- ✓ Identificar diferencias del modelo en distintos navegadores. Programar aplicaciones para que funcionen en los distintos navegadores.
- ✓ Separar las facetas, contenido, aspecto y comportamiento en aplicaciones Web.

En el capítulo anterior hemos visto que accedíamos a través de un objeto que se llamaba `document` a algunos elementos de la página web. El objeto `document` es el objeto de más alto nivel dentro del modelo de objetos del documento. ¿Qué es el modelo de objetos del documento? Es un estándar de W3C que define cómo acceder a los documentos, como por ejemplo HTML y XML. A este estándar se le llama *Document Object Model*, o DOM, que traducido significa *Modelo de Objetos del Documento*. El DOM es una interfaz de programación de aplicaciones (API) de la plataforma de W3C y cuenta con un lenguaje neutro que permite a los programas y *scripts* acceder y actualizar dinámicamente su contenido, estructura y estilo de documento.

6.1 EL MODELO DE OBJETOS DEL DOCUMENTO (DOM)

El DOM fue utilizado por primera vez con el navegador Netscape Navigator (versión 2.0 del navegador). A esta primera versión de DOM, se la considera *modelo básico* o *DOM nivel 0*. El primer navegador de Microsoft que comenzó a utilizar el modelo básico fue Internet Explorer 3.0. En la versión 3.0 de Netscape y la 4.0 de Internet Explorer se comenzaron a utilizar *rollovers*. *Rollover* es un efecto que hace cambiar una imagen, por ejemplo, cuando pasamos el ratón por encima. A partir de aquí comenzaron a incluir en algunos navegadores la capacidad de detectar eventos de ratón y de teclado. Debido a las diferencias entre los navegadores W3C se emitió una especificación a finales de 1998 que se llamó *DOM nivel 1*. En esta especificación ya se consideraban las características y manipulación de todos los elementos existentes en los archivos HTML y también de XML. A finales del año 2000, W3C emitió la especificación *DOM nivel 2*. En esta especificación se incluyó el manejo de eventos en el navegador y la capacidad de interactuar con hojas de estilo CSS. En 2004 se emitió la especificación *DOM nivel 3*, la cual utiliza la definición de tipos de documento (DTD) y la validación de documentos.

6.1.1 TIPOS DE MODELOS DOM

Actualmente DOM se divide en tres partes diferentes según la W3C. A estas partes también se les llama niveles. A continuación vamos a ver cuáles son estos tres niveles y en qué consiste cada uno de ellos:

- **Núcleo del DOM.** Este es el modelo estándar para cualquier documento estructurado. En este modelo se especifican a nivel general las pautas para definir los objetos y propiedades de cualquier documento estructurado, así como los métodos para acceder a ellos.
- **XML DOM.** Este es el modelo estándar para los documentos XML. Este modelo define los objetos y propiedades de todos los elementos XML, así como los métodos para acceder a ellos.
- **HTML DOM.** Modelo estándar para los documentos HTML. Este es el modelo en el que nos vamos a centrar. El DOM HTML es un modelo de datos estándar para HTML. Una interfaz de programación estándar para HTML independiente de la plataforma y el lenguaje. Es un estándar de la W3C. El DOM HTML define los objetos y las propiedades de los elementos HTML. También define los métodos para acceder a ellos. Por lo tanto, DOM HTML es un estándar sobre la forma de obtener, modificar, añadir o eliminar elementos HTML.

En el punto siguiente vamos a ver cómo se estructura el modelo DOM para facilitar la organización de un documento de aplicación web.

6.1.2 ESTRUCTURA DEL ÁRBOL DOM

El modelo DOM HTML se define a través de una estructura de árbol. Para poder utilizar las utilidades DOM, el navegador web transforma automáticamente todas las páginas web en una estructura de árbol. Las páginas web son una sucesión de caracteres, por lo que resultaría excesivamente complicado manipular los elementos si no fuera por esta conversión. La estructura de cada página web, se organiza de forma que se pueda acceder a los elementos a través de la estructura de árbol. DOM transforma los documentos HTML en elementos. Estos elementos se llaman nodos. A su vez los nodos están interconectados y muestran el contenido de la página web y la relación entre nodos. Por lo tanto, al conjunto de nodos se le llama, árbol de nodos. A continuación mostramos un ejemplo de una estructura de nodos generada por DOM a partir de una página HTML.

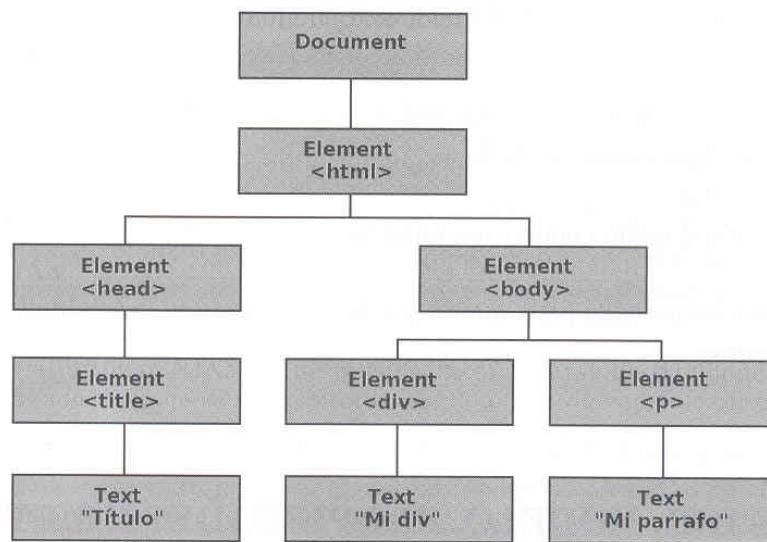


Figura 6.1. Estructura de arbol de una página HTML

Según vemos en la figura anterior, DOM crea una estructura por cada uno de los elementos de la página HTML. A estos elementos que se sitúan en el árbol se les llama nodos. Por lo tanto, el árbol de la figura anterior tendría once nodos.

En la estructura anterior el nodo `<html>` sería el nodo raíz, por tanto, todo está contenido dentro de él. La estructura en código HTML de la imagen de la Figura 6.1 sería la siguiente:

```

<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="">My link</a>
    <h1>My header</h1>
  </body>
</html>
  
```

El navegador convierte cada página de la aplicación web en una estructura de árbol. Como podemos observar los datos de cada elemento se almacenan en un nodo aparte, que cuelga del nodo elemento. Existe un nodo especial llamado documento, está en el nivel superior como hemos visto en la Figura 6.1. A partir de este nodo raíz, las etiquetas HTML se transforman en distintos tipos de nodo.

