INTRODUCCIÓN A JAVASCRIPT

1. Conceptos básicos

© Francisco Pérez
ies_paco@yahoo.es

CONTENIDO

1. Introducción a JavaScript 1
2. Cómo incluir JavaScript en un documento (X)HTML 1
   2.1. La etiqueta <noscript> 2
3. Sintaxis 2
4. Elementos del lenguaje 3
   4.1. Declaración de variables. 3
   4.2. Constantes 4
   4.3. Tipos de datos. 5
      4.3.1. Objetos 6
      4.3.2. Arrays 6
   4.4. Operadores 9
   4.5. Instrucciones de control de flujo 10
   4.6. Funciones 10
   4.7. Alcance de las variables 12
5. Funciones integradas 14
6. El DOM 14
   6.1. Funciones para el acceso a los nodos del DOM 16
   6.2. Los objetos nodo del DOM 17
   6.3. Inserción, modificación y eliminación de nodos del DOM 19
7. Eventos 22
   7.1. El identificador ***this*** 25
   7.2. Cómo obtener información del evento 26
8. El BOM 33
   8.1. El objeto ***window*** 33
   8.2. El objeto ***document*** 34
   8.3. El objeto ***location*** 35
   8.4. El objeto ***navigator*** 36
   8.5 El objeto ***screen*** 37
9. Validación de formularios 39
10. Orientación a objetos. Arrays. Clases nativas. Métodos y propiedades. 39
11. JQuery 39

1. Introducción a JavaScript

Javascript es un lenguaje de programación interpretado que permite crear pequeños programas o ***scripts*** que se incluyen en las páginas web. Los scripts permiten añadir cierto grado de interacción y dinamismo al documento web, como la aplicación de efectos sobre el texto, la realización de animaciones sencillas y la ejecución de determinadas tareas al pulsar sobre botones.

No obstante, JavaScript no es un lenguaje de programación completo y se diseñó con ciertas limitaciones para impedir la ejecución de scripts maliciosos. De este modo, los scripts de JavaScript no pueden comunicarse con recursos que no pertenezcan al mismo dominio desde el que se descargó el script. Un script tampoco puede acceder al sistema de entrada/salida y no puede realizar acciones como acceso a disco, conexiones de red o acceso a una base de datos, entre otros.

Además de las limitaciones anteriores, JavaScript plantea dos grandes inconvenientes:

- El soporte por parte de los navegadores no ha sido homogéneo, de tal forma que algunas funciones son reconocidas por algunos navegadores y otras no.

- No es un lenguaje "elegante" desde el punto de vista formal: no tiene un tipado fuerte de datos, y es sólo parcialmente orientado a objetos, lo que hace que en muchos casos el código JavaScript sea confuso.

Por otro lado, al ser un lenguaje interpretado, no es necesario compilar los scripts para poder ejecutarlos, pudiendo probarse directamente en el navegador, sin necesidad de procesos intermedios de compilación.

Para finalizar, debe quedar claro que JavaScript es un lenguaje de distinto de Java. Mientras que Java es un lenguaje completo, orientado a objetos, con tipado fuerte de datos, JavaScript no cumple ninguno de esos requisitos. Las únicas similitudes entre los dos lenguajes se dan en la sintaxis de algunas instrucciones.

2. Cómo incluir JavaScript en un documento (X)HTML

La integración de un script en un documento (X)HTML plantea alternativas similares a la inclusión de hojas de estilo:

- Definir el script en una etiqueta HTML, asociando la ejecución del código JavaScript a un evento:

```
 […]
<input type="button" value="haz click" onclick="alert('has pulsado el botón');" />
 […]
```

  Este método plantea varios inconvenientes. El más evidente de ellos es que si el script aumenta su complejidad, el código de la página se vuelve ilegible.

- Definir el script en el propio documento (X)HTML, dentro de en una etiqueta <script>. Aunque es correcto incluir el script en cualquier parte del documento, lo más habitual es hacerlo en dentro de la etiqueta <head> de manera que los scripts se carguen antes que el contenido de la página. De este modo, cuando el usuario active el código del script, éste ya estará cargado y disponible. Si se invoca a un script antes de que éste se haya cargado, es posible que el navegador muestre notifique algún error de secuencia de comandos.

```
[…]
<script type=”text/javascript”>
       alert(“Hola mundo”);
</script>
[…]
```

- Definir el script en un documento externo. Esta suele ser la opción más adecuada en la mayoría de escenarios ya que el script puede ser utilizado por múltiples documentos:

```
[…]
<script type=”text/javascript” src=”ficheroConScripts.js”></script>
[…]
```

Para que la etiqueta anterior funcione correctamente, debe incluirse la etiqueta de cierre separadamente (</script>) no pudiendo utilizarse la etiqueta autocerrada.

2.1. La etiqueta <noscript>

Aunque un script tiene un acceso muy limitado al equipo del usuario, todos los navegadores permiten desactivar la ejecución de scripts como medida de seguridad. En los casos en los que una página requiere JavaScript para funcionar correctamente, es posible utilizar la etiqueta <noscript> para mostrar un mensaje al usuario cuando su navegador no puede ejecutar JavaScript:

```
[…]
<body>
       <noscript>
              <p>Bienvenido a Mi Sitio</p>
              <p>La página que estás viendo requiere para su funcionamiento
              el uso de JavaScript. Si lo has deshabilitado
              intencionadamente, por favor vuelve a activarlo.</p>
       </noscript>
</body>
[…]
```

3. Sintaxis

La sintaxis de JavaScript es muy similar a la de otros lenguajes de programación como Java y C. Las normas básicas que definen la sintaxis de JavaScript son las siguientes:

- Se distingue entre mayúsculas y minúsculas. Si en una página XHTML se utilizan indistintamente mayúsculas y minúsculas, la página se visualiza correctamente

(aunque no sea correcto), siendo el único problema la no validación de la página. En cambio, si en JavaScript se intercambian mayúsculas y minúsculas el script no funciona.

- No se define el tipo de las variables: al crear una variable, no es necesario indicar el tipo de dato que almacenará. De esta forma, una misma variable puede almacenar diferentes tipos de datos durante la ejecución del script.

- No es necesario terminar cada sentencia con el carácter de punto y coma (;). En la mayoría de lenguajes de programación, es obligatorio terminar cada sentencia con el carácter ";". Aunque JavaScript no obliga a hacerlo, es conveniente seguir la tradición de terminar cada sentencia con el carácter del punto y coma (;).

- Se pueden incluir comentarios: los comentarios se utilizan para añadir información en el código fuente del programa. Aunque el contenido de los comentarios no se visualiza por pantalla, si que se envía al navegador del usuario junto con el resto del script, por lo que es necesario extremar las precauciones sobre la información incluida en los comentarios. Al igual que en Java, se pueden insertar comentarios de una sola línea, precedidos de la secuencia "//" o bien comentarios de más de una línea, que comienzan con la secuencia "/*" y terminan con "*/"

4. Elementos del lenguaje

4.1. Declaración de variables.

En el contexto de los lenguajes de programación, una variable se comporta igual que una variable matemática: actúa como un contenedor que almacena los datos con los que trabaja el programa. JavaScript es un lenguaje de tipado débil. Esto quiere decir que a diferencia de la mayoría de lenguajes, no es necesario declarar antes de su uso ni las variables ni el tipo de dato que contienen.

Además, sólo existen unos pocos tipos de datos predefinidos y las variables pueden cambiar en tiempo de ejecución el tipo de dato que contienen. Esta característica, que puede parecer una ventaja, plantea en realidad un cúmulo de inconvenientes ya que hace difícil encontrar errores derivados de una conversión de tipo inadvertida, al tiempo que reduce el rendimiento puesto que el intérprete debe deducir en tiempo de ejecución cuál es el tipo de dato que contienen las variables.

Si se quiere declarar una variable de forma explícita (lo que es opcional como ya se ha comentado, aunque muy recomendable en cualquier caso) se debe utilizar la palabra reservada "***var***" que permite declarar una o varias variables simultáneamente.

```
// declara e inicializa la variable numero1 con valor igual 7.
var numero1 = 7;

// declara la variable numero2. Inicialmente tiene el valor 'undefined'.
var numero2;

// Inicializa la variable numero3 con el valor 5. Nótese que esta variable no
// ha sido declarada previamente con la palabra reservada var.
numero3 = 5;

// declara dos variables en la misma instrucción.
var numero4, numero5;

// declara e inicializa la variable cadena que contiene una cadena de texto.
var cadena = "hola mundo";
```

El nombre de las variables (llamado ***identificador***) en JavaScript debe cumplir los siguientes requisitos:

- Sólo puede estar formado por letras, números y los símbolos "$" y "_".
- El primer carácter no puede ser un número.

4.2. Constantes

Una constante es un dato cuyo valor no cambia durante la ejecución del programa. Algunos navegadores admiten la palabra clave **const** para declarar constantes, pero su utilización no está muy extendida y al utilizarla se corre el riesgo de que el script no funcione correctamente en navegadores que no soportan esta instrucción. Su utilización sería similar a la instrucción **var** con la diferencia de que **const** declara e inicializa constantes. Para los navegadores que no soportan la instrucción **const** es necesario emplear var, teniendo la precaución de que dicha variable no debe ser sobreescrita.

