DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA, UNAM
CURSOS ABIERTOS

CURSO: ***CC043 Taller de Desarrollo de Páginas WEB con Java y JavaScript***
FECHA: ***8 al 12 de julio del 2002***

EVALUACIÓN DEL PERSONAL DOCENTE

(ESCALA DE EVALUACIÓN 1 A 10)

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACIÓN CON EL ASISTENTE	PUNTUALIDAD
Ing. Alejandro Velázquez Mena				

Promedio _____

EVALUACIÓN DE LA ENSEÑANZA

CONCEPTO	CALIF.
ORGANIZACIÓN Y DESARROLLO DEL CURSO	
GRADO DE PROFUNDIDAD DEL CURSO	
ACTUALIZACION DEL CURSO	
APLICACION PRACTICA DEL CURSO	

Promedio _____

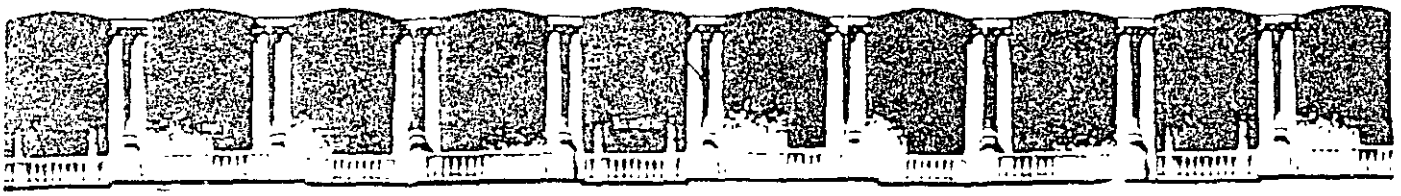
EVALUACIÓN DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDÁCTICO UTILIZADO	

Promedio _____

Evaluación total del curso_____

Continúa...2



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MATERIAL DIDACTICO DEL CURSO

TALLER DE DESARROLLO DE PAGINAS WEB CON JAVA Y JAVASCRIPT

JULIO, 2002

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México D.F. APDO. Postal M-2285
Teléfonos: 5510-0855 5512-8103 ext. 103 5521-1937 Fax: 5510-0573 5521-4021 AL 25

Introducción

Para poder seguir este curso necesitas un navegador capaz de ejecutar Javascript. Esto es: Netscape 2 o superior y Explorer 3 o superior. Sin embargo, según se avanza en el curso es posible que algunas cosas no funcionen en navegadores antiguos, aún cuando soporten Javascript. Para poder teclear los ejemplos no hace falta más que un editor como el bloc de notas de Windows o similares. Así que, si cumples todos los requisitos adelante.

¿Qué es Javascript?

JavaScript, al igual que Java o VRML, es una de las múltiples maneras que han surgido para extender las capacidades del lenguaje HTML. Al ser la más sencilla, es por el momento la más extendida. Antes que nada conviene aclarar un par de cosas:

1. JavaScript no es un lenguaje de programación propiamente dicho. Es un lenguaje *script* u orientado a documento, como pueden ser los lenguajes de macros que tienen muchos procesadores de texto. Nunca podrás hacer un programa con JavaScript, tan sólo podrás mejorar tu página Web con algunas cosas sencillas (revisión de formularios, efectos en la barra de estado, etc...) y, ahora, no tan sencillas (animaciones usando HTML dinámico, por ejemplo).

2. JavaScript y Java son dos cosas distintas. Principalmente porque Java sí que es un lenguaje de programación completo. Lo único que comparten es la misma sintaxis.

Este documento tiene como objetivo mostrar una parte de las potencialidades del JavaScript. No pretendo hacer aquí una guía completa, sino sólo una pequeña introducción. Para tener una guía de referencia es mejor acudir a la que ofrece Netscape que, al fin y al cabo, son los creadores del invento

Existen tres versiones de JavaScript. Casi todo lo que hay en este curso funciona con la versión 1.0, que nació con el Netscape Navigator 2.0. No nos detendremos en la compatibilidad demasiado, ya que en el mundo hispano no existen demasiadas copias de Netscape con versiones inferiores a la 4.

También existe una versión 1.3, introducida en la versión 4.06 (si no recuerdo mal). Esta versión es una pequeña revisión de la 1.2 creada para ajustarse al estándar internacional ECMA que regula el lenguaje Javascript, y por su poca importancia no será cubierto aquí.

El Microsoft Explorer soporta el JavaScript, sólo que cambiándole el nombre. La versión 3.0 interpreta el JScript, que es similar al JavaScript 1.0 pero con algunas diferencias para provocar ciertas incompatibilidades. El Explorer 4 parece que sí que admite JavaScript 1.1 con cierta fiabilidad, y muchas cosas del 1.2. La siguiente tabla ofrece un pequeño resumen de la compatibilidad de las versiones.

Versión	Navegador	Versión	Navegador
Javascript 1.0	Netscape 2.0	JScript 1.0	Explorer 3.0
Javascript 1.1	Netscape 3.0	JScript 3.0	Explorer 4.0
Javascript 1.2	Netscape 4.0-4.05	JScript 5.0	Explorer 5.0
Javascript 1.3	Netscape 4.06-4.71		

A partir de ahora asumo que quien esté leyendo esto conoce el HTML. También servirá de ayuda que estés familiarizado con la sintaxis de C, C++ o Java, aunque en los próximos capítulos explicaremos la misma.

Primeros pasos

Vamos a realizar nuestro primer "programa" en JavaScript. Haremos surgir una ventana que nos muestre el ya famoso mensaje "hola, mundo". Así podremos ver los elementos principales del lenguaje. El siguiente código es una página Web completa con un botón que, al pulsarlo, muestra el mensaje.

HolaMundo.html

```
<HTML>
<HEAD>
  <SCRIPT LANGUAGE="JavaScript">
    function HolaMundo() {
      alert("¡Hola, mundo!");
    }
  </SCRIPT>
</HEAD>
<BODY>
<FORM>
  <INPUT TYPE="button" NAME="Boton" VALUE="Pulsame"
         onClick="HolaMundo()">
</FORM>
</BODY>
</HTML>
```

Y aquí está nuestro ejemplo funcionando:

Ahora vamos a ver, paso por paso, que significa cada uno de los elementos extraños que tiene la página anterior:

```
<SCRIPT LANGUAGE="JavaScript">
</SCRIPT>
```

Dentro de estos elementos será donde se puedan poner funciones en JavaScript. Puedes poner cuantos quieras a lo largo del documento y en el lugar que más te guste. Yo he elegido la cabecera para hacer más legible la parte HTML de la página. Si un navegador no acepta JavaScript no leerá lo que hay entre medias de estos elementos. Así que si programamos algo que sólo funcione con la versión 1.1 pondríamos `LANGUAGE= "JavaScript1.1"` para que los navegadores antiguos pasen olímpicamente del código y no se hagan un lío.

```
function HolaMundo() {
  alert("¡Hola, mundo!");
}
```

Esta es nuestra primera función en JavaScript. Aunque JavaScript esté orientado a objetos no es de ningún modo tan estricto como Java, donde nada está fuera de un objeto. Para las cosas que se van a hacer en este tutorial, no vamos a crear ninguno, pero usaremos los que vienen en la descripción del lenguaje. En el código de la función vemos una llamada al método `alert` (que pertenece al objeto `window`) que es la que se encarga de mostrar el mensaje en pantalla. Por un fallo del Netscape no se pueden poner las etiquetas HTML de caracteres especiales en una función: no los reconoce. Así que pondremos directamente "¡" arriesgándonos a que salga de otra manera en ordenadores con un juego de caracteres distinto al del nuestro.

```
<FORM>
  <INPUT TYPE="button" NAME="Boton" VALUE="Pulsame"
         onClick="HolaMundo()">
</FORM>
```

Dentro del elemento que usamos para mostrar un botón vemos una cosa nueva: `onClick`. Es un controlador de evento. Cuando el usuario pulsa el botón, el evento click se dispara y ejecuta el código que tenga entre comillas el controlador de evento `onClick`, en este caso la llamada a la función `HolaMundo()`, que tendremos que haber definido con anterioridad. Existen muchos más eventos que iremos descubriendo según avancemos en el tutorial. En el cuarto capítulo hay un resumen de todos ellos.