El modelo DOM define la forma en que los objetos y elementos se relacionan entre sí, tanto en el navegador como en el documento. Cada objeto tiene un nombre único. Cuando existe más de un objeto de un tipo, estos objetos se organizan en un vector.

El primer objeto que nos encontramos es el objeto `document`, este objeto es la raíz del árbol de nodos. Por lo tanto, es un objeto nodo. Este nodo al ser el primero, no tiene nodos padre, tampoco tiene nodos a su mismo nivel. Solo tiene nodos hijo. De acuerdo con el DOM, todo lo que existe en un documento HTML es un nodo. Para ordenar la estructura del árbol, existen una serie de reglas que son de obligado cumplimiento:

- En el árbol de nodos, al nodo superior (`document`, visto anteriormente) se le llama raíz.
- Cada nodo, exceptuando el nodo raíz, tiene un padre.
- Un nodo puede tener cualquier número de hijos.
- Una hoja es un nodo sin hijos.
- Los nodos que comparten el mismo parente, son hermanos.

Estas normas siempre son respetadas para que la jerarquía tenga una integridad y poder asegurar el orden y acceso a los diferentes elementos.

6.2 OBJETOS DEL MODELO. PROPIEDADES Y METODOS DE LOS OBJETOS

Una vez que DOM ha creado de forma automática el árbol completo, tenemos los nodos del árbol DOM. El estándar considera un nodo a cada una de las partes del árbol, por tanto, en una de las propiedades del nodo se especifica el tipo de nodo. Esta propiedad es `nodeType`. La veremos en los puntos ~~posterior~~ anteriores junto con el resto de propiedades. Existen doce tipos de nodos posibles, cada uno tiene unas propiedades asociadas. A continuación vamos a estudiar cuáles son los tipos de nodos que pueden existir en un árbol DOM. Los tipos de nodos indicados son específicos del lenguaje XML, no obstante estos pueden ser aplicados a los lenguajes basados en XML como, por ejemplo, XHTML. En las páginas HTML, el navegador hace como si HTML estuviera basado en XML y lo trata de la misma forma.

6.2.1 OBJETOS DEL MODELO

Según los tipos de nodos, como hemos visto en el punto anterior, las características de estos van a variar considerablemente. A continuación vamos a definir cada uno de los tipos de nodos que existen en el DOM HTML. Según la W3C, estos son los más importantes.

Tipos de nodos:

- **Document.** Es el nodo raíz del documento HTML. Todos los elementos del árbol cuelgan de él.
- **DocumentType.** Este nodo indica la representación del DTD de la página. Un DTD es una definición de tipo de documento. Define la estructura y sintaxis de un documento XML. El *DOCTYPE* es el encargado de indicar el DocumentType.
- **Element.** Este nodo representa el contenido de una pareja de etiquetas de apertura y cierre (*<etiqueta>...</etiqueta>*). También puede representar una etiqueta abreviada que se cierra a si misma (*
*). Este es el único nodo que puede tener tantos nodos hijos como atributos.
- **Attr.** Este nodo representa el nombre del atributo o valor.
- **Text.** Este nodo almacena la información que es contenida en el tipo de nodo Element.
- **CDataSection.** Este nodo representa una secuencia de código del tipo *<! [CDATA[]]>*. Este texto solo será analizado por un programa de análisis.
- **Comment.** Este nodo representa un comentario XML.

Viendo la relación que genera el estándar entre las partes de una aplicación web a nivel de código y los tipos de nodos del árbol, queda estructurado el árbol para poder manipular de una forma fácil cada uno de los elementos. En el punto siguiente vamos a definir cómo se accede al documento y a los tipos de nodos desde DOM.

6.2.2 LA INTERFAZ NODE

Para poder manipular la información de los nodos, JavaScript crea un objeto, llamado `Node`. En este objeto se definen las propiedades y los métodos para procesar y manipular los documentos. El objeto `Node` define una serie de constantes que identifican los tipos de nodo. En la siguiente tabla vemos una relación entre las constantes y un número que se asigna a cada uno de los tipos de nodo.

Tabla 6.1 Asociación de constantes valor de tipo de nodo

Tipo de nodo=Valor	
	<code>Node.ELEMENT_NODE = 1</code>
	<code>Node.ATTRIBUTE_NODE = 2</code>
	<code>Node.TEXT_NODE = 3</code>
	<code>Node.CDATA_SECTION_NODE = 4</code>
	<code>Node.ENTITY_REFERENCE_NODE = 5</code>
	<code>Node.ENTITY_NODE = 6</code>
	<code>Node.PROCESSING_INSTRUCTION_NODE = 7</code>
	<code>Node.COMMENT_NODE = 8</code>
	<code>Node.DOCUMENT_NODE = 9</code>
	<code>Node.DOCUMENT_TYPE_NODE = 10</code>
	<code>Node.DOCUMENT_FRAGMENT_NODE = 11</code>
	<code>Node.NOTATION_NODE = 12</code>

Además de estas constantes el objeto `Node` proporciona una serie de propiedades y métodos para poder acceder a la jerarquía de elementos del árbol. Estos los mostramos en la tabla a continuación:

Tabla 6.2 Métodos y propiedades de `Node`

Propiedad/Método	Valor devuelto	Descripción
<code>nodeName</code>	<code>String</code>	Nombre del nodo (no está definido para algunos tipos de nodo).
<code>nodeValue</code>	<code>String</code>	Valor del nodo (no está definido para algunos tipos de nodo).
<code>nodeType</code>	<code>Number</code>	Una de las 12 constantes definidas anteriormente.
<code>ownerDocument</code>	<code>Document</code>	Referencia del documento al que pertenece el nodo.
<code>firstChild</code>	<code>Node</code>	Referencia al primer nodo de la lista <code>childNodes</code> .
<code>lastChild</code>	<code>Node</code>	Referencia al último nodo de la lista <code>childNodes</code> .
<code>childNodes</code>	<code>NodeList</code>	Lista de todos los nodos hijo del nodo actual.
<code>parentNode</code>	<code>Node</code>	Referencia al padre del nodo hijo.
<code>previousSibling</code>	<code>Node</code>	Referencia al nodo hermano anterior o <code>null</code> si este nodo es el primer hermano.
<code>nextSibling</code>	<code>Node</code>	Referencia al nodo hermano siguiente o <code>null</code> si este nodo es el último hermano.
<code>hasChildNodes()</code>	<code>Boolean</code>	Devuelve <code>true</code> si el nodo actual tiene uno o más nodos hijo.
<code>attributes</code>	<code>NamedNodeMap</code>	Lo empleamos con nodos de tipo <code>Element</code> . Contiene objetos de tipo <code>Attr</code> que definen todos los atributos del elemento.
<code>appendChild(nodo)</code>	<code>Node</code>	Añade un nuevo nodo al final de la lista <code>childNodes</code> .
<code>removeChild(nodo)</code>	<code>Node</code>	Elimina un nodo de la lista <code>childNodes</code> .
<code>replaceChild(nuevoNodo, anteriorNodo)</code>	<code>Node</code>	Reemplaza el nodo <code>anteriorNodo</code> por el nodo <code>nuevoNodo</code> .
<code>insertBefore(nuevoNodo, anteriorNodo)</code>	<code>Node</code>	Inserta el nodo <code>nuevoNodo</code> antes de la posición del nodo <code>anteriorNodo</code> dentro de la lista <code>childNodes</code> .