Además de las constantes que pueda definir el programador en su código fuente, en JavaScript se han definido una serie de constantes primitivas:

- **undefined** o **null**: es el valor que tiene asignado una variable que no está inicializada o un identificador que no hace referencia a ningún elemento.

- **NaN**: esta constante es el acrónimo de "Not A Number" ("no es un número") y suele ser el resultado de ciertas operaciones que deberían producir un resultado numérico pero dicho resultado no puede calcularse; por ejemplo cuando se intenta obtener el cociente 0 / 0 el resultado será NaN.

- **Infinity, -Infinity**: esta constante representa el valor matemático "infinito"; se obtiene, por ejemplo al dividir entre cero. Además la constante se comporta a todos los efectos como el valor matemático infinito por lo que una operación como 30/Infinity dará como resultado cero y 0 multiplicado por Infinity dará como resultado NaN.

4.3. Tipos de datos.

Al igual que su hermano mayor (Java), JavaScript clasifica los tipos de datos con los que trabajan los programas en dos grandes grupos: los **tipos de datos por valor** y los **tipos de datos por referencia**. Estas son las diferencias entre ambos:

- Cuando se crea una variable que almacena un tipo de dato por valor, el intérprete reserva en la memoria un espacio adecuado para almacenar dicho valor y asigna a dicha posición de memoria el nombre de la variable (su identificador). Cada vez que en el código del programa se hace referencia al identificador de la variable, se está accediendo al valor almacenado en esa posición de memoria.

- Cuando se crea una variable que almacena un tipo de dato por referencia, el intérprete reserva espacio para una referencia (un puntero) que apunta a otra estructura en memoria. Las variables que almacenan tipos por referencia son, por tanto, referencias a otras estructuras complejas llamadas **objetos**.

Aunque en la declaración de variables no se especifica el tipo de dato que almacena la variable, el intérprete de JavaScript deduce por el contenido de la variable el tipo al que pertenece (en un proceso denominado "*inferencia de tipos*") y le asigna internamente uno de los siguientes tipos, denominados **tipos primitivos por valor:**

- **Tipo numérico entero (integer)**: las variables de este tipo se utilizan para almacenar valores numéricos enteros (llamados *integer* en inglés).

- **Tipo numérico decimal (float)**: las variables de este tipo se utilizan para almacenar valores numéricos con decimales (llamados valores de punto flotante o *floating point*). Para escribir un valor de este tipo, se separa la parte entera de la parte decimal con un punto.

- **Cadena de texto (string)**: las variables de este tipo almacenan caracteres, palabras y/o frases. Los valores literales de cadena de texto van siempre delimitados por comillas dobles ("). Para poder incluir caracteres especiales dentro de una cadena de texto debe anteponerse una contrabarra (\) al carácter especial. Por ejemplo:

    - Para incluir una nueva línea se debe insertar la secuencia \n
    - Para incluir un tabulador se debe insertar la secuencia \t
    - Para incluir una comilla simple se debe insertar la secuencia \'
    - Para incluir una comilla doble se debe insertar la secuencia \"
    - Para incluir una nueva línea se debe insertar la secuencia \\

- **Tipo Booleano (boolean)**: este tipo almacena un valor especial llamado *valor lógico*. El valor lógico sólo puede ser verdadero (***true***) o falso (***false***).

De acuerdo con lo visto hasta ahora, si queremos que una variable almacene un tipo de dato por valor basta con declarar dicha variable mediante la instrucción var y asignarle un valor. El intérprete de JavaScript determinará en tiempo de ejecución cuál es el tipo de dato que almacena esa variable.

4.3.1. Objetos

Los tipos de datos por valor son prácticos y fáciles de usar ya que representan un único valor (un número, una cadena de texto o un valor booleano). Sin embargo, en muchas ocasiones resulta más potente y útil utilizar un tipo de dato más complejo que agrupa varios valores y además define operaciones específicas para trabajar con esos valores. Estas estructuras más complejas reciben el nombre de **objetos**.

Cuando se trabaja con objetos, no se crea una variable de tipo objeto sino que se crea una variable que almacena una referencia (una especie de puntero) que apunta al objeto creado en memoria. De este modo, cada vez que se necesita trabajar con un objeto, es necesario crear el objeto y la variable que contiene la referencia al objeto recién creado.

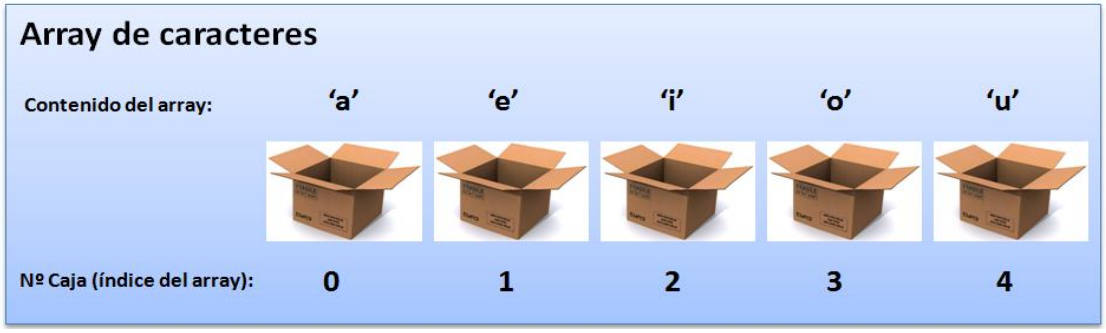
Los tipos de datos por referencia no son inferidos por el intérprete de JavaScript y, por tanto, para que la variable almacene las referencias a los objetos es necesario especificar dicho tipo en la declaración de la variable. Veremos más sobre estas características al abordar el estudio de la orientación a objetos en JavaScript

4.3.2. Arrays

Además de los tipos de datos que hemos visto anteriormente, existe un tipo de dato llamado Array que se clasifica formalmente como un tipo de dato por referencia. Al abordar las variables, hemos visto que éstas se comportan como si fueran una caja en la que se puede guardar información de un tipo determinado. De manera análoga, podemos ver un array como una colección de cajas que almacenan información acerca de una determinada entidad.

Por ejemplo, si en un programa tenemos que trabajar con los días de la semana, podría resultar útil definir un array de 7 posiciones para almacenar en cada posición un día de la semana.

Cada elemento del array tiene asociado un número consecutivo, el **índice del array**, que permite acceder a cada uno de los elementos del array. El índice del primer elemento del array es cero.



Como veremos más adelante, JavaScript permite acceder al valor almacenado en cada posición del array mediante el índice o bien mediante un identificador asociado a cada elemento del array.

Si ya tienes conocimientos básicos de programación, fíjate en que el concepto de array en JavaScript difiere del concepto de array en los demás lenguajes de programación: en JavaScript un array es una estructura de datos dinámica, cuyo tamaño puede cambiar en tiempo de ejecución y que puede contener valores de distinto tipo. Además, los arrays en JavaScript pueden ser asociativos, de modo que es posible acceder por un nombre a los valores del array en vez de utilizar el índice del array. Estas características lo hacen más parecido a un tipo de dato HashMap<Object> (en Java) o Dictionary<Object> (en .Net).

El array del ejemplo anterior tiene una única dimensión. Este tipo de arrays también se denomina **vector.** Si el array tiene dos dimensiones se le denomina **matriz**. En las matrices existen dos índices, uno para cada dimensión.

La **declaración** de vectores y matrices tiene la siguiente sintaxis:

- Si conocemos de antemano los valores a almacenar en las posiciones del array, puede utilizarse la sintaxis siguiente:

  var <*nombre del array*> **= [**<*valor 1*>, <*valor 2*>,…<*valor n*>**];**

  por ejemplo:

  var vocales = [“a”, “e”, “i”, “o”, “u”];

- Si no conocemos en tiempo de diseño el valor de las posiciones del array, sino que dichos valores se determinan en tiempo de ejecución (por ejemplo, porque el usuario debe introducirlos interactivamente), la declaración se realiza con la siguiente sintaxis:

  var <*nombre del array*> **= new Array(** [*número de posiciones inicial*] **)**

Una vez definido un array, es muy sencillo acceder a cada uno de sus elementos, indicando entre corchetes su posición dentro del array, teniendo siempre en cuenta que el primer elemento del array se encuentra en la posición 0:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado"];

var diaSeleccionado1 = días[0] // diaSeleccionado1 = “Lunes”
var diaSeleccionado2 = días[5] // diaSeleccionado2 = “Sábado”

alert(diaSeleccionado1);
alert(diaSeleccionado2);

// añade el domingo al array
dias[6] = “Domingo”;
```

Un array es tipo de dato por referencia (o lo que es lo mismo, un array es un ***objeto***). Esto quiere decir que, a diferencia de los tipos de datos por valor (que son tipos "simples"), un array es una estructura compleja que contiene no sólo los datos almacenados en las posiciones del array sino que define una serie de propiedades y funciones que permiten manipular esos datos. Por ejemplo, cualquier array dispone de una propiedad **length** que permite conocer cuántas posiciones tiene el array.

Para acceder a las propiedades o funciones definidas en un objeto se escribe el identificador del objeto seguido de un punto y del nombre de propiedad o función que se quiere invocar:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"];

var cuantasPosiciones = dias.length;

alert("El array dias tiene " + cuantasPosiciones + "posiciones");
```

Algunas funciones definidas para los array son:

- **concat** (<*datos separados por comas*>) → se emplea para concatenar los elementos que ya posee el array con los elementos pasados como parámetros.