En realidad, podríamos haber escrito lo siguiente:

```
<FORM>
  <INPUT TYPE="button" NAME="Boton" VALUE="Pulsame"
       onClick="alert('&iexcl;Hola,Mundo!')">
</FORM>
```

y nos habríamos ahorrado el tener que escribir la función y todo lo que le acompaña, además de conseguir que nos reconozca el caracter especial ¡. Sin embargo me pareció conveniente hacerlo de esa otra manera para mostrar más elementos del lenguaje en el ejemplo.

Elementos básicos

Lo primero que vamos a ver son los ladrillos básicos del lenguaje. Las cosas que no sirven para nada solas pero que es imprescindible aprender antes que nada.

Comentarios

Lo primero (por ser lo más fácil) es indicar cómo se ponen los comentarios. Un comentario es una parte de nuestro programa que el ordenador ignora y que, por tanto, no realiza ninguna tarea. Se utilizan generalmente para poner en lenguaje humano lo que estamos haciendo en el lenguaje de programación y así hacer que el código sea más comprensible.

En JavaScript existen dos tipos de comentarios. El primero nos permite que el resto de la línea sea un comentario. Para ello se utilizan dos barras inclinadas:

```
var i = 1; // Aqui esta el comentario
```

Sin embargo, también permite un tipo de comentario que puede tener las líneas que queramos. Estos comentarios comienzan con `/*` y terminan por `*/`. Por ejemplo:

```
/* Aqui comienza nuestro maravilloso comentario
   que sigue por aquí
   e indefinidamente hasta que le indiquemos el final */
```

Literales

Se llama así a los valores que puede tomar una variable o una constante. Aparte de los distintos tipos de números y valores booleanos:

```
"Soy una cadena"
3434
3.43
true, false
```

También podemos especificar vectores:

```
vacaciones = ["Navidad", "Semana Santa", "Verano"];
alert(vacaciones[0]);
```

Dentro de las cadenas podemos indicar varios caracteres especiales, con significados especiales. Estos son los más usados:

Carácter	Significado
\n	Nueva línea
\t	Tabulador
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida
\999	El número ASCII (según la codificación Latin-1) del carácter en hexadecimal

De este modo, el siguiente literal:

```
"El curso de Javascript (\xA9 1997-99 Daniel Rodríguez) es \"co..\"."
```

se corresponde con la cadena:

```
El curso de Javascript (© 1997-99 Daniel Rodríguez) es "co..".
```

Por último, también se pueden especificar objetos como literales, aunque no funcione en más que en Netscape 4 y superiores:

```
miNavegador = {nombre: "Netscape", version: 4.5,
               idioma: "Español", plataforma: "PC"};
alert(miNavegador.plataforma);
```

Sentencias y bloques

En Javascript las sentencias se separan con un punto y coma, y se agrupan mediante llaves ({ y }).

Tipos de datos

Un tipo de datos es la clase de valores que puede tomar un identificador (es decir, una variable o una constante). Si el tipo de datos es fecha, el identificador que tenga ese tipo sólo podrá almacenar fechas. En Javascript los tipos de datos se asignan dinámicamente según asignamos valores a las distintas variables y son los clásicos: cadenas, varios tipos de enteros y reales, valores booleanos, vectores, matrices, referencias y objetos.

Variables

Las variables son nombres que ponemos a los lugares donde almacenamos la información. En Javascript, deben comenzar por una letra o un subrayado (_), pudiendo haber además dígitos entre los demás caracteres. **No es necesario declarar una variable**, pero cuando se hace es por medio de la palabra reservada `var`. Una variable, cuando no es declarada, tiene siempre ámbito global, mientras que en caso contrario será de ámbito global si está definida fuera de una función, y local si está definida dentro:

```
var x;       // Accesible fuera y dentro de pruebas
y = 2;       // Accesible fuera y dentro de pruebas
function pruebas() {
  var z;       // Accesible sólo dentro de pruebas
  w = 1;       // Accesible fuera y dentro de pruebas
}
```

Se pueden declarar varias variables en una misma sentencia separándolos por comas:

```
var x, y, z;
```

El tipo de datos de la variable será aquel que tenga el valor que asignemos a la misma, a no ser que le asignemos un objeto por medio del operador `new`. Por ejemplo, si escribimos

```
b = 200;
```

Es de esperar que la variable `b` tenga tipo numérico.

Referencias

La parte sin duda más complicada de comprender y manejar en los lenguajes de programación tradicionales (y especialmente en C y C++) son los punteros. Por eso mismo (entre otras razones) fueron eliminados tanto de Java como de JavaScript. Sin embargo, algunas de sus capacidades han tenido que ser suplantadas con otras estructuras.

Los punteros se pueden usar para apuntar a otras variables, es decir, un puntero puede ser como un nuevo nombre de una variable dada. A esto se le suele llamar referencia. En JavaScript se pueden usar referencias a objetos y a funciones. Su mayor utilidad está en el uso de distinto código para distintos navegadores de forma transparente. Por ejemplo, supongamos que tenemos una función que sólo funciona en Internet Explorer 4, y tenemos una variable llamada `IE4` que hemos puesto previamente a `true` (verdedero) sólo si el explorador del usuario es ese.

```
function funcionIE4() {...}

function funcionNormal() {...}

var funcion = (IE4) ? funcionIE4 : funcionNormal;
  // Si IE4 es verdadero, funcion es una referencia de funcionIE4
  // Si no, funcion es una referencia de funcionNormal
```

```
funcion();
  // La llamada que haremos realmente depende de la
  // línea anterior
```

En este código, cuando llamemos finalmente a `funcion` al final en realidad llamaremos a la función a la que en la línea anterior hemos decidido que se refiera.

Vectores y matrices

Estos tipos de datos complejos son un conjunto ordenado de elementos, cada uno de los cuales es en sí mismo una variable distinta. **En Javascript, los vectores y las matrices son objetos.** Como veremos que hacen todos los objetos, se declaran utilizando el operador `new`:

```
miEstupendoVector = new Array(20)
```

El vector tendrá inicialmente 20 elementos (desde el 0 hasta el 19). Si queremos ampliarlo no tenemos más que asignar un valor a un elemento que esté fuera de los límites del vector:

```
miEstupendoVector[25] = "Algo"
```

De hecho, podemos utilizar de índices cualquier expresión que deseemos utilizar. Ni siquiera necesitamos especificar la longitud inicial del vector si no queremos:

```
vectorRaro = new Array();
vectorRaro["A colocar en los bookmark"] = "HTML en castellano";
```

Hacer una matriz bidimensional es más complicado, ya que tenemos que hacer un bucle que cree un vector nuevo en cada elemento del vector original.

Operadores

Los operadores nos permiten unir identificadores y literales para formar expresiones. Las expresiones son el resultado de operaciones matemáticas o lógicas. Un literal o una variable son expresiones, pero también lo son esos mismos literales y variables unidos entre sí mediante operadores.

JavaScript dispone de muchos más operadores que la mayoría de los lenguajes, si exceptuamos a sus padres C, C++ y Java. Algunos de ellos no los estudiaremos debido a su escasa utilidad y con algunos otros (especialmente el condicional) deberemos andarnos con cuidado, ya que puede lograr que nuestro código no lo entendamos ni nosotros.