Con estos métodos, podemos acceder a los diferentes nodos del árbol y también a la jerarquía. Así como crear, modificar y eliminar nodos.

6.3 ACCESO AL DOCUMENTO DESDE CÓDIGO

Cuando el árbol de nodos DOM ha sido construido por el navegador de forma automática, y se ha cargado completamente, podemos acceder a cualquier nodo. Si existe más de un tipo de un elemento, estos se van almacenando en un vector. A continuación vamos a ver la estructura de acceso a través del árbol DOM en una página HTML. Supongamos que tenemos una páginas HTML con la siguiente estructura:

```
<html>
  <head>
    <title>Titulo DOM</title>
  </head>
  <body>
    <p>Parrafo DOM</p>
    <p>Parrafo DOM segundo</p>
    <p>Parrafo DOM tres</p>
  </body>
</html>
```

Cuando la página ha sido cargada, podemos acceder a los elementos. Según la estructura de DOM, el primer paso es recuperar el objeto que representa el elemento raíz de la página. En realidad el objeto que se define como raíz es *HTMLDocument*. El objeto *document* es parte del BOM (*Browser Object Model*), pero se considera equivalente al *document* de DOM, por lo que *document* también hace referencia al nodo raíz las páginas HTML.

```
var obj_html = document.documentElement;
```

De esta forma cargamos en el objeto *obj_html* un objeto de tipo *HTMLElement* que representa al elemento *html* de nuestra estructura. De este elemento derivan siempre según la estructura DOM *<head>* y *<body>*, al saber que son dos, podríamos acceder a ellos recuperando el primer y último hijo del nodo *<html>* utilizando dos de los métodos de la tabla anterior.

```
var obj_head = obj_html.firstChild;
var obj_body = obj_html.lastChild;
```

En el caso de que existan más de dos nodos y queramos acceder a ellos, podríamos hacerlo a través del índice, el modo de acceso es a través de un vector que se genera con los objetos que están al mismo nivel. La forma de acceder es la siguiente:

```
var obj_head = obj_html.childNodes[0];
var obj_body = obj_html.childNodes[1];
```

En el caso de que no sepamos el número de nodos hijo, podemos acceder a este valor utilizando la propiedad *length* sobre el método *childNodes*.

```
var numeroHijos = obj_html.childNodes.length;
```

Otra forma de acceder a un nodo en el árbol, es a través de su hijo. Para recuperar el elemento deberíamos utilizar el método `parentNode` de la siguiente manera:

```
var obj_html = obj_body.parentNode;
```

Como dijimos en uno de los puntos anteriores, el nombre de los `element` coincide con el nombre de su etiqueta. Para acceder a los elementos, DOM permite acceder a través del nombre del elemento. Esto asigna el elemento al objeto que pretendamos. En el código que mostramos a continuación, asignamos el elemento `body` (llamado en la página HTML, `<body>...</body>`), al objeto `obj_body`.

```
var obj_body = document.body;
```

6.3.1 ACCESO A LOS TIPOS DE NODO

Los objetos del modelo, se diferencian por su tipo. Existen distintos tipos de nodo. La Tabla 6.1, muestra la relación entre la constante relacionada con el tipo de nodo y el número que se asigna al tipo. La forma de acceder al tipo de nodo, es a través de la propiedad `nodeType`. A través de la interfaz `Node`, JavaScript accede al tipo de nodo. Por lo tanto, la forma de acceder a un tipo de nodo es la siguiente:

```
obj_tipo_documento = document.nodeType; // 9  
obj_tipo_elemento = document.documentElement.nodeType; // 1
```

Las llamadas a los tipos de nodo anteriores, devuelven en el primer caso, “9” el valor asociado en la constante `Node.DOCUMENT_NODE`, asociado en la Tabla 6.1. En el segundo caso “1”, valor asociado en la constante `Node.ELEMENT_NODE` según la Tabla 6.1 de las constantes y los números asociados a los tipos. El uso de los tipos puede ser accedido a través de las constantes, que vienen definidas por defecto en la mayoría de los navegadores. Por lo tanto, podemos usar sentencias del tipo:

```
if(document.nodeType == Node.DOCUMENT_NODE) then{  
    alert("Verdadero")  
} // true  
  
if(document.documentElement.nodeType == Node.ELEMENT_NODE)  
then{  
    alert("Verdadero")  
} // true
```

Las condiciones devolverán verdadero, puesto que las constantes definidas en la Tabla 6.1, tienen los valores por defecto. En el caso de tener un navegador que no tenga las constantes definidas por defecto, habría que definirlas de forma explícita. Esto ocurre con la versión 7 y anteriores del navegador Internet Explorer.

6.3.2 ACCESO DIRECTO A LOS NODOS

Los métodos que hemos visto anteriormente, nos permiten acceder a los nodos a través de la jerarquía del árbol. Esta forma de acceso hace necesario acceder al nodo raíz de la página para llegar a cualquier otro. Al acceder a un nodo en una página HTML real, el árbol tiene infinidad de nodos. Acceder a estos nodos resulta una tarea tediosa. En el caso de que modifiquemos la estructura del árbol añadiendo, modificando o quitando nodos, podemos ocasionar problemas en el acceso a los nodos. Por esta razón DOM añade una serie de métodos, que nos permiten acceder de forma directa a los nodos. Los métodos son `getElementsByTagname()`, `getElementsByName()` y `getElementById()`. Los nombres son largos, pero tiene la ventaja de que el nombre del método es autodescriptivo sobre su función. A continuación vamos a describir en qué consiste cada uno y cuál es la forma de invocarlos en el código.

El método `getElementsByTagname()` recupera los elementos de la página HTML de la etiqueta que hayamos pasado como parámetro. Un ejemplo de cómo devuelve las etiquetas `div` es:

```
var divs = document.getElementsByTagName("div");
```

La función devuelve un vector con los nodos de tipo `div`. En realidad el vector es una lista de nodos (`nodelist`). La forma de acceder a los `div`, es:

```
var primerDiv = divs[0];
var segundDiv = divs[1];
```

Podemos recuperar todos los `div` existentes en la página con una estructura repetitiva como es el `for`. En el caso anterior la forma de recuperar los `div` de la página sería:

```
var divs = document.getElementsByTagName("div");
for(var i=0; i<div.length; i++) {
    var div = div[i];
}
```

Esta función también la podemos aplicar para recuperar otra estructura dentro de una de un tipo ya recuperado. Siguiendo con el ejemplo anterior, mostramos cómo recuperar todos los enlaces del primer `div` del `nodelist`.

```
var divs = document.getElementsByTagName("div");
var primerdiv = div[0];
var enlaces = primerdiv.getElementsByTagName("a");
```

En el código anterior recuperamos todos los enlaces contenidos en el primer `div` de la página.

