A continuación vamos a ver cómo machacar Javascript y el *DOM* de un documento *HTML5*.

**¿Qué es el DOM?**

El *DOM* (que corresponde a las siglas de *Document Object Model*) es, básicamente, la estructura jerarquizada que sigue nuestro *.html*. Por ejemplo:

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>HTML DOM</title>**

**</head>**

**<body onclick="this.innerHTML = '<p>DONE</p>'">**

**<div id="idx" style="position: absolute; left: 50px">**

**<p style="color: red">CLICK ME!</p>**

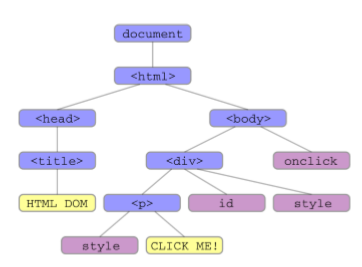
**</div>**

**</body>**

**</html>**

Hemos cogido este ejemplo que viene en la práctica para dar a entender que los tabulados que le hemos metido acá a la bostada de código no es más que una manera de que el usuario estructure las etiquetas (pues hay algunas que contienen a otras, otras que no, blablabla).

Si vamos colocando todas las etiquetas y otros elementos (vamos, todo lo que arriba está en colorines) en un orden descendente que se corresponde en el código a ir de dentro para fuera de la contención de etiquetas (por así decirlo) tenemos esta cosa de aquí:



Y aquí es donde empieza el calvario con *Javascript*, pues resulta que cada elemento que estamos viendo en este árbol puede ser representado, moldeado, aplastado y cambado (?) mediante un script.

Miremos el siguiente ejemplo para ver con algo más de claridad a qué me refiero:

**<script>**

**document.title = 'DOM Programming Interface'; document.getElementsByTagName("p")[0].innerHTML = document.body.nodeName; var x = document.getElementById("idx");**

**x.innerHTML = '<h1>New text</h1>' + x.innerHTML;**

**</script>**

Analicemos algunos aspectos importantes que tenemos en este código:

1. **.title** y **.body** aluden al título y al cuerpo (respectivamente) de nuestro documento. Con *.title* se puede manejar todo lo referido al nombre de nuestro coso, mientras que con *.body* estamos haciendo referencia a todos los elementos contenidos por la etiqueta *<body>*.
2. **.getElementByTagName(***etiqueta***)** básicamente devuelve un array de elementos, que son aquellos que estén contenidos dentro de la etiqueta que se ha pasado como parámetro.
3. **.getElementById(***nombre***)** lo que hace es devolver el elemento al que está asociado ese nombre. ¿Se acuerdan cuando en la otra práctica (la primera) hacíamos esto con el canvas para poder trabajar con él? Pues tal cual.

Ahora pasemos a hablar de cómo conectar el *DOM* con el *CSS* (se viene más *javascript*, muchachos).

Como era de esperar, el *DOM* nos puede permitir hacer cambios en el estilo que llevan los elementos de archivo *HTML* (es decir, su *CSS*, que recordemos que es el que se encarga en verdad de que todo luzca rebonito).

**<script>**

**var w = document.getElementById("idx");**

**w.style.left = '200px';**

**w.style.backgroundColor = 'lightgreen';**

**var y = document.getElementsByTagName("p")[0];**

**y.style.color = 'purple';**

**var z = document.getElementsByTagName("h1")[0];**

**z.style.fontFamily = 'courier';**

**</script>**

Como se puede observar, se tienen unos cuantos métodos que buscan cambiar el *style*. Todos siguen una estructura similar de *variable****.style.****(nombre de variable)*.

Y ahora sí que sí, va a tocar hablar en mayor profundidad de *Javascript*.

**Javascript**

De primeras vamos a responder a una serie de cuestiones básicas de sintaxis en *Javascript*, para luego enfocarnos más en la aplicación a *HTML5*.

**¿Cómo se define una variable?**

pues de la siguiente manera:

**var nombre\_variable=loquesea;**

*Var* soporta lo que sea que le eches: *int*, *string*, … La cosa es que todo se engloba en *var* (un poco como hacía *Python* con la ausencia de la declaración de tipos).