- **pop** ( ) → elimina el último elemento del array y lo devuelve. El array original se modifica y su longitud disminuye en un elmento.

- **push** (<*nuevo elemento*>) → añade un elmento al final del array. El array original se modifica y aumenta su longitud en 1 elmento.

- **shift** ( ) → elimina el primer lemento del array y lo devuelve. El array original se va modificado y su longitud disminuida en 1 elmento.

- **unshift** (<*nuevo elmento*>) → añade un elmento al principio del array. El array original se modifica y aumenta su longitud en 1 elemento.

- **reverse** ( ) → modifica un array colocando sus elementos en el orden inverso a su posición original.

Una de las operaciones más habituales que se realizan con un array es el recorrido secuencial de cada una de sus posiciones:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"];

for(var i = 0; i < dias.length; i = i + 1)
       {
       alert(días[i]);
       }
```

4.4. Operadores.

Un operador es una instrucción que permite realizar una operación sobre uno o varios operandos. En JavaScript están definidos los siguientes operadores:

- **Operador de asignación**: se representa mediante el carácter "**=**". Se utiliza para guardar un valor en una variable. Siempre se almacena en la variable de la izquierda el valor de la variable de la derecha:

```
// declara e inicializa la variable numero1.
// Mediante el operador asignación (=), guarda en dicha variable
// el valor numérico 7.
var numero1 = 7;

// declara la variable numero2. Inicialmente tiene el valor 'undefined'.
var numero2;

// Inicializa la variable numero2 asignándole el valor de
// la variable numero1.
numero2 = número1;

// sobreescribe el contenido de la variable numero2 y le
// asigna un nuevo valor.
numero2 = 35;
```

- **Operadores aritméticos:** suma ("**+**"), resta("**-**"), producto ("*****"), división("**/**") y módulo ("**%**").

```
// Declara e inicializa dos variables con los valores 5 y 3 respectivamente
var operando1 = 5;
var operando2 = 3;

// Declara una variable y le asigna el resultado de multiplicar el valor
// almacenado por las anteriores variables.
var resultado = operando1 * operando2;

alert(resultado);
```

- **Operadores relacionales:** igual que ("**==**"), distinto que ("**!=**"), mayor que ("**>**"), mayor o igual que ("**>=**"), menor que ("**<**"), menor o igual que ("**<=**"). Permiten comparar dos expresiones o variables. Cuando se utiliza el operador de igualdad, el intérprete trata de convertir las dos expresiones comparadas a un mismo tipo antes de realizar la comparación (autoboxing). Cuando se comparan dos cadenas de caracteres, la comparación se realiza de izquierda a derecha letra a letra. También existen el operador "idéntico que", representado por "**===**", que comprueba que los valores y los tipos de las variables son iguales.

```
// Declara e inicializa dos variables con los valores 5 y 3 respectivamente
var operando1 = 5;
var operando2 = 3;

// Verifica cuál de los dos operandos es mayor y muestra un mensaje.
if (operando1 > operando2)
      alert("operando1 es mayor que operando2");
else
      alert("operando1 es menor o igual que operando2");
```

- **Operadores lógicos:** AND (“**&&**”) , OR (“**||**”) y NOT (“**!**”). Permiten evaluar expresiones lógicas.

```
var edad = 23;
var sexo = “Mujer”;

if ((edad > 18) && (sexo == “Mujer”))
      alert(“Bienvenida al sitio web”);
else
      alert(“Debes ser mayor de edad y mujer para entrar a este sitio”);
```

- Operador de resolución de tipo: permite determinar el tipo de dato que almacena una variable. Se utiliza la palabra clave “**typeof**”.

```
var edad = 23;
var sexo = “Mujer”;

alert(typeof(edad));
alert(typeof(sexo));
```

4.5. Instrucciones de control de flujo

JavaScript define las siguientes instrucciones de control de flujo:

- Instrucciones de selección:
  - If
  - If – else
  - If – esle if - else
  - switch

- Instrucciones de iteración:
  - for
  - for…in
  - while
  - do…while

La sintaxis de estas instrucciones es la misma que hemos visto en el documento de introducción a los lenguajes de programación.

4.6. Funciones

Durante el desarrollo de un programa es muy habitual utilizar una y otra vez la misma secuencia de instrucciones. Cuando es necesario ejecutar una misma secuencia de instrucciones en distintas partes del programa resulta conveniente agrupar dicha secuencia de instrucciones en una **función**, de manera que la secuencia de instrucciones sólo se escribe una vez. Cada vez que sea necesario, puede ***invocarse*** la ejecución de las instrucciones que contiene la función. En definitiva, una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se puede reutilizar.

La sintaxis para declarar una función es la siguiente:

```
function <nombre>([{nombre de parámetro}])
       {
       <instrucciones de la función>
       [return <valor>]
       }
```

Cuando se declara una función en el código fuente de un script, dicha función sólo se ejecuta cuando la función es invocada. Para invocar la ejecución de una función basta con escribir su nombre y opcionalmente la lista de parámetros (en caso de que la función se haya diseñado para recibir parámetros).

Al finalizar la ejecución de una función, se devuelve un valor al programa principal. Para especificar el valor a devolver desde la función, se utiliza la instrucción **return** seguida de la variable, expresión o literal a devolver. Si no se especifica la instrucción return, la función devuelve el valor "null" al programa principal.

En el siguiente ejemplo, se ha implementado una función llamada "sumar" que recibe dos parámetros. Al ejecutarse la función, toma los parámetros recibidos, los suma y devuelve el valor de la suma al programa principal:

```
// Declaración de la función sumar.
function sumar(num1, num2)
       {
       var resultado = num1 + num2;
       return resultado;
       }

// Comienzo del programa principal.
var sumando1 = 3;
var sumando2 = 5;
var resultado;

// llama a la función sumar para que se ejecute y recoge el resultado
// devuelto por la función en la variable resultado.
resultado = sumar(sumando1, sumando2);

alert("El resultado de la suma es: " + resultado);
```

Cuando desde el programa principal se invoca a una función que recibe parámetros debe prestarse especial atención a que el número y el tipo de dato de cada parámetro sea el que la función espera recibir.

Al invocar a una función, es posible pasar como parámetro un literal, una expresión o una variable. El primer parámetro que se pase en la invocación se *mapeará* (asociará) con el primer parámetro definido en la declaración de la función, el segundo parámetro que se pase en la invocación se mapeará con el segundo parámetro definido en la declaración de la función y así sucesivamente.

4.7. Alcance de las variables

El alcance (también llamado ámbito o “scope”) de una variable es la parte del código fuente en la que dicha variable “existe” (es accesible). En general, el alcance de una variable se extiende desde su declaración hasta el cierre de llaves del bloque de código que la contiene.

Por ejemplo, si una variable se define dentro de una función, dicha variable existirá desde el punto en que se define dicha variable con la instrucción var hasta el punto en que finaliza la función.

En JavaScript, existen dos ámbitos para las variables:

- Variables de **ámbito global**: existen y son accesibles desde cualquier parte del programa.

- Variables de **ámbito local**: existen y son accesibles desde el punto en el que están declaradas hasta la finalización del bloque de instrucciones que contiene la declaración.

En el siguiente ejemplo se define una función llamada **crearMensaje** que crea una variable llamada “**mensaje**”. Esta variable es de ámbito local a la función y por tanto sólo existe desde su declaración hasta la finalización del bloque de instrucciones que contiene la declaración.

El programa principal comienza con la invocación a la función **crearMensaje**(), lo que hace que el punto de ejecución salte a dicha función. Una vez finalizada la función, la ejecución retorna de nuevo a la instrucción siguiente a la invocación de la función.

Seguidamente se muestra mediante la función alert() el valor de una variable llamada también “mensaje” que no ha sido declarada explícitamente:

```
function crearMensaje()
       {
       // la variable mensaje está definida dentro del bloque de la función
       // por lo que solo existe dentro de la función.
       var mensaje = “hola mundo”;
       alert(mensaje);
       }

// Comienzo del programa principal. Se invoca a la función crearMensaje para
// que se ejecute.

crearMensaje();

// Esta sentencia no muestra el mensaje porque la variable mensaje no existe
// fuera de la función
alert(mensaje);
```

A diferencia de las variables de ámbito local, las de ámbito global son definidas en el código del programa principal (fuera de cualquier función). Dichas variables son accesibles desde cualquier parte del programa:

```
var mensaje = "hola mundo";

function muestraMensaje()
       {
       // la variable mensaje está definida en el programa principal por lo
       // que es accessible desde el propio programa principal y desde dentro
       // de cualquier function.
       alert(mensaje);
       }
```

Pese a que la distinción entre variable local y global es en la mayoría de los lenguajes un concepto fácil de asimilar, JavaScript posee una propiedad que puede resultar confusa:

- Si una variable se declara en el programa principal (fuera de cualquier función), ya sea de manera explícita (utilizando la instrucción var) o de manera implícita (omitiendo la instrucción var y utilizando la variable directamente), dicha variable se considera **global.**

- Si en el interior de una función, una variable se declara mediante la instrucción var, dicha variable se considera **local.**

- Si en el interior de una función, una variable se declara implícitamente (omitiendo la instrucción var), dicha variable se considera **global.**

- Si una función declara una variable local con el mismo nombre que una variable global, las variables locales prevalecen sobre las globales pero sólo dentro de la función.