El método `getElementsByName()` recupera todos los elementos de la página HTML en los que el atributo `name` coincide con el parámetro pasado a través de la función:

```
var divPrimero = document.getElementsByName("primero");

<div name="primero">...</div>
<div name="segundo">...</div>
<div>...</div>
```

En el código anterior recuperamos el `div` que tiene por `name` primero. Habitualmente el `name` es único para cada elemento. En el caso de que existan varios elementos con el mismo `name`, recuperaríamos todos. En el caso de los `input` de tipo `radio`, todos los elementos relacionados tienen el mismo `name`, en este caso la función recupera la colección de elementos. Esta función podríamos utilizarla para aplicar algo a un grupo de elementos que tengan el mismo `name`.

El método `getElementById()` recupera el elemento HTML cuyo `id` coincide con el pasado a través de la función. Esta función accede directamente al nodo a través del `id`, que se le pasa a la función. Como el `id` de cada elemento tiene que ser único, es la función más utilizada para leer y modificar sus propiedades.

```
var pie = document.getElementById("pie");

<div id="pie">
  <a href="URL" id="imagen">...</a>
</div>
```

6.3.3 ACCESO A LOS ATRIBUTOS DE UN NODO TIPO ELEMENT

En los puntos anteriores hemos accedido a las etiquetas (nodo de tipo `element`) del árbol, pero DOM permite acceder directamente a todos los atributos de una etiqueta. Para que los atributos de una etiqueta, puedan ser accedidos, estos contienen la propiedad `attributes`. Esta propiedad nos permite acceder a todos los atributos de un nodo de tipo `element`. Para ello hace uso de los siguientes métodos:

- **`getNameItem(nomAttr)`**. Devuelve el nodo de tipo `attr` (atributo), cuya propiedad `nodeName` (nombre del nodo) contenga el valor `nomAttr`.
- **`removeNameItem(nomAttr)`**. Elimina el nodo de tipo `attr` (atributo) en el que la propiedad `nodeName` (Nombre del nodo) coincide con el valor `nomAttr`.
- **`setNameItem(nodo)`**. Este método añade el nodo `attr` (atributo) a la lista de atributos del nodo `element`. Lo indexa según la propiedad `nodeName` (del atributo).
- **`item(pos)`**. Devuelve el nodo correspondiente a la posición indicada por el valor numérico `pos`.

En los métodos indicados anteriormente, el tipo de nodo es `attr`. Este tipo de nodo corresponde con los atributos que están integrados dentro de un nodo de tipo `element` (etiqueta). El nodo de tipo `attr` no devuelve directamente el valor del atributo, si no el nombre del atributo. A continuación mostramos cómo podemos utilizar los métodos anteriores:

```
<p id="parraf" style="color:blue">Prueba de texto</p>

var p = document.getElementById("parraf");

// El valor de valorId es parraf
var valorId = p.attributes.getNamedItem("id").nodeValue;
var valorId = p.attributes.item(0).nodeValue;

// El valor de valorId es parraf
p.attributes.getNamedItem("id").nodeValue= "parrafModifica";
```

El párrafo anterior ha sido modificado, en el *id* y hemos añadido un atributo nuevo de tipo *name*, por tanto queda como lo mostramos a continuación:

```
<p id="parrafModifica" style="color:blue">Prueba de texto</p>
```

Estos métodos pueden resultar algo tediosos a la hora de acceder a los tributos de un nodo, por ello DOM proporciona unos métodos que permiten acceso directo a la modificación, inserción y borrado de los atributos de una etiqueta.

■ **getAttribute(nomAtributo).**

Este método equivale a: `attributes.getNameItem(nombAtributo)`.

■ **setAttribute(nomAtributo, valorAtributo).**

Este método equivale a la estructura: `attributes.getNamedItem(nomAtributo).value = valor`.

■ **removeAttribute(nomAtributo).**

Este método equivale a la estructura: `attributes.removeNameItem(nomAtributo)`.

Con los siguientes métodos, podemos hacer las acciones del código anterior de la siguiente manera:

```
<p id="parraf" style="color: blue">Prueba de texto</p>
```

```
var p = document.getElementById("parraf");
```

```
// El valorId es parraf
```

```
var valorId = p.getAttribute("id");
```

```
// Se añade al atributo id el valor parraTexto
```

```
p.setAttribute("id", " parraTexto ");
```

Con estos métodos podemos manejar todos los atributos que tiene una etiqueta HTML.

6.3.4 CREACIÓN Y ELIMINACIÓN DE NODOS

En los puntos anteriores hemos visto como acceder a los nodos y a las propiedades de los mismos, pero DOM nos permite la creación de nodos en el árbol de forma dinámica. Para crear nodos DOM proporciona una serie de métodos. Como los tipos de nodos son diferentes, existe un método para la creación de cada tipo de nodo. A continuación vamos a ver cuáles son los métodos que nos permiten crear nodos cuando el árbol ya ha sido creado.

- **createAttribute(nomAtributo).** Este método crea un nodo de tipo *atributo* con el nombre pasado a la función.
- **createCDataSection(textoPasado).** Este método crea una sección de tipo *CDATA* con un nodo hijo de tipo *texto* con el valor *textoPasado*.
- **createComent(textoPasado).** Este método crea un nodo de tipo *coment* (*comentario*), con el contenido de *textoPasado*.
- **createDocumentFragment().** Este método crea un nodo de tipo *DocumentFragment*.

■ createElement(nomEtiqueta).

Este método crea un elemento del tipo etiqueta, del tipo del parámetro pasado como nomEtiqueta.

■ createEntityReference(nomNodo).

Este método crea un nodo de tipo EntityReference.

■ createProcessingInstruction(objetivo,dato).

Este método crea un nodo de tipo ProcessingInstruction.

■ createTextNode(textoPasado).

Este método crea un nodo de tipo texto con el valor del parámetro pasado, textoPasado.

No todos los navegadores soportan estos métodos, Internet Explorer, por ejemplo, no soporta los métodos `createCDataSection()`, `createEntityReference()` y `createProcessingInstruction()`. En realidad los métodos más utilizados en la creación de nodos son `createElement()`, `createTextNode()` y `createAttribute()`. Con estos métodos podemos crear los nodos que aparecen mayoritariamente en una página HTML.