**¿Cómo van los operadores?**

== → igualdad.

= → asignación.

!= → desigualdad.

|| → *Or*.

&& → *And*.

Y poco más creo.

**¿Cómo van los condicionales (if y switch)?**

Esto va tal y como se usaba en *C#*:

**if (**expresión**){**

(acá metemos las acciones a realizar)

**} else {**

(más acciones a realizar)

**}**

Por otra parte, *switch* tiene una estructura algo más compleja pero bastante útil:

**switch (** /\*expresión\*/ **) {**

**case** /\*expresión 1\*/ **:**

(metemos lo que sea)

**break;**

**case** /\*expression 2\*/ **:**

(también metemos lo que sea)

**break;**

**...**

**default:**

(aquí toca meter lo que debe hacer el programa por defecto)

**}**

**¿Y los bucles?**

Pues hay 3, aunque como siempre solamente gastaremos los dos típicos porque el *do while* es para trozos de mierda (?).

For

**var variableEjemplo;**

**for (variableEjemplo = 0; variableEjemplo < 5; variableEjemplo++){**

(ejecutamos lo que sea, en este caso 5 veces en las cuales

variableEjemplo toma valores del 0 al 4)

**};**

While

**while (**condición**){**

(ejecutamos algo)

**};**

Do...While

**do {**

(lo que sea)

**} while (i<5);**

**¿Cómo van los métodos/funciones?**

Aquí debemos presentar un nuevo concepto: *arguments*. Éste no es más que una forma de llamar de manera genérica a los elementos que el usuario ha pasado al método en la cabecera. Aquellos argumentos que no se vean reflejados en el return de manera explícita o que queremos trabajar con ellos de manera específica sí que los tomaremos como parámetros de entrada separados. Por el resto, creo que no les será muy complicado comprender la estructura, pues es la misma que se sigue en, por ejemplo, *C#*.

Como es complicado de explicar, mejor veamos un ejemplo que eche un poco de luz al tema:

**function concatenar(separador){**

**var returneo =””;**

**for (var i = 1; i < arguments.length; i++)**

**resultado += arguments[i]+separador;**

**return resultado;**

**};**

Vale, pongámonos en situación: tenemos una serie de cadenas que queremos pegar las unas con las otras para guardar todo en una sola. Es decir, a partir de *“uno”, “dos”, “tres”, “catorce”* queremos acabar con un *string* bien bonito que ponga ***“uno, dos, tres, catorce”***.

Entonces hacemos esa función que hemos escrito arriba.

Como pueden observar, lo único que se ha pasado como parámetro de manera explícita es el separador, pues no necesitamos en ningún momento trabajar con nada relacionado con el mismo (más allá de usarlo para concatenar, pero como a lo largo del proceso éste no cambia para nada pues meh, no nos interesa entenderlo como *argument*s).

¿Entonces que es *arguments*? pues el resto de parámetros que a la hora de crear la función entendemos que no es necesario ponerlos uno a uno (pues no sabes ni siquiera cuántos van a ser, por ejemplo).

Así, podremos trabajar con el número de ellos (*arguments.length*, que por supuesto no cuenta a *separador*) e ir haciendo mil polladas que se reflejarán en el resultado que devuelva el *return*.

Para que se entienda un poco mejor, miren la llamada de la función, cómo se muestra:

**concatenar(“,”, “uno”, “dos”, “tres”, “catorce”);**

Por el momento dejemos este aspecto acá. Si en otras prácticas es necesario matizar algo más acerca de las funciones de *Javascript* ya miraremos :)))

**¿Cómo funcionan las clases en *Javascript*?**

La estructura básica que sigue es la siguiente (no les va a sonar a chino, que al final *Javascript* es el hermano tonto de *C#*):

**class caramelo{**

**constructor(nombre,marca){**

**this.nombre=nombre;**

**this.marca=marca;**

**}**

**}**

Como se puede observar, tenemos la principal (que es el **class**) y adentro tenemos un constructor, que sigue una estructura no muy diferente a la de *C#*.