Operadores aritméticos

JavaScript dispone de los operadores aritméticos clásicos y algún que otro más:

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Multiplicación	*	2*4	8
División	/	5/2	2.5
Resto de una división entera	%	5 % 2	1
Suma	+	2+2	4
Resta	-	7-2	5
Incremento	++	++2	3
Decremento	--	--2	1
Menos unario	-	-(2+4)	-6

Los operadores de incremento y decremento merecen una explicación auxiliar. Se pueden colocar tanto antes como después de la expresión que deseemos modificar pero sólo devuelven el valor modificado si están delante. Me explico.

```
a = 1;
b = ++a;
```

En este primer caso, `a` valdrá 2 y `b` 2 también. Sin embargo:

```
a = 1;
```

```
b = a++;
```

Ahora, a sigue valiendo 2, pero b es ahora 1. Es decir, estos operadores modifican siempre a su operando, pero si se colocan detrás del mismo se ejecutan después de todas las demás operaciones.

Operadores de comparación

Podemos usar los siguientes:

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Igualdad	==	2 == '2'	Verdadero
Desigualdad	!=	2 != 2	Falso
Igualdad estricta	===	2 === '2'	Falso
Desigualdad estricta	!==	2 !== 2	Falso
Menor que	<	2 < 2	Falso
Mayor que	>	3 > 2	Verdadero
Menor o igual que	<=	2 <= 2	Verdadero
Mayor o igual que	>=	1 >= 2	Falso

La igualdad y desigualdad estricta son iguales a las normales pero hacen una comprobación estricta de tipo. Han sido incluidos en el estándar ECMAScript y lo soportan Netscape 4.06 y superiores y Explorer 3 y superiores. Hay que indicar que versiones más antiguas de Netscape tratan la igualdad normal como si fuera estricta.

Operadores lógicos

Estos operadores permiten realizar expresiones lógicas complejas:

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Negación	!	!(2 = 2)	Falso
Y	&&	(2 = 2) && (2 >= 0)	Verdadero
Ó	||	(2 = 2) || (2 <> 2)	Verdadero

Operadores de asignación

Normalmente los lenguajes tienen un único operador de asignación, que en JavaScript es el símbolo =. Pero en este lenguaje, dicho operador se puede combinar con operadores aritméticos y lógicos para dar los siguientes:

Operador	Significado	Operador	Significado
x += y	x = x + y	x -= y	x = x - y
x /= y	x = x / y	x *= y	x = x * y
x % y	x = x % y		

Operadores especiales

Vamos a incluir en este apartado operadores que no hayan sido incluidos en los anteriores. La concatenación de cadenas, por ejemplo, se realiza con el símbolo +. El operador condicional tiene esta estructura:

```
condicion ? valor1 : valor2
```

Si la condición se cumple devuelve el primer valor y, en caso contrario, el segundo. El siguiente ejemplo asignaría a la variable a un 2:

```
a = 2 > 3 ? 1 : 2
```

Como podéis ver no resulta muy legible. Huid de este operador como de la peste. Evitad la tentación. Procurad no usarlo. Yo he tenido que descifrar código escrito por mí un par de meses antes que tenía esta clase de operadores, a veces incluso anidados. Todavía tengo escalofríos al recordarlo.

Para tratar con objetos disponemos de tres operadores:

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Crear un objeto	new	a = new Array()	a es ahora un vector
Borrar un objeto	delete	delete a	Elimina el vector anteriormente creado
Referencia al objeto actual	This		

`this` se suele utilizar en el código de los métodos de un objeto para referirse a otros métodos o a propiedades de su mismo objeto.

Estructuras de control

Ningún programa es una secuencia lineal de instrucciones. En todo lenguaje de programación existen estructuras que nos permiten variar el orden de ejecución dependiendo de ciertas condiciones. Estas estructuras se pueden clasificar en dos grandes grupos: bifurcaciones condicionales y bucles.

Aparte de los dos tipos clásicos de estructuras de control, en JavaScript disponemos de algunas estructuras adicionales para facilitar el manejo de objetos. Dispone de algunas estructuras más de las que explicaremos en esta página (el soporte de etiquetas), pero debido a que no son más que un recuerdo de lenguajes desfasados y su utilidad resulta escasa he preferido no incluirlas.

Bifurcaciones condicionales

Una bifurcación condicional en una estructura que realiza una tarea u otra dependiendo del resultado de evaluar una condición. La primera que vamos a estudiar es la estructura `if...else`. Esta estructura es la más sencilla y antigua (es posible que se utilizara con los ábacos y todo...) de todas:

```
if (bso.compositor == "Manuel Balboa")
  alert(¡Hombre, una banda sonora española!');
else
  alert('Seguro que es una americanada');
```

Hay que indicar que el `else` es opcional..

La siguiente estructura bifurca según los distintos valores que pueda tomar una variable específica. Es la sentencia `switch`:

```
switch(directorPreferido) {
  case "John Ford":
    alert('Eso es tener buen gusto, sí señor');
    break;
  case "Joel Coen":
    alert('Parece que te gustan las cosas raras');
    break;
  default:
    alert('¿Y ese quien es?');
}
```

Hay que indicar que no es compatible con estándar ECMA y no es soportado por el Explorer 3.

Bucles

Un bucle es una estructura que permite repetir una tarea un número de veces, determinado por una condición. Para hacer bucles podemos utilizar las estructuras `while` y

`do...while`. Estos bucle iteran indefinidamente mientras se cumpla una condición. La diferencia entre ellas es que la primera comprueba dicha condición antes de realizar cada iteración y la segunda lo hace después:

```
var numero=0;
while (numero==1) {
  alert('Soy un while');
}
do {
  alert('Soy un do...while');
} while (numero==1);
```

En este caso solo veríamos aparecer un ventana diciendo que es un `do...while`. ¿Qué por qué? Veamos. El `while` comprueba primero si `numero` es igual a 1 y, como no lo es, no ejecutaría el código que tiene dentro del bucle. En cambio, el `do...while` primero ejecuta el código y luego, viendo que la condición es falsa, saldría. Hay que resaltar que `do...while` no pertenece al estándar y no es soportado por el Explorer 3.

En JavaScript, el bucle `for` es singularmente potente. No se reduce a casos numéricos como en muchos otros lenguajes sino que nos da mucha más libertad. Tiene la siguiente estructura:

```
for (inicio; condición; incremento)
  código
```

El código contenido en el bucle se ejecutará mientras la `condición` se cumpla. Antes de comenzar la primera iteración del bucle se ejecutará la sentencia `inicio` y en cada iteración lo hará `incremento`. La manera más habitual de usar estas posibilidades es, claro está, la numérica:

```
var numero = 4;
for (n = 2, factorial = 1; n <= numero; n++)
  factorial *= n;
```

Por último, hay que decir que la ejecución de la sentencia `break` dentro de cualquier parte del bucle provoca la salida inmediata del mismo. Aunque a veces no hay más remedio que utilizarlo, es mejor evitarlo para mejorar la legibilidad y elegancia del código (toma ya, la frase que me ha salido).

Estructuras de manejo de objetos

JavaScript dispone de dos bien distintas. La primera es el bucle `for...in`, que nos permitirá recorrer todas las propiedades de un objeto. Se usa principalmente con vectores. Por ejemplo:

```
var vector = [1, 2, 2, 5];
for (i in vector)
  vector[i] += 2;
```

Este ejemplo sumaría dos a todos los elementos del vector. Sin embargo, conviene tener cuidado ya que, de los navegadores de Microsoft, sólo la versión 5 lo soporta.

La otra estructura es `with`, que nos permitirá una mayor comodidad cuando tengamos que tratar con muyas propiedades de un mismo objeto. En lugar de tener que referirnos a todas ellas con un `objeto.propiedad` podemos hacer:

```
with (objeto) {
  propiedad1 = ...
  propiedad2 = ...
  ...
}
```

Que resulta más cómodo (tenemos que teclar menos) y legible.