Crear un nodo en el árbol y añadirle un valor no es algo directo. Este proceso requiere de una serie de pasos. La razón de que existan varios pasos para crear y dar valor a un nodo es que existe un nodo para el elemento y, en el caso de que este elemento tenga un valor asociado, este valor es otro nodo hijo. Por lo tanto, un nodo de tipo `element`, que tiene contenido tendrá su nodo hijo de tipo texto asociado. De esta manera, es el nodo hijo el que tiene el contenido. Supongamos que partimos de una estructura HTML como la siguiente:

```
<html>
  <head><title>Ejemplo creación de un nodo</title></head>
  <body></body>
</html>
```

A continuación vamos a mostrar cómo crear un nodo de tipo `element` en nuestra página HTML. Los pasos a seguir son:

1 Creamos un nodo nuevo de tipo elemento utilizando la función vista anteriormente. Pasamos a la función el parámetro `div` para que nuestra etiqueta sea una de tipo `div`:

```
var div = document.createElement("div");
```

2 A continuación creamos un nodo de tipo texto. A la función la hemos pasado como parámetro el texto, `Hola mundo`, que será el contenido del nodo. La etiqueta la llamaremos `texto`.

```
var texto = document.createTextNode("Hola mundo");
```

3 Una vez que tenemos creados los dos nodos, asociamos al nodo de tipo `element` el de tipo texto para que éste sea su hijo.

```
div.appendChild(texto);
```

4 En este paso tenemos los dos nodos creados, y a la etiqueta `div` asociada a un nodo hijo de tipo texto. Esta etiqueta contiene la cadena `Hola mundo`. El elemento está creado pero no tiene una posición en el árbol, por tanto, si introdujéramos estos pasos en una página HTML, la etiqueta `div` no sería visualizada por el usuario. El último paso que tenemos que realizar es asociar al árbol de estructura de la página, nuestro nuevo nodo elemento.

```
document.body.appendChild(div);
```

En el código anterior hemos introducido la etiqueta `div` como hija del elemento `body`. Una vez realizados los pasos de creación e inserción de un nodo `div` en la página HTML, la estructura de la página anterior queda de la siguiente forma:

```
<html>
<head><title> Ejemplo creación de un nodo </head>
<body>
  <div>Hola mundo</div>
</body>
</html>
```

Es importante que la creación de nodos la realicemos cuando el árbol esté construido. El navegador construye el árbol de nodos DOM cuando la página se ha cargado por completo. Si el árbol no está generado por completo, no podemos utilizar las funciones de creación de nodos. Como veremos en los puntos siguientes, existen mecanismos para detectar si la página se ha cargado completamente, el evento `onload()`, visto en el tema anterior, nos permitía, por ejemplo, comprobar si la página se había cargado por completo.

Eliminar un nodo del árbol es una tarea que requiere identificar previamente el nodo a eliminar. Podemos identificar un nodo a través de su posición en el vector de tipos de etiquetas. A continuación vamos a seleccionar el nodo tipo `div` que creamos anteriormente para eliminarlo:

```
var div = document.getElementsByTagName("div")[0];
```

Para eliminar un nodo debemos hacerlo a través de su nodo padre. Para ello, utilizaremos el método `removeChild()` que nos encontrábamos en la Tabla 6.2 de la interfaz de Node.

```
document.body.removeChild(div);
```

Vemos como accedemos al documento, luego al `body`, en la llamada al método, pasamos como parámetro el valor que nos ha devuelto antes la posición primera del vector.

Como en casos anteriores, las páginas reales tienen un árbol excesivamente complejo. Por esta razón es inviable detectar la posición en el árbol de un elemento. Las páginas, como hemos visto, las podemos modificar dinámicamente, lo que altera la estructura del árbol. En estos casos podrían producirse errores si realizáramos llamadas desde el primer nivel de la jerarquía. Para solucionar este problema podemos utilizar el método `parentNode()`, de esta forma no tenemos que acceder al nodo padre desde el primer nivel de la jerarquía del árbol. El código anterior para eliminar el nodo `div` quedaría de la siguiente manera:

```
div.parentNode.removeChild(div);
```

La inicialización del elemento `div` en una variable es igual. En la llamada al método hemos realizado la llamada sobre el nodo padre, pero subiendo un nivel en la jerarquía en vez de bajar hasta el padre desde el elemento de más alto nivel del árbol.

Modificar un nodo del árbol es una tarea que podemos utilizar para alternar o modificar la estructura de un árbol en la página HTML. Existen casos en los que puede ser conveniente remplazar un nodo por otro para conseguir una visualización diferente. En este caso utilizaremos el método `replaceChild(paramNuevo, paramOriginal)`. Para

remplazar un nodo existente por otro tendremos que realizar la siguiente implementación. Segundo el ejemplo anterior, si queremos remplazar un div por el div que teníamos sería:

```
var nuevoDiv = document.createElement("div");
var nuevoTexto = document.createTextNode("Cadena que
    sustituye a Hola mundo");

nuevoDiv.appendChild(nuevoTexto);

var originalDiv = document.getElementsByTagName("div")[0];
originalDiv.parentNode.replaceChild(nuevoDiv, originalDiv);
```

En el código anterior, creamos un nuevo elemento div con su texto correspondiente, asociamos el div al hijo de tipo texto. Introducimos en la variable originalDiv el valor actual. En la última línea, llamamos al método remplazar hijo, invocando desde el método padre. Los parámetros que pasamos al método son, en primer lugar, el nodo nuevo y, en segundo lugar, el nodo que va a ser remplazado. La estructura de la página quedaría de la siguiente manera:

```
<html>
  <head><title> Ejemplo creación de un nodo </head>
  <body>
    <div> Cadena que sustituye a Hola mundo </div>
  </body>
</html>
```

Para poder situar en el árbol, donde queremos incluir un nuevo nodo, DOM, nos permite añadir un nodo antes de otro ya existente. Esto aumenta mucho el rendimiento de la programación. Imaginemos que queremos incluir una etiqueta div, que indique que hemos introducido mal un teléfono. Pretendemos que este mensaje se visualice justo antes de la estructura tipo input que nos permite introducir el teléfono. Bastaría con añadir la etiqueta div en el caso de que el valor del teléfono que se ha introducido no coincida con el patrón establecido. A continuación vamos a ver como añadiríamos una etiqueta:

```
var divAntes = document.createElement("div");
var textoAnt = document.createTextNode("div antes que
    original");

divAntes.appendChild(textoAnt);

var divOriginal = document.getElementsByTagName("div")[0];
document.body.insertBefore(divAntes, divOriginal);
```

En el código anterior creamos un nuevo div. Inicializamos el ya existente a una variable. Una vez en variables los dos div, utilizamos el método insertar, pasando como parámetros, primero la variable que queremos que se posicione antes y después la original. De esta forma la estructura de la página quedaría de la siguiente manera:

```
<html>
  <head><title> Ejemplo creación de un nodo </head>
  <body>
    <div> div antes que original </div>
    <div> Cadena que sustituye a Hola mundo </div>
  </body>
</html>
```

A través de estos métodos, podemos modificar la estructura de la página, para conseguir un dinamismo total en el lado del cliente, consiguiendo una limpieza y estructura de código limpia y ordenada.