Algo muy importante que deben tener en cuenta: a diferencia de otros lenguajes de programación, *JavaScript* no permite declarar un objeto antes de haber declarado la clase. Es decir, no puedes poner:

**var nuevoObjeto= new Caramelo ();**

**class Caramelo{** . . . **}**

En todo caso será al revés, ya que si no en la lectura del programa habrá un error de referencia. Entonces, ¿qué hacemos para evitar posibles problemas? pues poner todas las clases bien juntas arriba del todo (antes del *main*, por así decirlo). De esa forma no nos tendremos que preocupar de estas cosas :-)

Dentro del cuerpo de la clase (lo que está entre corchetes) no solamente está el constructor. Como ya sabemos, también pueden haber (y de hecho es lo suyo) métodos para los objetos.

Su constitución no tiene mucha ciencia, la verdad:

**class Caramelo{**

**.**

**.**

**.**

**Funcion(){**

**return 0;**

**}**

**.**

**.**

**.**

**}**

También podemos ser unos trozos de mierda y declarar la clase a la vez que el objeto porque equisdé:

**var clash = {shield:undefined, mjolnir:undefined};**

**clash.timer = undefined;**

*Nótese que el .timer es otro atributo de clash.*

En caso de querer seguir siendo unos trozos de mierda (sé que me repito, pero no deja de ser una putada de formato ya que uno no lo ve de primeras) se podrían ir declarando las funciones de la siguiente manera:

**clash.initWeapons = function () {**

**var sh = document.getElementById("capshield");**

**clash.shield = new Weapon(sh);**

**var mj = document.getElementById("mjolnir");**

**clash.mjolnir = new Weapon(mj);**

**};**

Atentos al function, que es importantísimo para indicar que se trata de una función del objeto que hemos creado.

**¿Qué hay de los arrays?**

Los arrays se crean de una forma bastante intuitiva:

**var colors = ["cyan","green","gray","blue"];**

Y bueno, creo que en cuanto a lo más básico, no queda nada por tocar. Mejor pasemos ya a mirar más cosas relacionadas con *Javascript* en *HTML5*.

Como ya sabemos, para ejecutar código en *HTML5* solemos tirar de la etiqueta *<script>*, la cual puede usarse tanto en el *<body>* como en el *<head>*, aunque es más recomendado en el primero, pues va mucho más rápido a la hora de ser entendido por el navegador (además de que jopé, tiene más sentido meter la tralla donde van todas las cosas tochas).

Ahora bien, de la misma forma que hacíamos con los *styles* (que lo metíamos todo en un archivo *CSS* y a la mierda), acá también prima fleje el poder reutilizar el código. ¿Cómo lo conseguimos esta vez? Pues mediante un archivo de extensión *.js* (se crea de la misma forma que hicimos con el *CSS*). Para hacer una llamada a este documento en el *.html* deberemos añadir un *src* a la etiqueta *script*:

**<script src="js/shieldhammer.js"></script>**

Y ya estaría, carajo.

Ahora bien, *JavaScript* dispone de algunos objetos específicos para navegadores,como ocurre con **window**, que (como deja entrever su nombre) se trata de la ventana del navegador. También está **document** que es… bueno, el documento (valga la redundancia) con el que estamos trabajando.

Algo muy importante: en el caso de **window**, siempre tenemos que inicializar la función ***entryPoint***, que contiene la acciones iniciales con las que parte el programa. La línea de código es la siguiente:

**window.onload = entryPoint;**

Para el objeto **window** también tenemos la función ***setInterval*,** qué básicamente indica la realización de una acción determinada cada x tiempo,el cual se especifica en los parámetros de la misma. Por ejemplo:

**<button onclick="myFunction()">Try it</button>**

**<script>**

**function myFunction() {**

**setInterval(function(){ alert("Hello"); }, 3000);**

**}**

**</script>**

Analicemos esto: De primeras, fuera del *script*, hemos definido un botón que será el que nos servirá de ejemplo. Lo importante es que hemos creado un evento llamado *myFunction()* que solamente se activará en caso de que se realice click.

En el *script* lo que estamos haciendo es, mediante el *setInterval*, programar *myFunction()* (que la hemos metido dentro del *script* renombrándola como *function*) de tal forma que muestre *“Hello”* cada 3 segundos. La medida con la que se mide de forma estándar el intervalo de tiempo en el que se repetirá la acción es el milisegundo, de tal forma que deberemos ajustarlo en base a lo que necesitemos en nuestro programa.