Funciones

Incluso los programas más sencillos tienen la necesidad de dividirse. Las funciones son los únicos tipos de subprogramas que acepta JavaScript. Tienen la siguiente estructura:

```
function nombre(argumento1, argumento2,..., argumento n) {
  código de la funcion
}
```

Los parámetros se pasan por valor. Eso significa que si cambiamos el valor de un argumento dentro de una función, este cambio no se verá fuera:

```
function sumarUno(num) {
  num++;
}

var a = 1;
sumarUno(a);
```

En este ejemplo, `a` seguirá valiendo 1 después de llamar a la función. Esto tiene una excepción, que son las referencias. Cuando se cambia el valor de una referencia dentro de una función también se cambia fuera.

Para devolver un valor de retorno desde la función se utiliza la palabra reservada `return`:

```
function cuadrado(num) {
  return num * num;
}

a = cuadrado(2);
```

En este ejemplo, `a` valdrá 4.

Se pueden definir funciones con un número variable de argumentos. Para poder luego acceder a dichos parámetros dentro de la función se utiliza el vector `arguments`. Este ejemplo sumaría el valor de todos los parámetros:

```
function sumarArgumentos() {
  resultado = 0;
  for (i=0; i<arguments.length; i++)
    resultado += arguments[i];
  return resultado;
}
```

Funciones predefinidas

JavaScript dispone de las siguientes funciones predefinidas:

eval(cadena)

Ejecuta la expresión o sentencia contenida en la cadena que recibe como parámetro.

```
mensaje = 'Hola';
eval("alert('" + mensaje + "');");
```

Este ejemplo nos muestra una ventana con un saludo.

parseInt(cadena [, base])

Convierte en un número entero la cadena que recibe, asumiendo que está en la base indicada. Si este parámetro falta, se asume que está en base 10. Si fracasa en la conversión devolverá el valor `NaN`.

```
parseInt("3453");
```

Devuelve el número 3453.

parseFloat(cadena)

Convierte en un número real la cadena que recibe, devolviendo `NaN` si fracasa en el intento.

```
parseFloat("3.12.3");
```

Este ejemplo devuelve `NaN` ya que la cadena no contiene un número real válido.

isNaN(valor)

Devuelve `true` sólo si el argumento es `NaN`.

isFinite(numero)

Devuelve `true` si el número es un número válido y no es infinito.

Number(referencia)
String(referencia)

Convierten a número (o referencia) el objeto que se les pase como argumento.

Objetos

Un objeto es una estructura que contiene tanto variables (llamadas propiedades) como las funciones que manipulan dichas variables (llamadas métodos). A partir de esta estructura se ha creado un nuevo modelo de programación (la programación orientada a objetos) que atribuye a los mismos propiedades como herencia o polimorfismo. Como veremos, JavaScript simplifica en algo este modelo.

El modelo de la programación orientada a objetos normal y corriente separa los mismos en dos: clases e instancias. Las primeras son entes más abstractos que definen un conjunto determinado de objetos. Las segundas son miembros de una clase, poseyendo las mismas propiedades que la clase a la que pertenecen. En JavaScript esta distinción se difumina. Sólo tenemos objetos.

Propiedades y métodos

Por si acaso ya lo hemos olvidado, recordemos ahora cómo se accede a los métodos y propiedades de un objeto:

```
objeto.propiedad
objeto.metodo(parametros)
```

Creación mediante constructores

Vamos a aprender a hacer nuestros propios objetos. Ya habíamos visto como hacerlo por medio de literales, pero dado que es una solución bastante propietaria y poco flexible, estudiaremos el método normal de lograrlo mediante constructores.

Un constructor es una función que inicializa un objeto. Cuando creamos un objeto nuevo del tipo que sea, lo que hacemos en realidad es llamar al constructor pasándole argumentos. Por ejemplo, si creamos un objeto `Array` de esta manera:

```
vector = new Array(9);
```

En realidad, estamos llamando a un constructor llamado `Array` que admite un parámetro. Sería algo así:

```
function Array(longitud) {
  ...
}
```

Vamos a crear nuestro primero objeto. Supongamos que queremos codificar en JavaScript una aplicación que lleve nuestra biblioteca de libros técnicos, de esos de Informática. Para lograrlo, crearemos un objeto `Libro` que guarde toda la información de cada libro. Este sería el constructor:

```
function Libro(titulo, autor, tema) {
  this.titulo = titulo;
  this.autor = autor;
  this.tema = tema;
}
```

Como vemos, accederemos a las propiedades y métodos de nuestros objetos por medio de la referencia `this`. Ahora podemos crear y acceder a nuestros objetos tipo `Libro`:

```
miLibro = new Libro("JavaScript Bible", "Danny Goodman", "JavaScript");
alert(miLibro.autor);
```

Sencillo, ¿no? Sin embargo, para disfrutar de toda la funcionalidad de los objetos nos falta algo. Ese algo son los métodos. Vamos a incluir uno que nos saque una ventana con el contenido de las propiedades escrito:

```
function escribirLibro() {
  alert("El libro " + this.titulo + " de " + this.autor +
    " trata sobre " + this.tema);
}
```

Para incluirlo en nuestro objeto añadimos la siguiente línea a nuestra función constructora:

```
this.escribir = escribirLibro;
```

Y podremos acceder al mismo de la manera normal:

```
miLibro.escribir();
```

Herencia

Una de las capacidades más empleadas de la programación orientada a objetos es la herencia. **La herencia supone crear objetos nuevos y distintos que, aparte de los suyas propios, disponen de las propiedades y métodos de otro objeto, al que llamaremos padre.**

Vamos a ver en nuestro ejemplo cómo se puede aplicar. En la actualidad, en muchos libros de informática, se incluye un CD-ROM con ejemplos y a veces aplicaciones relacionadas con el tema del libro. Si quisiéramos tener también esa información nos sería difícil, ya que algunos libros tienen CD y otros no. Podríamos tener otro objeto, pero tendríamos que reproducir el código (al menos del constructor) y si cambiáramos algo del mismo en el objeto `Libro` también tendríamos que hacerlo en el nuevo. Lo mejor será crear un objeto que herede las características de `Libro` y añada otras nuevas.

```
function LibroConCD (titulo, autor, tema, ejemplos, aplicaciones) {
  this.base = Libro
  this.base(titulo, autor, tema);
  this.tieneEjemplos = ejemplos;
  this.tieneAplicaciones = aplicaciones;
}
```

El problema es que, ahora, para acceder a las propiedades de `Libro` tenemos que hacerlo con intermediarios:

```
miLibro = new LibroConCD("JavaScript Bible", "Danny Goodman",
  "JavaScript", true, true);
alert('El libro Javascript Bible de Danny Goodman trata sobre
Javascript');
```

Para poder tener más transparencia y llamar a las propiedades de `Libro` como si fueran de `LibroConCD`, tenemos que incluir, fuera del constructor, la siguiente asignación:

```
LibroConCD.prototype = new Libro;
```

Y ya podremos. `prototype` es una propiedad que viene por defecto en todos los objetos que creemos. Sólo admite un valor, así que los aficionados a la herencia múltiple se quedarán con las ganas.

Objetos predefinidos

JavaScript dispone de varios objetos predefinidos para acceder a muchas de las funciones normales de cualquier lenguaje, como puede ser el manejo de vectores o el de fechas. En algunos casos estaremos tratando con objetos aunque no nos demos cuenta, ya que los usos más habituales de los mismos disponen de abreviaturas que esconden el hecho de que sean objetos.

Este capítulo no pretende ser una referencia completa, sino un resumen de las propiedades y métodos más usados. Si quieres más información consulta el manual de referencia de JavaScript de Netscape.

Objeto Array

Como dijimos antes, este objeto permite crear vectores. Se inicializa de cualquiera de las siguientes maneras:

```
vector = new Array(longitud);
vector = new Array(elemento1, elemento2, ..., elementoN);
```

En el primer caso crearemos un vector con el número especificado de elementos, mientras que en el segundo tendremos un vector que contiene los elementos indicados y de longitud N. Para acceder al mismo debemos recordar que el primero elemento es el número cero.