6.4 PROGRAMACIÓN DE EVENTOS

A lo largo de los puntos anteriores, hemos visto multitud de acciones asociadas al modelo de objetos del documento (DOM). Hemos visto como podíamos modificar la estructura del árbol, añadir nodos, eliminarlos. En una aplicación web, es el usuario el que interactúa, además de desencadenantes como el tiempo u otras razones externas a la aplicación web. Por esta razón necesitamos algo que relacione la interacción del usuario con las acciones de DOM vistas anteriormente. Para relacionar esto, utilizamos los eventos. En el capítulo anterior veíamos como un evento era capaz de capturar una acción determinada. En función de la acción, por ejemplo hacer clic con el ratón, podíamos tomar una decisión programada. El lenguaje de programación de *script* que vamos a utilizar es JavaScript.

6.4.1 CARGA DE LA PÁGINA HTML

Una condición indispensable para que se genere la estructura de árbol, es que la página se haya cargado completamente. En el caso de que la página no se haya cargado, no podemos acceder a la estructura de árbol, puesto que esta no se ha generado, o al menos no se ha generado completamente. Por lo tanto, necesitamos conocer si la página se ha cargado, antes de realizar cualquier actuación sobre la estructura del árbol. Para comprobar si la página se ha cargado completamente, disponemos del evento `onload`. Este evento está definido para el nodo de tipo `element`, `<body>`. A continuación vamos a invocar al evento en una página HTML. Nos mostrará el mensaje ("Página cargada completamente") cuando ésta se cargue completamente.

```
<html>
  <head><title>Tituto DOM</title></head>
  <body onload="alert('Página cargada completamente');">
    <p>Primer parrafo</p>
  </body>
</html>
```

El código anterior muestra el mensaje "Página cargada completamente" cuando la pagina se carga. De esta forma podemos asegurarnos de que el árbol esta completado.

6.4.2 COMPROBAR SI EL ÁRBOL DOM ESTÁ CARGADO

En DOM podemos modificar la estructura original de la página como hemos visto en los puntos anteriores. Por lo tanto, necesitamos poder saber si una vez hemos realizada una modificación en el árbol, la página se ha cargado de nuevo por completo.

A continuación vamos a comprobar a través de una función si se ha cargado la página. En el caso de que se haya cargado, al hacer clic sobre el párrafo, mostraremos el mensaje: "Se cargo la pagina correctamente".

```
<html>
  <head><title>Titulo DOM</title>
    <script>
      function cargada() {
        window.onload = "true";
        if(window.onload) {
          return true;
        }
        return false;
      }

      function pulsar(){
        if(cargada()){
          alert("Se cargo la página correctamente");
        }
      }
    </script>
  </head>
  <body>
    <p onclick="pulsar();">Primer párrafo</p>
  </body>
</html>
```

En el código anterior vemos que la función `cargada()`, comprueba sobre el objeto Window (objeto de más alto nivel, la ventana), si ocurrió el evento `onload` (el evento `onload` ocurre, cuando la página se ha cargado completamente). Cuando pulsemos con un clic sobre "Primer párrafo" se comprueba si la función `cargada()` es verdadera. En caso de ser verdadera, nos muestra el mensaje "Se cargo la página correctamente". Si no, no muestra nada.

6.4.3 ACTUAR SOBRE EL DOM AL DESENCADENARSE EVENTOS

Una de las grandes ventajas que nos aporta DOM es poder actuar sobre la estructura del árbol, de forma inmediata. Los eventos por otro lado nos permiten definir el momento en el que queremos que se realice una acción. Estas dos características combinadas, convierten la arquitectura web en el lado del cliente, en algo dinámico, rápido y versátil, para conseguir cualquier objetivo.

A continuación vamos a ver cómo podemos actuar sobre la estructura del árbol, cuando se desencadena un evento. Una etiqueta `div`, tiene un valor por defecto. Cuando pasamos el ratón por encima cambia su valor, cuando el ratón no está por encima, el `div` toma otro valor diferente.

```
<html>
  <head><title>Titulo DOM</title>
  <script>

    function ratonEncima() {
      document.getElementsByTagName("div")
        [0].childNodes[0].nodeValue="EL RATON ESTA ENCIMA";
    }

    function ratonFuera() {
      document.getElementsByTagName("div")
        [0].childNodes[0].nodeValue="NO ESTA EL RATON
ENCIMA";
    }

  </script>
</head>
<body>
  <div onmouseover="ratonEncima();
    "onmouseout="ratonFuera();">
    VALOR POR DEFECTO
  </div>
</body>
</html>
```

En el código anterior hay dos funciones que modifican el texto que mostramos en la etiqueta `div`. Estas funciones actúan directamente a través de la estructura DOM, sobre el valor de la etiqueta `div`. En la etiqueta `div` se llama a dos eventos, el primero, para que se accione cuando el ratón esta encima de la etiqueta `div`. El segundo, para que se accione cuando el ratón no se sitúa encima del texto de la etiqueta `div`. Cuando la página se carga por primera vez, el texto que aparece, es el texto por defecto.

6.5 DIFERENCIAS EN LAS IMPLEMENTACIONES DEL MODELO

Una de las principales dificultades que nos encontramos a la hora de utilizar DOM, es que no todos los navegadores hacen una misma interpretación. W3C hace unas recomendaciones y crea unos estándares, que los navegadores van adoptando, con arreglo a la capacidad que tienen en el desarrollo del navegador. Los navegadores han de respetar los estándares que existían antes y adoptar los nuevos que van saliendo. Solo de esta forma es posible que la gran cantidad de páginas web que existen y no están adaptadas a las recomendaciones W3C puedan sobrevivir entre las que si cumplen los nuevos estándares. Las altas expectativas en generar los estándares, los caminos paralelos, los intereses de las empresas que están detrás de los diferentes navegadores, hacen que esta tarea sea lenta y exija un consenso entre los principales navegadores y la asociación estandarizadora W3C.