```
function crearMensaje()
       {
       // la variable mensaje está definida dentro del bloque de la función
       // pero no tiene una declaración explícita con la instrucción var por
       //  lo  que  el  intérprete  considera  que  es  una  variable  global.
       mensaje = "hola mundo";
       alert(mensaje);
       }

crearMensaje();
alert(mensaje);
```

5. Funciones integradas

JavaScript, al igual que el resto de lenguajes ofrece un conjunto de funciones predefinidas que pueden ser invocadas por el programador. Estas son algunas de las más utilizadas:

- **alert(** *<mensaje>* **)** → muestra una ventana emergente con el mensaje o expresión que se pasa como parámetro.

- **prompt(** *<mensaje>* [, *<texto por defecto>*] **)** → muestra una ventana emergente con el texto que se pasa como parámetro. El usuario debe introducir algún valor en la ventana emergente. Admite un valor opcional que especifica el valor por defecto que aparecerá en la ventana emergente. Esta función retorna al programa principal el valor introducido por el usuario.

- **confirm(** *<mensaje>* **)** → muestra una ventana emergente con el texto que se pasa como argumento. En la ventana se muestra un botón "Aceptar" y otro "Cancelar". La función devuelve el valor lógico "true" si el usuario pulsa "Aceptar" y retorna "false" en caso contrario.

- **parseInt(** *<variable tipo String>* [, *<precisión>*] **)** → Retorna una variable de tipo "integer" (número entero) correspondiente a la variable de tipo String que se le pasa como parámetro.

- **parseFloat(** *<variable tipo String>* **)** → Retorna una variable de tipo "float" (número con decimales) correspondiente a la variable de tipo String que se le pasa como parámetro.

- **.toString()** → convierte el contenido de una variable a una cadena de texto. En sentido estricto no es una función integrada del lenguaje sino una función de los objetos de tipo String. Sin embargo, aunque no hayamos definido una variable como un objeto de tipo String, esta función puede utilizarse sobre cualquier tipo de variable y al hacerlo, el intérprete de JavaScript convierte dicha variable a un objeto de tipo String (cadena de texto) y sobre dicho objeto invoca a la función toString(). Esta conversión implícita de una variable en un objeto de tipo String recibe el nombre de *autoboxing*.

6. El DOM

Cuando un navegador solicita una página, ésta es descargada íntegramente en el equipo cliente y seguidamente su estructura se reconstruye en memoria principal siguiendo un modelo en árbol. Dicho modelo se conoce como DOM o Document Object Model.

El DOM es una representación en memoria de un documento de marcas que permite acceder de manera sencilla a cada una de las etiquetas y atributos definidos en el documento.

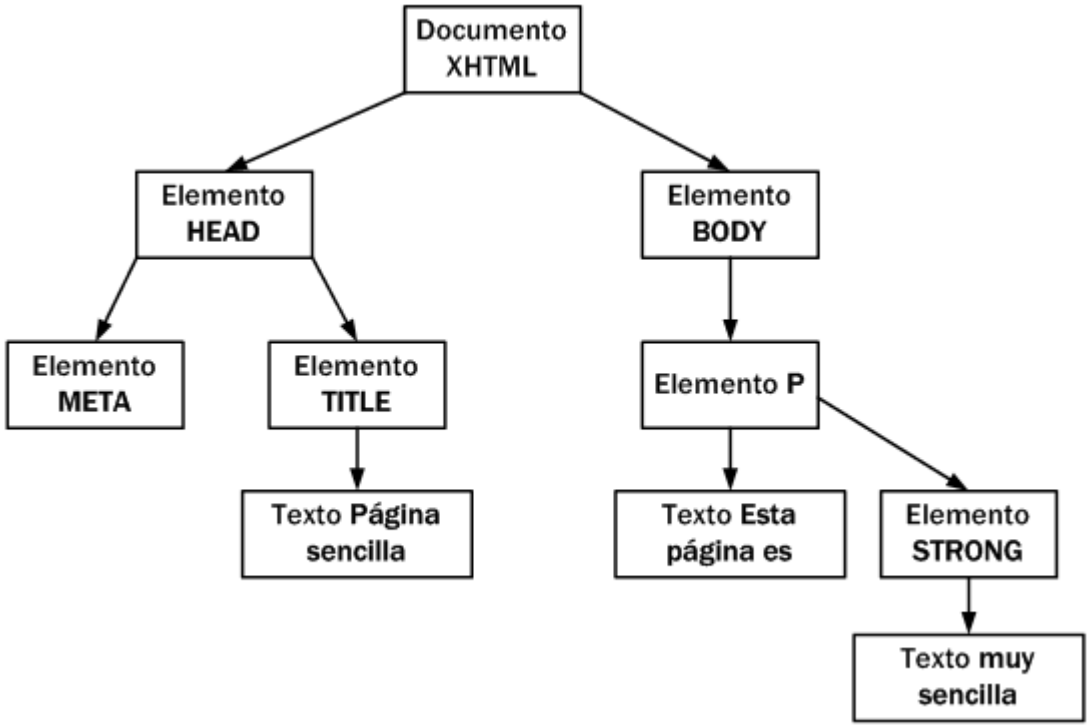
Por ejemplo, a partir del siguiente documento:

```
<!DOCTYPE html>

<html>
	<head>
		<meta charset="UTF-8"/>
		<title>Página sencilla</title>
	</head>

	<body>
		<p>Esta página es <strong>muy sencilla</strong></p>
	</body>
</html>
```

… el navegador construye la siguiente representación DOM:



Como puede verse, cada elemento del documento original se transforma en un **"nodo"** de una estructura arborescente. La raíz del árbol de nodos siempre es un tipo especial de nodo llamado "**nodo documento**".

Las etiquetas del documento original generalmente se transforman en alguno de los siguientes tipos de nodos:

- Nodo de tipo **elemento**: corresponde a una etiqueta del documento.
- Nodo de tipo **texto**: corresponde al texto encerrado entre una etiqueta de apertura y su correspondiente etiqueta de cierre.
- Nodo de tipo **atributo:** corresponde a un atributo de una etiqueta

No obstante, además de los tipos de nodo anteriores, la especificación del DOM recoge otros 12 tipos de nodos adicionales.

Una vez construido el árbol del DOM es posible acceder de manera sencilla a cualquiera de los nodos para leer, insertar, borrar o modificar. Al estar cargado en memoria principal el árbol de nodos, cualquier acceso al DOM se realiza de manera eficiente. Sin embargo, la propia naturaleza del DOM impone algunas restricciones: en primer lugar, para poder construir correctamente el árbol de nodos, el documento original debe haberse descargado completamente en la máquina cliente. Por otro lado, como el árbol de nodos reside en memoria principal, un documento excesivamente grande podría penalizar el rendimiento del equipo (los dispositivos móviles como smartphones o tablets no disponen de una memoria principal muy grande). En cualquier caso, el tamaño medio de cualquier página "normal" no suele aportar una penalización real para ningún equipo.

6.1. Funciones para el acceso a los nodos del DOM

JavaScript dispone de un conjunto de funciones para acceder a cada uno de los nodos del DOM. La función más utilizada, por ser la que ofrece mayor compatibilidad con los distintos navegadores es **getElementById(** *<id etiqueta>* **)**. Esta función localiza cualquier nodo cuyo atributo "id" sea igual al valor especificado como parámetro. Como el atributo "id" debe tener un valor único para cada etiqueta, esta función devuelve un único nodo.

El valor que devuelve la función **getElementById** *NO ES UNA CADENA DE TEXTO* sino un objeto nodo del DOM. Por tanto para acceder a las propiedades de dicho nodo, como el valor o los atributos de la etiqueta será necesario acceder a los miembros del nodo como veremos a continuación.

Otra de las funciones que permite acceder a los nodos del DOM es **getElementsByTagName(** *<nombre de etiqueta>* **)**. Esta función permite obtener todas las etiquetas cuyo nombre coincide con el parámetro especificado. El valor devuelto por la función es un array de todos los objetos nodo del DOM.

6.2. Los objetos nodo del DOM

Al abordar el estudio de los tipos de datos en JavaScript mencionamos que un objeto es un tipo de dato especial. A diferencia de los tipos de datos primitivos (enteros, decimales o cadenas de texto), los objetos son tipos de datos más complejos que generalmente almacenan uno o más valores primitivos junto con una serie de funciones que permiten acceder a dichos valores o modificarlos.

Por tanto, un objeto es un "paquete" que encapsula una cierta información (tipos de datos primitivos) y un conjunto de funciones que operan sobre dicha información. Las variables que almacenan la información del objeto reciben el nombre de atributos o **propiedades** y las funciones reciben el nombre de **métodos**.

Para poder acceder a una propiedad o invocar la ejecución de un método, debe escribirse el identificador del objeto seguido de un punto (que es el **operador de acceso a miembros del objeto**) y finalmente se escribe el nombre de la propiedad a la que se quiere acceder o el método cuya ejecución se quiere invocar.

Al utilizar las funciones del apartado anterior para acceder al DOM, dichas funciones devuelven un objeto. El objeto devuelto es un nodo del DOM que contiene (encapsula):

- **Información del contenido del nodo:** es el texto encerrado entre la etiqueta de apertura y la de cierre del nodo seleccionado. Este contenido se almacena en el objeto nodo mediante un tipo de dato primitivo (string). La propiedades que permiten acceder a esta información son
    - **innerText** ó **textContent**: Se utilizan para devolver el contenido de una etiqueta cuando dicho contenido sólo está formado por texto (no contiene etiquetas anidadas). Estas propiedades no están soportadas por todos los navegadores.
    - **innerHTML**: se utiliza para devolver el contenido de una etiqueta cuando dicho contenido incluye código HTML.