El objeto `Array` tiene, entre otros, los siguientes métodos y propiedades:

length

Propiedad que contiene el número de elementos del vector.

concat(vector2)

Añade los elementos de `vector2` al final de los del vector que invoca el método, devolviendo el resultado. No funciona en Explorer 3 y no forma parte del estándar ECMA.

sort(funcionComparacion)

Ordena los elementos del vector alfabéticamente. Si se añade una función de comparación como parámetro los ordenará utilizando ésta. Dicha función debe aceptar dos parámetros y devolver 0 si son iguales, menor que cero si el primer parámetro es menor que el segundo y mayor que cero si es al revés.

```
function compararEnteros(a,b) {
  return a<b ? -1 : (a==b ? 0 : 1);
}
```

Usando esta función ordenaría numéricamente (y de menor a mayor) los elementos del vector.

Objeto Date

Este objeto nos permitirá manejar fechas y horas. Se invoca así:

```
fecha = new Date();
fecha = new Date(año, mes, dia);
fecha = new Date(año, mes, dia, hora, minuto, segundo);
```

Si no utilizamos parámetros, el objeto fecha contendrá la fecha y hora actuales, obtenidas del reloj del sistema. En caso contrario hay que tener en cuenta que los meses comienzan por cero. Así, por ejemplo:

```
navidad99 = new Date(1999, 11, 25)
```

El objeto `Date` dispone, entre otros, de los siguientes métodos:

getTime()
setTime(milisegundos)

Obtienen y ponen, respectivamente, la fecha y la hora tomados como milisegundos transcurridos desde el 1 de enero de 1970.

getYear()
setYear(año)

Obtienen y poner, respectivamente, el año de la fecha. Éste se devuelven como números de 4 dígitos excepto en el caso en que estén entre 1900 y 1999, en cuyo caso se

devolverán las dos últimas cifras. Hay que tener cuidado, ya que la implementación de éstos métodos puede variar en las últimas versiones de Netscape.

getFullYear()
setFullYear(año)

Realizan la misma función que los anteriores, pero sin tantos líos, ya que siempre devuelven números con todos sus dígitos. Funciona en Explorer 4 y Netscape 4.06 y superiores.

getMonth()
setMonth(mes)
getDate()
setDate(dia)
getHours()
setHours(horas)
getMinutes()
setMinutes(minutos)
getSeconds()
setSeconds(segundos)

Obtienen y ponen, respectivamente, el mes, día, hora, minuto y segundo de la fecha, también respectivamente, respectivamente hablando.

getDay()

Devuelve el día de la semana de la fecha en forma de número que va del 0 (domingo) al 6 (sábado).

Objeto Math

Este objeto no está construido para que tengamos nuestras variables `Math`, sino como un contenedor donde meter diversas constantes (como `Math.E` y `Math.PI`) y los siguientes métodos matemáticos:

Método	Descripción	Expresión de ejemplo	Resultado del ejemplo
abs	Valor absoluto	Math.abs(-2)	2
sin, cos, tan	Funciones trigonométricas, reciben el argumento en radianes	Math.cos(Math.PI)	-1
asin, acos, atan	Funciones trigonométricas inversas	Math.asin(1)	1.57
exp, log	Exponenciación y logaritmo, base E	Math.log(Math.E)	1
ceil	Devuelve el entero más pequeño mayor o igual al argumento	Math.ceil(-2.7)	-2
floor	Devuelve el entero más grande menor o igual al argumento	Math.floor(-2.7)	-3
round	Devuelve el entero más cercano o igual al argumento	Math.round(-2.7)	-3
min, max	Devuelve el menor (o mayor) de sus dos argumentos	Math.min(2,4)	2
Pow	Exponenciación, siendo el primer argumento la base y el segundo el exponente	Math.pow(2,4)	16
Sqrt	Raíz cuadrada	Math.sqrt(4)	2

Objeto Number

Al igual que en el caso anterior, no se pueden crear objetos de tipo `Number`, sino que debemos referirnos al genérico. Este objeto contiene como propiedades los siguientes valores numéricos

Propiedad	Descripción
NaN	Valor que significa "no es un número"
MAX_VALUE y MIN_VALUE	Máximo y mínimo número representable
NEGATIVE_INFINITIVE y POSITIVE_INFINITIVE	Infinito negativo y positivo, se utilizan cuando hay desbordamiento al realizar alguna operación matemática

Objeto String

Este es un objeto que se puede confundir con los datos normales de tipo cadena. Conviene utilizar estos últimos, ya que los objetos `String` tienen un comportamiento extraño cuando se utilizan como cadenas normales. Además, al crear una cadena estamos creando a la vez un objeto `String` asociado. Su utilidad está en sus métodos, entre los que cabe destacar:

charAt(pos)
charCodeAt(pos)

Devuelven el caracter o el código numérico del carácter que está en la posición indicada de la cadena. El último no funciona en Explorer 3.

indexOf(subcadena)

Devuelven la posición de la subcadena dentro de la cadena, o -1 en caso de no estar.

split(separador)

Devuelven un vector con subcadenas obtenidas separando la cadena por el carácter separador. No funciona en Explorer 3.

```
cadena = "Navidad,Semana Santa,Verano";
vector = cadena.split(",");
```

En el ejemplo, el vector tendrá tres elementos con cada una de las vacaciones de un escolar español normal.

concat(cadena2)

Devuelve el resultado de concatenar `cadena2` al final de la cadena. No funciona en Explorer 3 y no forma parte del estándar ECMA.

substr(indice, longitud)
substring(indice1, indice2)

Devuelven una subcadena de la cadena, ya sea cogiendo un número de caracteres a partir de un índice o pillando todos los caracteres entre dos índices.

toLowerCase()
toUpperCase()

Transforman la cadena a minúsculas y mayúsculas, respectivamente.

Objeto RegExp

Este objeto se utiliza para comprobar si las cadenas se ajustan a ciertos patrones formalizados como expresiones regulares. Funcionan en las versiones 4 y superiores de los dos navegadores. Resultan bastante complicadas de manejar, así que les dedicaremos próximamente un capítulo en exclusiva.

Objeto Boolean

Objeto creado para crear confusiones a los programadores. Se supone que sirve para meter en un objeto los valores lógicos de verdadero y falso pero resulta que no se pueden utilizar en condiciones de la manera habitual.

Objeto Function

Sirve para crear funciones de manera ilegible y para que las referencias (que son siempre referencias a objetos) puedan usarse con funciones como en el ejemplo que vimos. La manera de crear funciones es la siguiente:

```
funcion = new Function([arg1, arg2, ..., argN], codigo);
```

Por ejemplo:

```
esElMejor = new Function([cadena], "return cadena=='Daniel' ? true :
false");
```

Eventos

Un evento, como su mismo nombre indica, es algo que ocurre. Para que una rutina nuestra se ejecute sólo cuando suceda algo extraño deberemos llamarla desde un controlador de eventos. Estos controladores se asocian a un elemento HTML y se incluyen así:

```
<A HREF="http://home.netscape.com" onMouseOver="MiFuncion()">
```

Lista de eventos

Aquí tienes una pequeña guía de eventos definidos en JavaScript. Para más información, lee la guía de Netscape.