La guerra entre los navegadores a la hora de generar sus propios estándares, ha generado muchos problemas a los programadores de páginas web. Todos los navegadores utilizan JavaScript como uno de los lenguajes de programación en el entorno cliente, pero los objetos y eventos no se comportan de la misma forma en todos ellos. Esto obliga a generar diferente código dependiendo del navegador. Otra opción es limitar el uso de una aplicación web a uno o dos navegadores concretos. Los principales navegadores Internet Explorer, Firefox, Google Crome, Opera, Safari o Nescape Navigator implementan sus navegadores intentando adaptar la interpretación del código a los estándares. Además, dentro de un mismo navegador existen distintas versiones, de tal forma que unas soportan unos mecanismos y otras no.

6.5.1 ADAPTACIONES DE CÓDIGO PARA DIFERENTES NAVEGADORES

A la hora de abordar configuraciones de código que se adapten a varios navegadores, tenemos que tener en cuenta, que esto complica la estructura de la programación. Se pueden ocasionar problemas de interpretación, complicando la actualización futura de la página y aceptando que los navegadores que no soportan los estándares tienen intención de soportarlos en el futuro y es probable que el código implementado se tenga que retocar. Aun así, por exigencias nos podemos ver forzados a realizar adaptaciones para poder dar un servicio web en diferentes navegadores. W3C con el fin de estandarizar la Web, creo un modelo de objetos único, que es el que hemos visto en los puntos anteriores. Aun así, algunos navegadores, como Internet Explorer de Microsoft, han añadido su propia extensión de DOM, por lo que se generan problemas de interoperabilidad entre los navegadores web. A continuación vamos a ver algunas de las adaptaciones que podemos realizar para que nuestra aplicación web sea compatible con posibles carencias tecnológicas del navegador.

- **Crear de forma explícita las constantes predefinidas.** En uno de los puntos anteriores hablábamos de las constantes predefinidas que tenía la interfaz `Node` del DOM. El navegador que respeta este estándar genera de forma automática una serie de constantes, asociadas a los tipos de nodo. Estas constantes asocian una cadena de texto, que describe mejor el tipo de nodo al que nos estamos refiriendo.

De esta forma no es necesario aprenderse a qué numero corresponde cada tipo de nodo. La programación así se facilita y aumenta la legibilidad del código. Un ejemplo de lectura de una constante predefinida por el navegador sería:

```
alert(Node.DOCUMENT_NODE); // Devolvería 9  
alert(Node.ELEMENT_NODE); // Devolvería 1  
alert(Node.ATTRIBUTE_NODE); // Devolvería 2
```

En los ejemplos anteriores vemos que las constantes que incluyen los tipos de nodos están definidas. En el caso de que estas constantes no estén definidas, podemos crearlas de forma explícita dentro del navegador.

```
if(typeof Node == "undefined") {  
    var Node = {  
        ELEMENT_NODE: 1,  
        ATTRIBUTE_NODE: 2,  
        TEXT_NODE: 3,  
        CDATA_SECTION_NODE: 4,  
        ENTITY_REFERENCE_NODE: 5,  
        ENTITY_NODE: 6,  
        PROCESSING_INSTRUCTION_NODE: 7,  
        COMMENT_NODE: 8,  
        DOCUMENT_NODE: 9,  
        DOCUMENT_TYPE_NODE: 10,  
        DOCUMENT_FRAGMENT_NODE: 11,  
        NOTATION_NODE: 12  
    };  
}
```

En el código anterior comprobamos si el navegador ha definido el objeto `Node`. En caso de no estar definido, creamos de forma explícita las constantes asociadas a los tipos de nodo. El navegador Internet Explorer 7, no soporta las constantes predefinidas por lo que, si vamos a programar una aplicación web, y usamos estas constantes, debemos definirlas antes.

La realidad en la actualidad, es que los cambios en las versiones de los navegadores son constantes. Por esta razón, no podemos abarcar las múltiples opciones que surgen a la hora de adaptar el código a todos los navegadores y versiones de los navegadores de una compañía. El programador es sometido a la dualidad de programar siguiendo los estándares que propone W3C, que aparentemente sería lo correcto, o seguir ciertas reglas independientes que pueden tener otros navegadores como Internet Explorer (tiene su propia especificación de DOM, aunque ha colaborado con el estándar W3C), puesto que estos navegadores son de uso mayoritario. En el primer caso puede conseguir un efecto rechazo de las páginas que sigan el estándar (al no ser soportadas por el navegador de uso mayoritario). En el segundo caso se aleja del estándar, comprometiendo su código a la voluntad de una entidad determinada.

6.6 USO DE LIBRERÍAS DE TERCEROS

Como hemos visto en el punto anterior, el desarrollo de aplicaciones web en diferentes navegadores, complica sobremanera la implementación del código. Para solucionar este problema nace *cross-browser*. *Cross-browser* es un concepto que nace con la intención de visualizar una página o aplicación web exactamente igual en todos los navegadores. Aplicando este concepto el programador web se ve obligado a unificar y asociar las utilidades que tienen comportamientos diferentes en los navegadores. Para conseguir este objetivo, surgen utilidades que nos permiten unificar los eventos y sus propiedades.

Es importante no confundir el concepto *cross-browser* con el de multinavegador. Este concepto tiene que ver con la posibilidad de que el programador pueda visualizar en diferentes navegadores una aplicación web. De esta manera puede adaptar la aplicación web al mayor número de navegadores posibles. Cabe indicar que existen varias formas de conseguir este objetivo. A continuación vamos a definir algunas de ellas.

- **Renderizar a través de una Web.** Existen páginas web que nos permiten introducir una dirección de una página web y elegir la versión del navegador con el que queremos visualizarlo. Por ejemplo, *netrenderer* nos permite visualizar una página web en las distintas versiones de Internet Explorer. Normalmente este tipo de páginas solo nos muestran una imagen del resultado, por lo que puede resultar complicado en ocasiones solucionar algunos problemas.
- **Programas para renderizar.** Existen programas que nos permiten instalar varias versiones del mismo navegador, como *multi IE* o *Internet Explorer Collection*. Estos programas dan problemas de compatibilidad de versiones con los últimos navegadores.
- **Instalar los navegadores en máquinas virtuales.** Otra opción es instalar las versiones de navegadores en máquinas virtuales, que además estén acorde con los sistemas operativos para los que hay instalables de la versión de navegador.

Para el uso de las utilidades *cross-browser* es necesario implementar funciones que habitualmente vienen definidas en librerías. A continuación vamos a mostrar cómo sería la estructura básica condicional, que comprueba el navegador utilizado y toma una decisión u otra en función de la respuesta.

```
if (window.XMLHttpRequest){  
    // código para IE7+, Firefox, Chrome, Opera, Safari  
    Xmlhttp = new XMLHttpRequest();  
}else{  
    // código para IE6, IE5  
    Xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");  
}
```

En el código anterior vemos que si la respuesta del `window.XMLHttpRequest` es verdadero, crea un objeto de un tipo. En el caso de que no sea verdadero, crea otro objeto, compatible con Internet Explorer en su versión 5 y 6.