```
<!DOCTYPE html>

<html>
        <head>
                <meta charset="UTF-8"/>
                <title>Página sencilla</title>
                <script type="text/javascript">
                        function test()
                            {
                            var nodoParrafo = document.getElementById("parrafo");
                            alert(nodoParrafo.innerText);
                            }
                </script>
        </head>
        <body onload="test()">
                <p id="párrafo">
                        Esta página es <strong>muy sencilla</strong>
                </p>
        </body>
</html>
```

- **Información del contenido de los atributos del nodo:** es el texto asignado a cada uno de los atributos del nodo. En cada objeto nodo hay definida una propiedad por cada atributo que posee la etiqueta. El nombre de dichas propiedades coincide con el nombre del atributo, con lo siguientes casos excepcionales:

    - Para el atributo **class** de una etiqueta se crea una propiedad cuyo nombre es **className.**

    - Para acceder a las propiedades del atributo **style** de una etiqueta debemos utilizar la notación siguiente:

        *<objeto nodo>***.style.***<propiedad CSS>*

        Al usar esta notación, las propiedades CSS cuyo nombre contenga algún guión (como “font-size” o “background-color”) se reescriben suprimiendo el guión poniendo en mayúscula la inicial de la segunda y siguientes palabras (por tanto, tendremos “fontSize” y “backgroundColor”).

  Por ejemplo, supongamos que en una etiqueta <img> se define el atributo **“src”:**

```
<!DOCTYPE html>

<html>
       <head>
              <meta charset=”UTF-8”/>
              <title>Página sencilla</title>
              <script type=”text/javascript”>
                     function test()
                         {
                         var imagen = document.getElementById(“imagen1”);
                         alert(imagen.src);
                         alert(imagen.style.border);
                         }
              </script>
       </head>

       <body onload=”test()”>
          <img id=”imagen1” src=”foto.jpg” style=”border: 0px” alt=”” />
       </body>
</html>
```

- **Métodos (funciones) para modificar las propiedades del nodo:** permiten insertar, modificar o eliminar nodos hijos. En el apartado siguiente veremos cómo insertar, modificar o eliminar nodos del DOM

6.3. Inserción, modificación y eliminación de nodos del DOM

JavaScript permite insertar, modificar y eliminar nodos del DOM. Esta capacidad suele utilizarse frecuentemente para mostrar u ocultar elementos de la página de manera interactiva.

Un aspecto a tener en cuenta a la hora de realizar estas operaciones es que el DOM es una representación en memoria de la estructura del documento. Por tanto, cualquier inserción, modificación o eliminación que hagamos se realiza sobre dicha estructura en memoria y no sobre el fichero del documento almacenado en disco. Esa es la razón por la que, si hacemos alguna modificación al DOM y a continuación mostramos el código fuente de la página, no veremos en dicho listado los cambios introducidos (pese a que el navegador sí que los muestra).

En JavaScript no existen funciones integradas para realizar operaciones de inserción, modificación y borrado sino que son los propios objetos nodo del DOM los que disponen de métodos para realizar estas operaciones. Sobre cualquier nodo del DOM podemos invocar los siguientes métodos:

- <*objeto nodo padre*>**.createElement(“**<*nombre de etiqueta*>**”)** → Crea una nueva etiqueta del tipo especificado como parámetro. Este método no inserta la nueva etiqueta en el DOM, únicamente la crea. Normalmente, <objeto nodo padre> suele ser el nodo **document.**

- <*objeto nodo padre*>**.createTextNode(“**<*texto contenido en la etiqueta*>**”)** → Crea un nodo que representa el texto contenido en una etiqueta. Este método crea un nodo de texto pero no lo inserta en una etiqueta. Normalmente, <objeto nodo padre> suele ser el nodo **document.**

- <*objeto nodo*>**.setAttribute(“**<*nombre de atributo*>**”, “**<*valor de atributo*>**”)** → Crea e inserta un nuevo atributo con el valor especificado dentro del nodo que invoca a este método. Si el atributo ya existe, es sobrescrito.

- <*objeto nodo padre*>**.appendChild(**<*objeto nodo hijo*>**)** → Inserta el objeto nodo especificado como parámetro como un nodo hijo del objeto nodo que invoca a este método. La inserción tiene lugar al final de los nodos hijos del objeto nodo padre.

- <*objeto nodo padre*>**.insertBefore(**<*nuevo nodo*>, <*nodo de referencia*>**)** → inserta un nuevo nodo, hijo de <objeto nodo padre> inmediatamente antes que <nodo de referencia>.

- <*objeto nodo padre*>**.removeChild(**<*nodo a eliminar*>**)** → Elimina el nodo especificado como parámetro de la lista de nodos hijos que tiene el objeto nodo que invoca a este método.

El DOM fue creado para trabajar con documentos XML (XHTML es sólo una extensión de XML en la que se definen ciertas etiquetas válidas) por lo que estos métodos pueden utilizarse para trabajar con cualquier tipo de documento XML, no sólo con documentos web.

La inserción de un nuevo nodo en el DOM se realiza a través de los siguientes pasos:

1. Crear un nodo de tipo *Element* que representa la nueva etiqueta a insertar en el documento.

2. Crear un nodo de tipo *Text* que representa el contenido de la etiqueta. Si la etiqueta no contiene texto (por ejemplo, una etiqueta
 o <img/>) este paso puede omitirse.

3. Añadir el nodo Text como nodo hijo del nodo Element.

4. Añadir el nodo Element al documento en forma de nodo hijo del nodo correspondiente al lugar en el que se quiere insertar el elmento.

Por ejemplo, si se quiere añadir un párrafo en el cuerpo de un documento, deberíamos incluir un código como éste:

```
<!DOCTYPE html>

<html>
   <head>
      <meta charset="UTF-8"/>
      <title>Página sencilla</title>
      <script type="text/javascript">
          function insertaParrafo()
             {
             // 1) crea un nodo de tipo Element
             var nuevoParrafo = document.createElement("p");

             // 2) crea un nodo de tipo Text
             var contenido = document.createTextNode("Hola mundo");

             // 3) Añade el nodo Text como hijo del nodo Element
             nuevoParrafo.appendChild(contenido);

             // 4) Inserta el nodo Element encima del tercer párrafo
            var contenedor = document.getElementById("contenedor");
            var parrafo3 = document.getElementById("parrafo3");
            contenedor.insertBefore(nuevoParrafo, parrafo3);
             }
      </script>
   </head>

   <body onload="insertaParrafo()">
       <div id="contenedor">
          <p>Este es el primer p&aacute;rrafo</p>
          <p id="parrafo3">Este es el tercer p&aacute;rrafo</p>
       </div>
   </body>
</html>
```

Por otro lado, para eliminar un nodo basta con accede al nodo padre del nodo a eliminar e invocar al método removeChild:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset=”UTF-8”/>
      <title>Página sencilla</title>
      <script type=”text/javascript”>
          function eliminaParrafo()
             {
              var parrafo3 = document.getElementById(“parrafo3”);

              // la propiedad parentNode devuelve una referencia al nodo
              // nodo padre del elemento que invoca a esta propiedad
              parrafo3.parentNode.removeChild(parrafo3);
             }
      </script>
   </head>
   <body onload=”eliminaParrafo()”>
       <div id=”contenedor”>
          <p id=”parrafo3”>Este es el tercer p&aacute;rrafo</p>
       </div>
   </body>
</html>
```

Por último, para modificar el valor de un nodo basta con obtener dicho nodo y hacer uso del método **setAttribute** o de las propiedades **value, innerText** e **innerHTML**. Como hemos mencionado anteriormente, **setAttribute** permite establecer (o sobreescribir) el valor de un atributo, mientras que **value** permite obtener el contenido de los controles de un formulario. **innerText** e **innerHTML** se utilizan para modificar el contenido de las etiquetas.