Evento	Descripción	Elementos que lo admiten
OnLoad	Terminar de cargarse una página	<BODY...> <FRAMESET...>
OnUnLoad	Salir de una página (descargarla)	<BODY...><FRAMESET...>
OnMouseOver	Pasar el ratón por encima	<A HREF..> <AREA...>
OnMouseOut	Que el ratón deje de estar encima	<A HREF..> <AREA...>
OnSubmit	Enviar un formulario	<FORM...>
OnClick	Pulsar un elemento	<INPUT TYPE="button, checkbox, link, radio"...>
OnBlur	Perder el cursor	<INPUT TYPE="text"...> <TEXTAREA...>
OnChange	Cambiar de contenido o perder el cursor	<INPUT TYPE="text"...> <TEXTAREA...>
OnFocus	Conseguir el cursor	<INPUT TYPE="text"...> <TEXTAREA...>
OnSelect	Seleccionar texto	<INPUT TYPE="text"...> <TEXTAREA...>

Como ejemplo, vamos a hacer que una ventana aparezca automáticamente en cuanto pasemos un cursor por encima de un elemento <A> (e impidiendo, de paso, que quien esté viendo la página pueda hacer uso del mismo).

eventos.html

```
<HTML>
<HEAD>
  <SCRIPT LANGUAGE="JavaScript">
  <!-- Los comentarios esconden el código a navegadores sin JavaScript
    function Alarma() {
      alert("No me pises, que llevo chanclas");
      return true;
    }
  // -->
  </SCRIPT>
```

```
</HEAD>
<BODY>
<A HREF="eventos.html" onMouseOver="Alarma()">
  Pasa por aquí encima
</A>
</BODY>
</HTML>
```

Definición mediante código

Hemos visto como declarar un controlador de eventos desde etiquetas HTML. Sin embargo, y desde las versiones 3 de Netscape y 4 de Explorer, existe otro modo de hacerlo mediante código.

Muchos objetos cuyas etiquetas HTML correspondientes permiten atributos que definen controladores de evento, permiten el acceso a dichos controladores por medio de propiedades con el mismo nombre. Por ejemplo, la página:

load1.html

```
<HTML>
<HEAD>
  <SCRIPT LANGUAGE="JavaScript">
  <!-- Los comentarios esconden el código a navegadores sin JavaScript
    function Alarma() {
      alert("Hola");
    }
  // -->
  </SCRIPT>
</HEAD>
<BODY onLoad="Saludo()">
...
</BODY>
</HTML>
```

Se puede reescribir como:

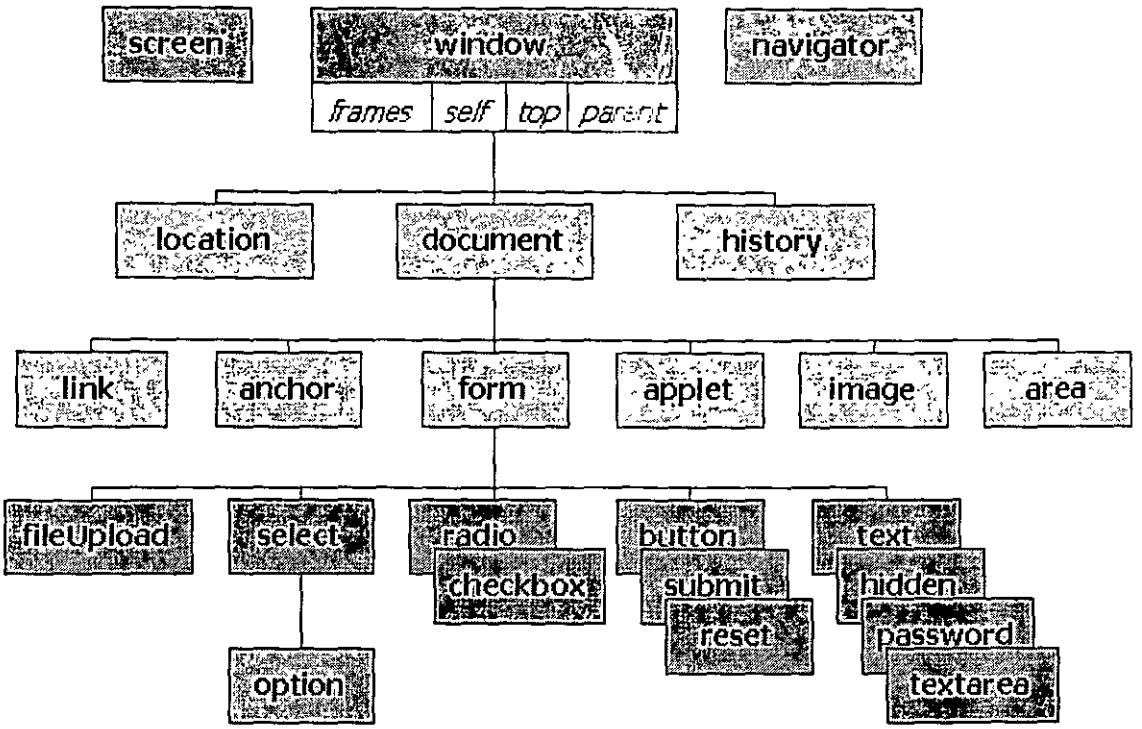
load2.html

```
<HTML>
<HEAD>
  <SCRIPT LANGUAGE="JavaScript">
  <!-- Los comentarios esconden el código a navegadores sin JavaScript
    function Saludo() {
      alert("Hola");
    }
    window.onload = Saludo;
  // -->
  </SCRIPT>
</HEAD>
<BODY>
...
</BODY>
</HTML>
```

Modelo de objetos del documento

Cuando funciona en un navegador, el lenguaje JavaScript dispone de una serie de objetos que se refieren a cosas como la ventana actual, el documento sobre el que trabajamos, o el navegador que estamos utilizando. En los próximos capítulos vamos a hacer un pequeño repaso de algunos de ellos con los métodos y propiedades más usados (los que tenga ganas de contaros, vamos).

La jerarquía de dichos objetos toma la siguiente forma:



Para los más iniciados en la programación orientada a objetos, conviene aclarar que en esta jerarquía, contra lo que pueda parecer, no existe herencia alguna. Los objetos se relacionan por composición, es decir, un objeto Window se compone (entre otras cosas) de un objeto Document, y éste a su vez se compone de diversos objetos Form, Image, etc..

El padre de esta jerarquía es el objeto Window, que representa una ventana de nuestro navegador. Dado que cada marco se considera una ventana distinta, cada uno de ellos dispone de su propio objeto Window. El objeto Document representa el documento HTML y cada uno de los objetos que lo componen se corresponden con diversas etiquetas HTML.

Objeto Window

Es el objeto principal. Define la ventana sobre la que estamos trabajando e incluye los objetos referentes a la barra de tareas, el documento o la secuencia de direcciones de la última sesión. En este capítulo veremos los métodos y propiedades más utilizadas, al menos por mí, dejando el estudio de dichos objetos para sus capítulos correspondientes.

Aún cuando el objeto se llame `Window`, disponemos siempre de una referencia a él llamada `window` (recordad que JavaScript distingue entre mayúsculas y minúsculas). Será con esa referencia con la que trabajemos. Este hecho será común a todos los objetos del DOM.

Por último, indicar que en JavaScript, se supone que todas las propiedades y métodos que llamamos sin utilizar ninguna referencia, en realidad se hacen utilizando utilizando esa referencia `window`. Así, por ejemplo, cuando ejecutamos el método `alert` en realidad lo que estamos haciendo es ejecutar el método `window.alert`.