Los pasos para generar las librerías son: crear los objetos que agrupan los objetos y métodos relacionados con los eventos. Crear un método para poder obtener el nuevo objeto evento. Este objeto agrupará los métodos relacionados. En el caso de que exista algún navegador, como Internet Explorer, que le faltan eventos, deberemos añadir al objeto evento de Internet Explorer, los nuevos eventos, para así estandarizarlo. De esta forma iremos comprobando el navegador en el que estamos. Según sea, cada caso de navegador, iremos creando objetos del tipo de navegador, para así poder dar respuesta a los eventos según el comportamiento de la versión y tipo de navegador.

ACTIVIDADES 6.1



- Realice un *script* que recorra y muestre con un `alert` de JavaScript, cada tipo de nodo del árbol de una página HTML, que previamente haya creado.
- Realice dos tablas, una con las constantes de la interfaz `node` y otra con los métodos, indicando los valores que reciben y los que devuelven.
- Haga un pequeño esquema sobre cuáles son las distintas formas de acceder a un nodo en la estructura de árbol.
- Describa en forma de puntos, cuáles son los pasos para crear y eliminar un nodo tipo `element`, que tiene a su vez un valor de texto.
- Enumere los pasos para realizar la inserción de un nuevo nodo en el árbol. Para ello hay que tener en cuenta que la página está cargada completamente.



RESUMEN DEL CAPÍTULO

El modelo de objetos del documento (DOM) es un estándar de la entidad W3C que define cómo acceder a la estructura de documentos HTML y XML. Para ello cuenta con programas de *script*, que permiten acceder y actualizar los datos de forma dinámica en el mismo navegador.

El DOM crea una estructura en árbol formada por nodos que permite organizar de una forma ordenada la complejidad de una página HTML. El acceso a la estructura se puede realizar, recorriendo el árbol o recuperando nodos de forma independiente, a partir de su identificador. Podemos recuperar vectores de un tipo de nodo y acceder a ellos a través de un índice.

Existen varios tipos de nodos que representan cada una de las partes que nos vamos a encontrar en una página HTML. Para facilitar el acceder, insertar, modificar o borrar nodos, DOM hace uso de métodos.

Los nodos pueden contener atributos, que también pueden ser creados, modificados y eliminados a través de métodos del modelo DOM.

Para actuar sobre la estructura de árbol de DOM, la página HTML tiene que estar cargada completamente.

Las adaptaciones de código para los diferentes navegadores son complejas. No todos los navegadores hacen la misma interpretación del código. Esto requiere que tengamos en cuenta en qué navegador se ejecuta el código, si queremos que sea compatible con diferentes navegadores del mercado.

El uso de los navegadores por parte de los usuarios es vital, para que en el futuro los estándares sigan un camino o cambien de rumbo.



EJERCICIOS PROPUESTOS

1. Se propone realizar una calculadora. El funcionamiento de la misma será el siguiente. Dispondrá de números del 0 al 9 y de los signos, +, -, *, /, =. Para contener cada uno de estos números y signos usaremos una etiqueta `div` (un `div` para cada uno). Existirá otra etiqueta `div` para mostrar el resultado. El funcionamiento de la calculadora será el mismo que el de la calculadora de Windows. Al pulsar en la etiqueta `div` que contenga cada número, el `div` de resultado añadirá el valor a la izquierda del que existía antes, y así sucesivamente, hasta pulsar un signo. Cuando pulsemos el signo se mostrará en la pantalla de resultado. En el siguiente paso volveremos a introducir otro número como antes, hasta que pulsemos la tecla *igual*. Al haber pulsado el *igual* se mostrará el resultado en el `div` de resultado. Se deberá incluir un `div` que nos permite inicializar la calculadora.



TEST DE CONOCIMIENTOS



1 DOM fue utilizado por primera vez por Internet Explorer.

- a) No, la primera vez lo utilizó el navegador Netscape Navigator.
- b) Sí, Internet Explorer colabora habitualmente con la W3C.
- c) El primer navegador de Microsoft que lo utilizó fue la versión 3.0.
- d) Las respuestas *a* y *c* son correctas.

2 Los nodos de tipo `element`:

- a) Tienen todo lo necesario para mostrar una etiqueta con su texto.
- b) Necesitan un nodo hijo de tipo texto para mostrar algún valor.
- c) No son nodos de tipo etiqueta.
- d) No tienen nodos hijo.

3 En una estructura de árbol de nodos:

- a) Cada nodo tiene un parente, excepto el nodo raíz.
- b) Un nodo no puede tener cualquier número de hijos.
- c) Los nodos que no comparten parente pueden ser hermanos.
- d) Las respuestas *a* y *b* son correctas.

4 El nodo raíz de un documento HTML:

- a) Se llama `DocumentType`.
- b) Se llama `Text`.
- c) Se llama `Document`.
- d) Se llama `DOM`.

5 La propiedad `nodeType` devuelve:

- a) El nombre de la constante definida por la interfaz `Node`.
- b) El número asociado al tipo de nodo.
- c) El tipo de `element` al que corresponde el nodo.
- d) El valor del nodo en ese momento.

6 Si queremos acceder al primer `div` de la estructura del árbol de una página HTML:

- a) Tenemos que recorrer el árbol desde el nodo raíz obligatoriamente.
- b) Podemos acceder a través de un vector que se genera con las etiquetas `div`.
- c) Podemos acceder a través de su identificador.
- d) Las respuestas *b* y *c* son correctas.

7 Los atributos de un nodo:

- a) Son solo atributos de ese nodo.
- b) Pueden ser creados y modificados una vez se carga la página.
- c) No pueden ser accedidos si no se identifica a su nodo.
- d) Todas las respuestas anteriores son correctas.

8 La estandarización por parte de W3C:

- a) Pretende que todos los navegadores sigan las normas del estándar.
- b) Desvincula los intereses particulares de los estándares a seguir.
- c) El éxito de sus estándares, depende en gran medida de los navegadores usados por los usuarios finales.
- d) Todas las anteriores son correctas.

- 9** Un multinavegador permite al programador:
- a) Programar sin preocuparse de que el código se vea bien en todos los navegadores.
 - b) Programar permitiendo ver en los distintos navegadores los errores que se puedan occasionar.
 - c) Desvincularse de la programación para varios navegadores.
 - d) Conseguir que un navegador interprete el código de una manera correcta.

- 10** *Cross-Browser* en el mundo de las aplicaciones web:
- a) Es un concepto.
 - b) Hace posible visualizar una aplicación web en distintos navegadores.
 - c) Unifica y asocia los comportamientos de los eventos.
 - d) Todas las anteriores son correctas.