**¿Qué es el *prototype*?**

El *prototype* es el objeto por defecto. Por el momento no nos es de gran utilidad más allá de ser un elemento que nos facilita reutilizar el código… pero bueno, que eso no viene a cuento ahora mismo, en verdad.

**Creando nodos en el DOM**

Volvamos al concepto de *DOM*. Como ya sabemos, éste hace referencia a una estructura jerarquizada en forma de árbol. Cuando trabajamos en *HTML5* nosotros vamos poniendo etiquetas dentro de otras etiquetas y blablabla para conformar nuestro *DOM*. ¿Qué ocurre? pues que se puede hacer de forma más eficiente mediante el uso de *JavaScript*, creando los nodos que unen todos los elementos entre sí. Veamos un ejemplo para que lo entendamos.

Partimos de esto de aquí:

**<div id=”battlefield”>**

**<div>O</div>**

**</div>**

Como se puede observar, tenemos una etiqueta identificada como *battlefield* que agrupa a su vez a una *div* en la que tenemos la letra O.

¿Cómo se crearía la estructura de nodos de este cacho de código en *HTML5*? Pues de la siguiente manera:

**var bg=document.getElementById("battlefield");**

**var div1 = document.createElement("div");**

**bg.appendChild(div1);**

**var letterO = document.createTextNode("O");**

**div1.appendChild(letterO);**

* En la primera línea, como de costumbre, lo que hemos hecho ha sido coger el ID de nuestra etiqueta y meterla en una nueva variable para así poder trabajar.
* En la segunda nos encontramos ante una nueva función para el objeto *document*: **createElement**, que tiene como parámetro el nombre de un nuevo elemento.
* En la línea 3 ocurre algo muy importante: se enlaza este nuevo elemento con *battlefield*, que pasa a ser su padre (mientras que *div* cumple el rol de hijo).
* La función *createTextNode* nos permite crear un nuevo elemento de tipo *string*.
* Finalmente, en la última línea hacemos lo mismo que hicimos antes: unimos *letterO* para que sea hijo de *div1*.

**Canvas.**

En lo relativo a la práctica 1 ya habíamos visto algo de *canvas*, que no era más que una ventana gráfica que nos permitía hacer cosillas de animación y tal. Ahora iremos un paso más allá y veremos cómo extrapolar todo esto a *JavaScript* (somos monotemáticos, muchachos).

Para trabajar con el canvas, lo primero que va a tocar es inicializarlo, que eso lo haremos en el *.html* tal y como habíamos visto en otras ocasiones. Aunque ojo,esta vez le pondremos un identificador. Veamos el ejemplo *weaboo* del profesor:

**<div id="battlefield">**

**<canvas width="800" height="600" id="scene">**

**</canvas>**

**</div>**

Dentro del *<div>* (que ya habíamos visto antes) hemos creado un objeto de tipo *canvas* y le hemos llamado *scene*.

Si ahora vamos al *.js*, nos toparemos con que debemos hacer una función (la cual hemos colocado como método del objeto *clash*) que inicialice nuestros dibujines (es decir, las imágenes que vamos a emplear). Como en este ejemplo gastamos el martillo de Thor y el escudo del capitán americanocaucásico, el método se ha llamado *initWeapons*… pero vamos, que eso en verdad suda, lo que nos interesa es el interior:

**clash.initWeapons = function () {**

**clash.cv = document.getElementById("scene");**

**var ctx = clash.cv.getContext("2d");**

**var sh = "images/CaptainAmericaShield.png";**

**clash.shield = new Weapon(ctx,sh,0,260);**

**var mj = "images/ThorMjolnir.png";**

**clash.mjolnir = new Weapon(ctx,mj,651,250);**

**};**

¿En qué nos vamos a fijar?