[Variable=][window.]open(URL, nombre, propiedades)

> Permite crear (y abrir) una nueva ventana. Si queremos tener acceso a ella desde la ventana donde la creamos deberemos asignarle una variable, si no simplemente invocamos el método: el navegador automáticamente sabrá que pertenece al objeto `window`. El parámetro `URL` es una cadena que contendrá la dirección de la ventana que estamos abriendo: si está en blanco, la ventana se abrirá con una página en blanco. El

`nombre` será el que queramos que se utilize como parámetro de un `TARGET` y las `propiedades` son una lista separada por comas de algunos de los siguientes elementos:

- toolbar[=yes|no]
- location[=yes|no]
- directories[=yes|no]
- status[=yes|no]
- menubar[=yes|no]
- scrollbars[=yes|no]
- resizable[=yes|no]
- width=pixels
- height=pixels

Debemos tener cuidado con las propiedades que modifiquemos, es posible que algunas combinaciones de los mismos no funcionen en todos los navegadores. El Explorer 4, por ejemplo, da error ante la combinación `toolbar=no, directories=no, menubar=no`. Veamos un ejemplo:

ventanas.html

```
<HTML>
<HEAD>
  <SCRIPT LANGUAGE="JavaScript">
    function AbrirVentana() {
      MiVentana=open("","MiPropiaVentana",
        "toolbar=no,directories=no,menubar=no,status=yes");
      MiVentana.document.write(
        "<HEAD><TITLE>Una nueva ventana</TITLE></HEAD>");
      MiVentana.document.write("<CENTER><H1><B>" +
        "Esto puede ser lo que tu quieras</B></H1></CENTER>");
    }
  </SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" NAME="Boton1" VALUE="Abreme, Sésamo"
  onClick="AbrirVentana()">
</FORM>
</BODY>
</HTML>
```

Este ejemplo muestra la posibilidad de abrir nuevas ventanas de nuestro navegador con JavaScript. Se llama a la función `AbrirVentana` desde el evento `onClick`, como ya sabemos hacer desde el primer ejemplo. Esta función crea la nueva ventana `MiVentana` y escribe en ella por medio del objeto `Document` (tranquilos, en el próximo capítulo se estudiará este objeto) todo el código HTML de la página. Aquí debajo podéis probarlo.

close()

Cierra la ventana. A no ser que hayamos acabado de abrirla nosotros mostrará al usuario una ventana de confirmación para que decida él si quiere o no cerrarla. Esto se hace por motivos de seguridad, ya que sería demasiado fácil hacer un *script* de esos mal intencionados que nos cerrase la ventana del navegador, con lo que fastidia eso.

alert(mensaje)

Muestra una ventana de diálogo con el `mensaje` especificado.

confirm(mensaje)

Muestra una ventana de diálogo con el `mensaje` especificado y dos botones. Si el usuario pulsa OK, el método devuelve `true`. Si, en cambio, pulsa Cancelar, devolverá `false`.

prompt(mensaje, sugerencia)

Muestra una ventana de diálogo con el `mensaje` especificado y un campo de texto en el que el usuario pueda teclear, cuyo valor inicial será igual a `sugerencia`. Si el usuario pulsa OK, el método devuelve la cadena introducida en el campo de texto. Si, por el contrario, pulsa Cancelar, devolverá el valor `null`. Dado que este valor se considera igual a `false`, podemos comprabarlo directamente en una condición para ver que ha hecho el usuario.

Por ejemplo, el siguiente código muestra un saludo sólo si el usuario ha pulsado el botón de Aceptar:

```
var contestacion = prompt("¿Cómo te llamas, criatura?", "");
if (contestacion)
  alert("Hola, " + contestacion);
```

status

Define la cadena de caracteres que saldrá en la barra de estado en un momento dado.

defaultStatus

Define la cadena de caracteres que saldrá por defecto en la barra de estado. Cuando no la especificamos, `defaultStatus` será igual al último valor que tomó `status`.

setTimeout("función",tiempo)

Llama a `función` cuando hayan pasado `tiempo` milisegundos. Imprescindible a la hora de realizar cualquier rutina que deba ejecutarse a cierta velocidad.

La barra de estado

En muchas páginas web se puede observar cómo sus creadores controlan por completo la barra de estado del navegador. Quizá también hayas visto esos scrolls tan bonitos, y que tan rápido cansan. Ahora voy a explicar cómo se hacen ambas cosas, utilizando los que ya hemos visto del objeto Window.

Para empezar vamos con lo más sencillito: escribir mensajes diversos en la barra de estado. En el capítulo en que hablábamos sobre los objetos predefinidos aparecía el objeto `window`. En este objeto se definían dos atributos: `status` y `defaultStatus`. Para poner un mensaje en la barra de estado nada más cargar el documento y que se mantenga ahí mientras no haya otro más importante (un sustituto del famoso `Document: Done` del Netscape, vamos) haremos:

```
<BODY onLoad=
  "window.defaultStatus='El documento ya se ha leído';return true">
```

El código lo único que hace es modificar `defaultStatus` y devolver true como resultado del controlador de eventos. Por alguna misteriosa razón es obligatorio hacer esto cuando modificas algo de la barra de estado. No me preguntéis por qué, al parecer no es más que una convención.

Ahora veremos cómo se puede definir el valor de la barra de estado cuando el ratón pasa por encima de un elemento `<A>`:

```
<A HREF="MiPagina.html" onMouseOver=
  "window.status='Vente a mi pásgina';return true">
```

¿Fácil, no? Bueno, ahora vamos a ver cómo se hacen scrolls.

scrolls.html

```
<HTML>
<HEAD>
  <TITLE>Ejemplo</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
    var texto="   Aquí está el mensaje que espero " +
      "observes y leas con suma atención   ";
    var longitud=texto.length;
    function scroll() {
      texto=texto.substring(1,longitud-1)+texto.charAt(0);
      window.status = texto;
      setTimeout("scroll()",150);
      }
  </SCRIPT>
</HEAD>
<BODY onLoad="scroll();return true;">
Esta es la mejor página del mundo conocido.
</BODY>
</HTML>
```

Como puedes ver, la cosa no es ni más larga ni más compleja que los ejemplos anteriores. Simplemente escribe el texto en la barra de estado y luego coge el caracter más a la izquierda del mismo y lo pone a la derecha, para después volver a escribirlo. La única pega que tiene es cómo demonios hago para que la función se llame cada cierto tiempo predeterminado para ir desplazando el texto a una velocidad constante. Y la respuesta esta en el método `setTimeout`.

Frames

Uno de los problemas más frecuentes a los que se enfrenta un programador de JavaScript es el manejo de frames. Es decir, ¿cómo accedo yo al código o a objetos como `Window` o `Document` de otros frames? JavaScript propone una manera bastante sencilla de hacerlo.

JavaScript considera el documento de declaración de marcos (es decir, aquel en el que escribimos las etiquetas `FRAME` y `FRAMESET`) como un objeto `Window` normal y corriente. Permite acceder a los marcos que hemos declarado en él por medio del vector `frames`. Es decir, si en nuestro documento estuviera la siguiente línea:

```
<FRAME NAME="principal" SRC="MiPagina.html">
```

Podríamos acceder a su objeto `Window` por medio de la referencia `window.frames["principal"]`.

A su vez, desde el documento "hijo", es decir, desde el documento que está encerrado en un frame, podemos acceder al padre por medio de la referencia `parent`. También podemos acceder al documento que esté arriba del todo en esta jerarquía por medio de `top`. Por ejemplo:

```
window == window.top
```

Esta igualdad comprobará si nuestro documento está en la ventana principal o en un frame. Comprueba si la ventana en la que está (`window`) es igual a la ventana principal (`window.top`). Mediante esta sencilla comprobación podemos crear fácilmente (toda vez que conozcamos el manejo del objeto `Location`), rutinas que aseguren que nuestra ventana es la principal, o que recarguèn nuestra estructura de marcos si algún usuario pretende acceder a un frame específico, etc.

Objeto Document

Este objeto representa el documento HTML en el que estamos. Se accede a él por medio de la referencia `document`. Su mayor importancia viene por el número de vectores que posee, que referencian a objetos `Image`, `Form` o `Link`, los cuales permiten acceder a las imágenes, formularios y enlaces del documento, respectivamente.