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset=”UTF-8”/>
      <title>Página sencilla</title>
      <script type=”text/javascript”>
          function modificaParrafo()
             {
              var parrafo = document.getElementById(“parrafo”);
              parrafo.innerText = ”hola mundo con innerText”;
              parrafo.innerHTML = ”hola mundo con innerHTML”;

              var imagen = document.getElementById(“imagen”);
              imagen.setAttribute(“width”, “10”);
             }
      </script>
   </head>
   <body onload=”modificaParrafo()”>
       <div id=”contenedor”>
          <p id=”parrafo”>un simple párrafo</p>
          <img id=”imagen”src=”foto.jpg” width=”500” />
       </div>
   </body>
</html>
```

7. Eventos

Un evento es un cambio que tiene lugar durante la ejecución de una aplicación. Por ejemplo, se produce un cambio (se dice que se dispara un evento) cuando se carga completamente la página web o cuando se hace clic sobre u botón o enlace.

JavaScript ofrece un modelo de programación basado en eventos. Esto quiere decir que el programador puede diseñar el script para que las funciones sean invocadas cuando se dispare un evento determinado.

Cada etiqueta de un documento (X)HTML tiene definida una lista de los posibles eventos que puede tener asociada dicha etiqueta. La lista completa de todos los eventos es la siguiente:

Evento	Descripción	Elementos para los que está definido
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
ondblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla y no soltarla	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón *"sale"* del elemento	Todos los elementos
onmouseover	El ratón *"entra"* en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos
onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

Existen varias alternativas para asociar el disparo de un evento con la ejecución de una función de un script:

- **Utilización de un manejador de eventos:** un manejador de eventos es un atributo especial que se añade a una etiqueta. Este atributo se llama igual que el evento y tiene como valor un fragmento de código JavaScript o una llamada a una función JavaScript definida en un script cargado previamente en la página.

```
[…]

<input type=”button” onclick=”alert(‘hola mundo’);” />

[…]
```

```
[…]
<head>
   <title></title>
   <script type=”text/javascript”>
      function saluda()
          {
          alert(“hola mundo”);
          }
   </script>
</head>

[…]

<input type=”button” onclick=”saluda();” />

[…]
```

Como en cualquier otro atributo, el valor asignado al manejador de eventos debe ir encerrado entre dobles comillas. Si, como ocurre en el primer ejemplo, dentro del código JavaScript del manejador de eventos hemos de incluir otras comillas (las que delimitan el literal “hola mundo”), deberemos sustituir esas dobles comillas por comillas simples para que el intérprete pueda reconocer correctamente el contenido del atributo.

- **Utilización de manejadores de eventos semánticos**: en vez de incluir el manejador de eventos como un atributo de la etiqueta, se puede asociar el manejador de eventos a la etiqueta desde el propio script. De este modo el código de la página queda más limpio y la asociación de eventos queda centralizada en un único punto:

```
var elemento = document.getElementById(<id_etiqueta>);
elemento.<evento> = <función de manejo de evento>;
```

Al especificar la función que va a ejecutarse cuando se dispare el evento **no se deben incluir paréntesis**, sino únicamente el nombre de la función.

Este modo de enlazar los eventos hace uso del DOM (función ***getElementById***). Como hemos visto en el apartado 6, antes de poder acceder al DOM hemos de esperar a que la página se cargue por completo por lo que, antes de capturar la etiqueta con la función *getElementById*, es necesario esperar a que tenga lugar el evento ***onload***. Por tanto, para utilizar correctamente los manejadores de eventos semánticos deberíamos utilizar una estructura similar a la siguiente:

```
[…]
<head>
   <title></title>
   <script type="text/javascript">

      function saluda()
          {
          alert("hola mundo");
          }

       function asignaEventos()
          {
          var boton = document.getElementById("miBoton");
          boton.onclick = saluda;
          }

       window.onload = asignaEventos;
   </script>
</head>

[…]

<input id="miBoton" type="button" />

[…]
```

Los manejadores de eventos fueron introducidos en la especificación DOM de nivel 1 y en la actualidad están soportados por todos los navegadores. Por otro lado, los manejadores de eventos semánticos se introdujeron con la especificación DOM de nivel 2 y también tienen soporte universal por todos los navegadores actuales.

No obstante estos manejadores de eventos plantean una pequeña limitación: no es posible asignar múltiples funciones a un mismo evento. Por ejemplo, si tenemos el siguiente código:

```
[…]
function muestraMensaje()
       {
       alert("hola");
       }

function muestraOtroMensaje()
       {
       alert ("hola otra vez");
       }

var div = document.getElementById("div_contenedor");
div.onclick = muestraMensaje;
div.onclick = muestraOtroMensaje;
[…]
```

Al hacer clic sobre el div no se ejecutan las dos funciones, como cabría esperar. La solución pasaría por crear una nueva función que invocara a las dos funciones que se quiere ejecutar. Como puede verse, esta limitación no plantea graves inconvenientes aunque sí obliga a crear una nueva función.

Para superar esta limitación, los distintos navegadores han ido introduciendo otros modos de asociar eventos con funciones. Sin embargo, estas alternativas dependen del navegador e incluso a día de hoy hay grandes variaciones de un navegador a otro.

Internet Explorer ofrece las funciones *attachEvent y detachEvent*, mientras que otros navegadores soportan métodos específicos (*addEventListener y removeEventListener*) introducidos en posteriores revisiones del DOM.

```
[…]
function muestraMensaje()
       {
       alert(“hola”);
       }

function muestraOtroMensaje()
       {
       alert (“hola otra vez”);
       }

var div = document.getElementById(“div_contenedor”);
div.attachEvent = muestraMensaje;
div.attachEvent = muestraOtroMensaje;
[…]
// más adelante se desasocia la función del evento

div.detachEvent(“onclick”, muestraMensaje);
```

7.1. El identificador ***this***

JavaScript define un identificador especial llamado ***this*** que se crea automáticamente. Cuando se trabaja con manejadores de eventos, se puede utilizar *this* para referirse al elemento de la página que ha disparado el evento. De este modo, no es necesario capturar el elemento que ha disparado el evento mediante document.getElementById(<id>) sino que basta con especificar ***this.***

Por ejemplo, supongamos que tenemos un div cuyo color de fondo cambia cuando el usuario pasa el ratón por encima. Si no usamos el identificador this, el código sería similar al siguiente:

```
[…]
<div id=”texto”
onmouseover=”document.getElementById(‘texto’).style.backgroundColor=’red’;”
onmouseout=”document.getElementById(‘texto’).style.backgroundColor=’green’;”>

       Dolorem Ipsum …

</div>
```

Sin embargo, haciendo uso del identificador this, el código sería algo más sencillo:

```
[…]
<div id=”texto” onmouseover=”this.style.backgroundColor=’red’;”
onmouseout=”this.style.backgroundColor=’green’;”>

       Dolorem Ipsum …

</div>
```

7.2. Cómo obtener información del evento

Normalmente, las funciones que se disparan en respuesta a un evento requieren información adicional para realizar la tarea para la que han sido diseñadas. Por ejemplo, si una función responde al evento **onclick**, quizás necesite saber en qué posición estabe el ratón en el momento de pinchar el botón. Del mismo modo , es posible que una función que se dispara ante pulsaciones de teclado neceseite conocer cuál es la tecla que se ha pulsado.

JavaScript permite obtener información sobre el contexto en el que se han disparado los eventos a través de un objeto especial llamado ***event***.

Desafortunadamente, los diferentes navegadores presentan diferencias muy notables en el tratamiento de la información sobre los eventos. La principal diferencia reside en la forma en la que se obtiene el objeto event:

- Internet Explorer considera que este objeto forma parte del objeto window
- El resto de navegadores lo consideran como el único argumento que tienen las funciones manejadoras de eventos.

Aunque es un comportamiento que resulta extraño (y que viola los principios más elementales de diseño de software), todos los navegadores excepto Internet Explorer crean mágicamente y de forma automática un argumento que se pasa a la función manejadora, por lo que no es necesario incluirlo en la llamada a la función manejadora.

De esta forma, para utilizar este "argumento mágico", sólo es necesario asignarle un nombre, ya que los navegadores lo crean automáticamente.

En resumen, en los navegadores tipo Internet Explorer, el objeto event se obtiene directamente mediante:

```
var evento = window.event
```

En el resto de navegadores, el objeto event se obtiene mágicamente a partir del argumento que el navegador crea automáticamente:

```
function manejadorDeEventos(eventoDisparado)
	{
	var evento = eventoDisparado;
	}
```

Si se quiere programar una aplicación que funcione correctamente en todos los navegadores, es necesario obtener el objeto event de forma correcta según cada navegador. El siguiente código muestra la forma correcta de obtener el objeto event en cualquier navegador:

```
function manejadorDeEventos(eventoDisparado)
	{
	var evento;
	// la siguiente comprobación verifica si el navegador ha creado el
	// objeto event mediante el "parámetro mágico" eventoDisparado.
	// Esta comprobación devolverá el valor lógico true en los navegadores
	// DOM-compliant, como FireFox.
	if (eventoDisparado != null)
		{
		evento = eventoDisparado;
		}
	else
		{
		// la siguiente comprobación verifica si el navegador ha creado
		// el objeto event mediante el objeto window. Esta comprobación
		// devolverá el valor lógico true en IE.
		if (window.event != null)
			evento = window.event;
		}
	}
```

La función anterior puede reescribirse de un modo mucho más compacto utilizando la siguiente sintaxis:

```
function manejadorDeEventos(eventoDisparado)
	{
	var evento = (eventoDisparado || window.event);
	}
```

La sintaxis anterior puede parecer sorprendente: normalmente utilizamos los operadores booleanos (AND, OR y NOT) para construir expresiones en las que intervienen variables de tipos básicos (variables de tipo booleano, de tipo numérico o de tipo cadena) que al ser evaluadas devuelven el valor lógico *true* o el valor lógico *false*. Por ejemplo, estamos habituados a construir expresiones como

```
if ( (a == 1) || (b < 7) )
	{
	[…]
	}
```

Sin embargo, además de este uso, JavaScript permite utilizar operadores booleanos aplicados a objetos y a propiedades de objetos. En este caso, evaluar una expresión en la que interviene un objeto, el intérprete de JavaScript evalúa si dicho objeto ***existe*** o, en otras palabras, si dicho objeto ***no es null***. De este modo, las siguientes expresiones son equivalentes:

`if (eventoDisparado) {[…]}`	`if (eventoDisparado != null) {[…]}`

Por otro lado, cuando utilizamos una construcción del tipo:

```
<variable> = <expresión booleana>
```

el intérprete de JavaScript evalúa la expresión booleana y, en caso de ser cierta, guarda en la variable el valor de la expresión. En caso contrario, guarda en la variable el valor ***null.***

Por tanto, volviendo a la expresión:

```
var evento = (eventoDisparado || window.event);
```

en este caso el intérprete de JavasScript evalúa la expresión en la siguiente secuencia:

1. ***¿eventoDisparado*** == ***null?***

    a. Si eventoDisparado == null, salta al paso 2.

    b. Si eventoDisparado != null devuelve el objeto eventoDisparado y termina la evaluación (sin pasar por el paso 2). Por tanto, la variable evento contiene una referencia al objeto eventoDisparado.

2. ***¿window.event*** == ***null?***

    a. Si window.event == null, la expresión devuelve null y termina la evaluación. Por tanto, la variable evento contiene null.

    b. Si window.event != null, la expresión devuelve el objeto window.event y termina la evaluación. Por tanto, la variable evento contiene una referencia al objeto window.event.

Una vez obtenido el objeto event, ya se puede acceder a toda la información relacionada con el evento, que depende del tipo de evento producido.

Las propiedades que expone el objeto event varían en función del navegador. Casi todos los navegadores con excepción de IE soportan el modelo especificado en el DOM. IE, por el contrario, implementa su propio modelo de eventos:

Para IE:

Propiedad/ Método	Devuelve	Descripción
`altKey`	Boolean	Devuelve `true` si se ha pulsado la tecla ALT y `false` en otro caso
`button`	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: `0` – Ningún botón pulsado 1 – Se ha pulsado el botón izquierdo 2 – Se ha pulsado el botón derecho 3 – Se pulsan a la vez el botón izquierdo y el derecho 4 – Se ha pulsado el botón central 5 – Se pulsan a la vez el botón izquierdo y el central 6 – Se pulsan a la vez el botón derecho y el central 7 – Se pulsan a la vez los 3 botones
`cancelBubble`	Boolean	Si se establece un valor `true`, se detiene el flujo de eventos de tipo *bubbling*
`clientX`	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
`clientY`	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
`ctrlKey`	Boolean	Devuelve `true` si se ha pulsado la tecla CTRL y `false` en otro caso
`fromElement`	Element	El elemento del que sale el ratón (para ciertos eventos de ratón)
`keyCode`	Número entero	En el evento `keypress`, indica el carácter de la tecla pulsada. En los eventos `keydown` y `keyup` indica el código numérico de la tecla pulsada
`offsetX`	Número entero	Coordenada X de la posición del ratón respecto del elemento que origina el evento
`offsetY`	Número entero	Coordenada Y de la posición del ratón respecto del elemento que origina el evento
`repeat`	Boolean	Devuelve `true` si se está produciendo el evento `keydown` de forma continuada y `false` en otro caso
`returnValue`	Boolean	Se emplea para cancelar la acción predefinida del evento
`screenX`	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
`screenY`	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
`shiftKey`	Boolean	Devuelve `true` si se ha pulsado la tecla SHIFT y `false` en otro caso
`srcElement`	Element	El elemento que origina el evento
`toElement`	Element	El elemento al que entra el ratón (para ciertos eventos de ratón)
`type`	Cadena de texto	El nombre del evento
`x`	Número entero	Coordenada X de la posición del ratón respecto del elemento padre del elemento que origina el evento
`y`	Número entero	Coordenada Y de la posición del ratón respecto del elemento padre del elemento que origina el evento

Todas las propiedades excepto *repeat* son de lectura/escritura.

Para el resto de navegadores:

Propiedad/Método	Devuelve	Descripción
`altKey`	Boolean	Devuelve `true` si se ha pulsado la tecla ALT y `false` en otro caso
`bubbles`	Boolean	Indica si el evento pertenece al flujo de eventos de *bubbling*
`button`	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: 0 – Ningún botón pulsado 1 – Se ha pulsado el botón izquierdo 2 – Se ha pulsado el botón derecho 3 – Se pulsan a la vez el botón izquierdo y el derecho 4 – Se ha pulsado el botón central 5 – Se pulsan a la vez el botón izquierdo y el central 6 – Se pulsan a la vez el botón derecho y el central 7 – Se pulsan a la vez los 3 botones
`cancelable`	Boolean	Indica si el evento se puede cancelar
`cancelBubble`	Boolean	Indica si se ha detenido el flujo de eventos de tipo *bubbling*
`charCode`	Número entero	El código unicode del carácter correspondiente a la tecla pulsada
`clientX`	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
`clientY`	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
`ctrlKey`	Boolean	Devuelve `true` si se ha pulsado la tecla CTRL y `false` en otro caso
`currentTarget`	Element	El elemento que es el objetivo del evento
`detail`	Número entero	El número de veces que se han pulsado los botones del ratón

eventPhase	Número entero	La fase a la que pertenece el evento: 0 – Fase capturing 1 – En el elemento destino 2 – Fase bubbling
isChar	Boolean	Indica si la tecla pulsada corresponde a un carácter
keyCode	Número entero	Indica el código numérico de la tecla pulsada
metaKey	Número entero	Devuelve true si se ha pulsado la tecla META y false en otro caso
pageX	Número entero	Coordenada X de la posición del ratón respecto de la página
pageY	Número entero	Coordenada Y de la posición del ratón respecto de la página
preventDefault()	Función	Se emplea para cancelar la acción predefinida del evento
relatedTarget	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
screenX	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
screenY	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
shiftKey	Boolean	Devuelve true si se ha pulsado la tecla SHIFT y false en otro caso
stopPropagation()	Función	Se emplea para detener el flujo de eventos de tipo *bubbling*
target	Element	El elemento que origina el evento
timeStamp	Número	La fecha y hora en la que se ha producido el evento
type	Cadena de texto	El nombre del evento

Habitualmente resulta útil obtener la etiqueta que ha disparado el evento. Esto puede hacerse a partir del objeto event, que entre otra información, contiene una referencia a la etiqueta que ha disparado el evento. Para obtener dicho elemento en cualquier navegador puede utilizarse la siguiente secuencia de instrucciones:

```
function manejadorDeEventos(eventoDisparado)
       {
       var evento = (eventoDisparado || window.event);
       var elementoDisparador = (evento.srcElement || evento.target);
       }
```

`eventPhase`	Número entero	La fase a la que pertenece el evento: `0` – Fase capturing 1 – En el elemento destino 2 – Fase bubbling
`isChar`	Boolean	Indica si la tecla pulsada corresponde a un carácter
`keyCode`	Número entero	Indica el código numérico de la tecla pulsada
`metaKey`	Número entero	Devuelve `true` si se ha pulsado la tecla META y `false` en otro caso
`pageX`	Número entero	Coordenada X de la posición del ratón respecto de la página
`pageY`	Número entero	Coordenada Y de la posición del ratón respecto de la página
`preventDefault()`	Función	Se emplea para cancelar la acción predefinida del evento
`relatedTarget`	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
`screenX`	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
`screenY`	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
`shiftKey`	Boolean	Devuelve `true` si se ha pulsado la tecla SHIFT y `false` en otro caso
`stopPropagation()`	Función	Se emplea para detener el flujo de eventos de tipo *bubbling*
`target`	Element	El elemento que origina el evento
`timeStamp`	Número	La fecha y hora en la que se ha producido el evento
`type`	Cadena de texto	El nombre del evento

A diferencia de las propiedades de event para IE, las propiedades definidas por el DOM para el resto de navegadores son de sólo-lectura.

Si desde la función manejadora del evento se quiere acceder al elemento que ha disparado el evento se debe utilizar el siguiente código:

```
// Internet Explorer
var objetivo = elEvento.srcElement;