* De primeras, en que hemos creado un nuevo atributo a *clash* que será nuestro canvas (es *cv*).
* Es también muy importante crear el contexto, que será el que nos permita determinar si lo que tenemos requiere de un formato 2D (relájense porque este cachito de código se explica más adelante).
* También tenemos las imágenes, que están colocadas en dos variables. Mi consejo es que no se compliquen la vida y pongan las fotos en la misma carpeta que el archivo *.html*, porque si no va a ser harto coñazo (ya que debes colocar la ruta de la foto referenciada, tal y como se ve en el ejemplo).

Ahora veamos la función que se menciona arriba (la dichosa *weapon*) así como unas cuantas más que… bueno, son solamente una excusa para que veamos mejor cómo se produce un desplazamiento en este coso.

**function Weapon(c2d,filename,l,t) {**

**this.canvas2d = c2d;**

**this.image = new Image();**

**this.image.myparent = this;**

**this.image.src = filename;**

**this.image.onload = function () {**

**var m = this.myparent;**

**m.canvas2d.drawImage(this,l,t);**

**m.width = this.naturalWidth;**

**};**

**this.left = l;**

**this.top = t;**

**this.width;**

**this.clear = function () {**

**var l = this.left, t = this.top;**

**var w = this.image.naturalWidth;**

**var h = this.image.naturalHeight;**

**this.canvas2d.clearRect(l,t,w,h);**

**};**

**this.move = function (n) {**

**this.left += n;**

**var i = this.image;**

**var l = this.left;**

**var t = this.top;**

**this.canvas2d.drawImage(i,l,t);**

**};**

**};**

No nos asustemos y vayamos por partes. La primera función de arriba es la famosa *Weapon* que tanto nombrábamos antes. Sus parámetros son, respectivamente, una movida que hemos extraído del canvas (“2d” en nuestro caso), la imagen que usamos, el largo y el ancho. Atentos, porque creamos un nuevo objeto del tipo *image*, que nos servirá para cargar nuestra foto. Este objeto carga la imagen mediante el atributo ***src*** (sí, es un equivalente al que veíamos en *HTML5*). Finalmente creamos una función *onload* que haga exactamente lo mismo que el *onload* de *HTML5* (osea, cargar la imagen por pantalla). ¿Cómo lo hacemos? Pues mediante un método que tiene nuestro *canvas* cuando está en *JavaScript*, que es **drawImage()**, cuyos parámetros son la foto y sus medidas. Ya por último se especifica que el tamaño que tendrá la foto será la correspondiente al original (de ahí que pona *naturalWidth*).

La función *clear* se encarga, como su nombre indica, de limpiar la pantalla (para que así no nos topemos con un porrón de repeticiones de la misma imagen). Lo realmente importante de esta función es que llama a ***clearRect()***, que se encarga de borrar lo dibujado con anterioridad. También atento a que *this* dispone de unos valores de *left, right* y *top*, que evidentemente hacen referencia a la posición del “lápiz” (por decirlo de alguna forma) en el *canvas*.

Y de acá pasamos a la función **move**, que se encarga de ir moviendo los muñequitos mediante la alteración de los valores de posición que hemos mencionado antes (como le estamos sumando el parámetro de entrada *n* al valor de la posición en el eje X, entonces basta con ir añadiendo valores positivos o negativos en función de qué objeto sea con tal de que se produzca el choque que buscamos).

**Eventos y su manejo.**

Un evento es, básicamente, una acción que ocurre en caso de que se cumplan unos requisitos determinados (como que se pulse un botón, por ejemplo). Los eventos en nuestro caso serán los más habituales: *load, click, mouseover, keypress,...*

Para poder manejar estas cosas, tal y como he comentado en otra ocasión, tendremos que hacer uso de la siguiente línea, que será como un *padre nuestro* (o algo así):

**window.onload = entryPoint;**

Nuestro *canvas* dispondrá de dos funciones principales para manejar la adición o eliminación de eventos, que se introducirán en la declaración de la función *entryPoint* (porque, en efecto, tenemos que crearlo nosotros): hablamos de ***addEventListener*** y ***removeEventListener***.

Realmente tampoco vamos a entrar ahora en mucho detalle, más que nada porque en la siguiente práctica se ve mucho mejor este aspecto. De la misma forma que ocurrió con el *canvas* en la primera, que quede esto como una especie de introducción :-P