// Navegadores que siguen los estandares
var objetivo = elEvento.target;
```

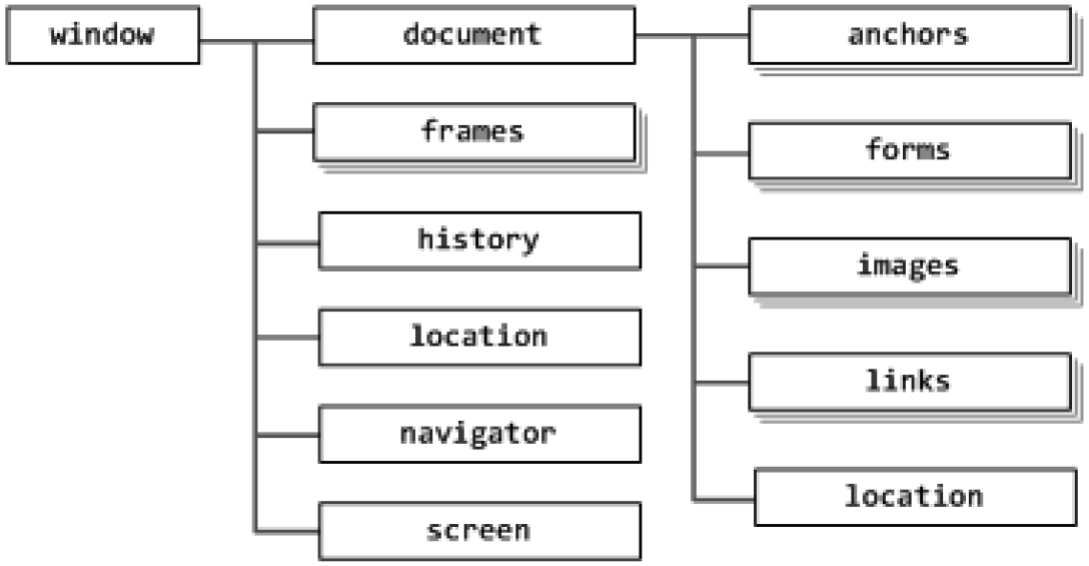
8. El BOM

El Browser Object Model es una especificación introducida en las primeras versiones de los navegadores que define algunas características acerca de los mismos, permitiendo acceder y modificar las propiedades de las ventanas del propio navegador.

Mediante las funciones del BOM es posible redimensionar y mover la ventana del navegador, modificar el texto que se muestra en la barra de estado y realizar muchas otras modificaciones no relacionadas con el contenido del documento (X)HTML.

El mayor inconveniente del BOM es que, al contrario de lo que sucede con el BOM, ninguna entidad se encarga de estandarizarlo o de definir unos mínimos de interoperabilidad entre navegadores.

El siguiente esquema muestra los objetos del BOM y su relación:



(en este esquema, los elementos mostrados con varios recuadros superpuestos son arrays. El resto de objetos son simples.)

8.1. El objeto *window*

El objeto ***window*** representa la ventana completa del navegador. Los métodos que expone este objeto son de uso frecuente en los scripts por lo que la mayoría de intérpretes de JavaScript permiten omitir el prefijo "window." haciendo posible invocar los métodos de este objeto directamente.

Mediante este objeto es posible mover, redimensionar y manipular la ventana del navegador mediante los métodos ***moveBy(x, y)*** , ***moveTo(x, y)***, ***resizeBy(x, y)*** y ***resizeTo(x, y).***

Dos de los métodos del objeto *window* que más se utilizan son ***setTimeout() y setInterval()***:

- **setTimeout(*<función>, <tiempo en ms>*) :** este método permite ejecutar una función al transcurrir un determinado periodo tiempo. El parámetro <función> se especifica con el nombre de la función a ejecutar (sin incluir los paréntesis). Esta función será ejecutada 1 vez, cuando haya transcurrido el tiempo especificado en el segundo parámetro del método.
  El método setTimeout devuelve un identificador que puede utilizarse para cancelar la ejecución de la función antes de que se cumpla el tiempo especificado, utilizando para ello el método ***clearTimeout***(*<id>*):

```
function muestraMensaje()
       {
       alert("Han transcurrido 3 segundos");
       }

function cancelar()
       {
       clearTimeout(id);
       }

var id;
id = setTimeout(muestraMensaje, 3000);

[…]

<input type="button" onclick="cancelar()" value="cancelar temporizador" />
```

- **setInterval(*<función>, <tiempo en ms>*):** su funcionamiento es similiar al método anterior pero en este caso la función que se pasa como parámetro es invocada repetidamente en intervalos de tiempo cuya duración coincide con el segundo parámetro del método. De forma análogo al método clearTimeout, para este método se define ***clearInterval(***<id>***)***, que permite desactivar el temporizador.

Estos métodos resultan útiles para realizar animaciones o para permitir que una función espere como máximo un determinado lapso de tiempo antes de continuar.

8.2. El objeto ***document***

El objeto document pertenece tanto al DOM como al BOM y proporciona información sobre la página:

Propiedad	Descripción
`lastModified`	La fecha de la última modificación de la página
`referrer`	La URL desde la que se accedió a la página (es decir, la página anterior en el array `history`)
`title`	El texto de la etiqueta `<title>`
URL	La URL de la página actual del navegador

Además de las propiedades anteriores, el objeto document contiene varios arrays con información sobre algunos elementos de la página:

Array	Descripción
`anchors`	Contiene todas las "anclas" de la página (los enlaces de tipo `<a name="nombre_ancla"></a>`)
`applets`	Contiene todos los applets de la página
`embeds`	Contiene todos los objetos embebidos en la página mediante la etiqueta `<embed>`
`forms`	Contiene todos los formularios de la página
`images`	Contiene todas las imágenes de la página
`links`	Contiene todos los enlaces de la página (los elementos de tipo `<a href="enlace.html"></a>`)

8.3. El objeto ***location***

Este objeto representa la URL de la página que se muestra en la ventana del navegador. Debido a la falta de estandarización, este objeto forma parte tanto del objeto document como del objeto window:

Propiedad	Descripción
hash	El contenido de la URL que se encuentra después del signo # (para los enlaces de las anclas) http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion hash = #seccion
host	El nombre del servidor http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion host = www.ejemplo.com
hostname	La mayoría de las veces coincide con host, aunque en ocasiones, se eliminan las www del principio http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion hostname = www.ejemplo.com
href	La URL completa de la página actual http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion URL = http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion
pathname	Todo el contenido que se encuentra después del host http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion pathname = /ruta1/ruta2/pagina.html
port	Si se especifica en la URL, el puerto accedido http://www.ejemplo.com:8080/ruta1/ruta2/pagina.html#seccion port = 8080 La mayoría de URL no proporcionan un puerto, por lo que su contenido es vacío http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion port = (vacío)
protocol	El protocolo empleado por la URL, es decir, todo lo que se encuentra antes de las dos barras inclinadas // http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion protocol = http:
search	Todo el contenido que se encuentra tras el símbolo ?, es decir, la consulta o *"query string"* http://www.ejemplo.com/pagina.php?variable1=valor1&variable2=valor2 search = ?variable1=valor1&variable2=valor2

8.4. El objeto ***navigator***

Este objeto permite obtener información sobre el navegador con el que se está visualizando el documento (X)HTML. Se emplea habitualmente para detectar el tipo y/o versión del navegador en scripts que hacen uso de funciones para navegadores específicos. Además, también se utiliza para detectar si el navegador tiene habilitadas las cookies y determinados plugins.

Propiedad	Descripción
appCodeName	Cadena que representa el nombre del navegador (normalmente es Mozilla)
appName	Cadena que representa el nombre oficial del navegador
appMinorVersion	(Sólo Internet Explorer) Cadena que representa información extra sobre la versión del navegador
appVersion	Cadena que representa la versión del navegador
browserLanguage	Cadena que representa el idioma del navegador
cookieEnabled	Boolean que indica si las cookies están habilitadas
cpuClass	(Sólo Internet Explorer) Cadena que representa el tipo de CPU del usuario ("x86", "68K", "PPC", "Alpha", "Other")
javaEnabled	Boolean que indica si Java está habilitado
language	Cadena que representa el idioma del navegador
mimeTypes	Array de los tipos MIME registrados por el navegador
onLine	(Sólo Internet Explorer) Boolean que indica si el navegador está conectado a Internet
oscpu	(Sólo Firefox) Cadena que representa el sistema operativo o la CPU
platform	Cadena que representa la plataforma sobre la que se ejecuta el navegador
plugins	Array con la lista de plugins instalados en el navegador
preference()	(Sólo Firefox) Método empleado para establecer preferencias en el navegador
product	Cadena que representa el nombre del producto (normalmente, es Gecko)
productSub	Cadena que representa información adicional sobre el producto (normalmente, la versión del motor Gecko)
securityPolicy	Sólo Firefox
systemLanguage	(Sólo Internet Explorer) Cadena que representa el idioma del sistema operativo
userAgent	Cadena que representa la cadena que el navegador emplea para identificarse en los servidores
userLanguage	(Sólo Explorer) Cadena que representa el idioma del sistema operativo
userProfile	(Sólo Explorer) Objeto que permite acceder al perfil del usuario

8.5 El objeto ***screen***

Se utiliza para obtener información sobre la pantalla del usuario. Uno de los datos más importantes que proporciona el objeto screen es la resolución del monitor en el que se están visualizando las páginas. De este modo, es posible adaptar el diseño del documento a dicha resolución:

Propiedad	Descripción
`availHeight`	Altura de pantalla disponible para las ventanas
`availWidth`	Anchura de pantalla disponible para las ventanas
`colorDepth`	Profundidad de color de la pantalla (32 bits normalmente)
`height`	Altura total de la pantalla en píxel
`width`	Anchura total de la pantalla en píxel