También dispone de varios métodos para trabajar con cookies, pero eso lo veremos en otro capítulo.

lastModified

Contiene la fecha y hora en que se modificó el documento por última vez y se suele usar en conjunción con `write` para añadir al final del documento estas características.

bgColor

Modifica el color de fondo del documento. El color deberá estar en el formato usado en HTML. Es decir, puede ser `red` o `FF0000`.

forms[]

Vector que contiene los formularios del documento. El primero será el número 0, el segundo el 1, etc.

images[]

Vector que contiene las imágenes del documento. Se ordenan igual que en el anterior caso, aunque también permiten ser accedidas con el nombre como índice. Es decir, a una imagen definida como `<IMG SRC=".." NAME="miImagen">` se puede acceder con `document.inages["miImagen"]`.

links[]

Vector que contiene los enlaces del documento. Se ordenan igual que en los dos anterioers, aunque no se suele utilizar en el código Javascript. Su razón de ser es que, al ser los enlaces objetos, permiten incluir código Javascript en ellos por medio de la pseudo-URL `javascript:codigo`.

write(cadena)
writeln(cadena)

Escribe el código HTML indicado en `cadena` en nuestro documento HTML. `writeln` hace lo mismo, pero incluyendo al final un retorno de carro.

open()

Abre un nuevo documento para escribir. Un documento debe estar abierto para poder escribir en él. Sin embargo, no se utiliza mucho ya que los dos métodos anteriores abren automáticamente el documento si no lo está ya.

close()

Cierra el documento, impidiendo escribir de nuevo en él.

Cómo escribir en el documento

A la hora de escribir en un documento por medio de `write` o `writeln` hay que tener en cuenta un hecho fundamental. Para poder escribir en un documento, éste debe estar abierto y, **al abrirlo, se destruye todo el código que haya en él**.

Un documento se abre automáticamente cuando empieza a cargarse y se cierra cuando termina de hacerlo. Así pues, si deseamos escribir en un documento si sobreescribir su contenido, deberemos hacerlo antes de que éste termine de cargar. Si lo hacemos después, sobreescribiremos su contenido.

Por lo tanto:

escribir.html

```
<HTML>
<HEAD>
  <TITLE>Escribe y no sobreescribe</TITLE>
</HEAD>
```

```
<BODY>
Este es un documento muy interesante y que fue modificado por última vez
el día
<SCRIPT LANGUAGE="JavaScript">
document.write(document.lastModified);
</SCRIPT>
</BODY>
</HTML>
```

Este ejemplo os mostrará la cadena completa, ya que se llama a `write` antes de cerrarse el documento. Es decir, antes de que termine de cargar. Sin embargo, el siguiente ejemplo:

sobreescribir.html

```
<HTML>
<HEAD>
  <TITLE>Sobreescribe</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
    function escribir() {
      document.write("Este es un documento muy interesante y " +
        "que fue modificado por última vez el día");
      document.write(document.lastModified);
    }
  </SCRIPT>
</HEAD>
<BODY onLoad="escribir()">
Hola.
</BODY>
</HTML>
```

Sobreescribirá el documento y no podremos ver ese "Hola", al ser llamado después de cargar el documento, es decir, después de cerrarlo.

Cookies

Parece más que necesario definir algo que lleva el absurdo nombre de cookie (literalmente: galletita). Al parecer, en Estados Unidos (ignoro si también en Gran Bretaña) reciben ese nombre las fichas que dan en los guardarropas a modo de resguardo. Y es un símil bastante acertado de lo que son en realidad las cookies cuando hablamos de Internet.

Una cookie es un elemento de una lista que se guarda en el ordenador del visitante. Cada elemento de esa lista tiene dos campos obligatorios: el nombre y su valor; y uno opcional: la fecha de caducidad. Este último campo sirve para establecer la fecha en la que se borrará la galleta. Tiene este formato:

```
nombre=valor;[expires=caducidad;]
```

Sólo el servidor que ha enviado al usuario una determinada cookie puede consultarla. Cada una tiene un tamaño máximo de 4 Kb y puede haber un máximo de 300 cookies en el disco duro. Cada servidor podrá almacenar como mucho 20 galletas en el fichero `cookies.txt` (en el caso de usar Netscape) o en el directorio `cookies` (si utilizamos Explorer) del usuario. Si no especificamos la fecha de caducidad la "galleta" se borrará del disco duro del usuario en cuanto éste cierre el navegador.

Funciones básicas

Para poder hacer algo con cookies deberemos programar dos funciones: una que se encargue de mandar una cookie al usuario y otra que consulte su contenido.

```
function mandarGalleta(nombre, valor, caducidad) {
  document.cookie = nombre + "=" + escape(valor)
    + ((caducidad == null) ? "" : ("; expires=" +
caducidad.toGMTString()))
}
```

Con esta función mandamos una galleta. Vemos que el valor es codificado por medio de la función `escape` y que la caducidad (en caso de decidir ponerla) debe ser convertida a formato GMT. Esto se hace mediante el método `toGMTString()` del objeto `Date`.

```
function consultarGalleta(nombre) {
  var buscamos = nombre + "=";
  if (document.cookie.length > 0) {
    i = document.cookie.indexOf(buscamos);
    if (i != -1) {
      i += buscamos.length;
      j = document.cookie.indexOf(";", i);
      if (j == -1)
        j = document.cookie.length;
      return unescape(document.cookie.substring(i,j));
    }
  }
}
```

Declaramos la variable buscamos que contiene el nombre de la cookie que queremos buscar más el igual que se escribe justo después de cada nombre, para que así no encontremos por error un valor o una subcadena de otro nombre que sea igual al nombre de la cookie que buscamos. Una vez encontrada extraemos la subcadena que hay entre el igual que separa el nombre y el valor y el punto y coma con que termina dicho valor.

Un ejemplo: el contador individualizado

Vamos a ver un ejemplo. Utilizaremos una galleta llamada `VisitasAlCursoDeJavaScript` para guardar el número de veces que has visitado este curso:

cookies.txt

```
<HTML>
<HEAD>
  <SCRIPT LANGUAGE="JavaScript">
  <!-- Comentario para esconder el código a navegadores sin JavaScript
    function Contador() {
      var fecha=new Date (2004, 12, 31);
        // La fecha de caducidad es 31 de diciembre del 2004
      if (!(num=consultarGalleta("VisitasAlCursoDeJavaScript")))
        num = 0;
      num++;
      mandarGalleta("VisitasAlCursoDeJavaScript", num, fecha);
      if (num==1)
        document.write("esta es la primera vez que lees este capítulo.");
      else {
        document.write("has visitado este curso "+num+" veces.");
      }
    }
  // -->
  </SCRIPT>
</HEAD>
<BODY>
Por lo que veo,
<SCRIPT LANGUAGE="JavaScript">
<!--
  Contador();
// -->
</SCRIPT>
</BODY>
</HTML>
```

La función consulta el valor de la cookie incrementandolo y, si no existe, lo pone a uno. Luego escribe en el documento el número de veces que has visitado el curso. Y, por lo que veo, esta es la primera vez que lees este capítulo.

Objeto Image

Este objeto representa a una imagen. Se puede acceder a las diversas imágenes del documento por medio del vector de referencias `document.images[]`. Este objeto está disponible desde el Netscape 3 y el Explorer 4. Sus propiedades más importantes son:

src

Contiene el archivo de la imagen.

lowsrc

Los navegadores admiten incluir un nombre de imagen que se cargue antes que la imagen final. Es útil para poner una imagen que ocupe poco mientras esperamos que se cargue la imagen final, que puede ocupar una cantidad insultante de espacio. Esta propiedad permite accede al archivo de esa primera imagen pequeña.

complete

Valor lógico que será `true` si la imagen ha terminado ya de cargarse.

También dispone de diversas propiedades de sólo lectura que se corresponden con los atributos de la etiqueta `<IMG>` como, por ejemplo, `width` o `border`.

Tratamiento de imágenes

Lo que vamos a lograr con este truco es un pequeño cambio de imágenes. En muchas páginas, al pasar el ratón por encima de una imagen puedes observar que el gráfico cambia. Para hacer esto deberemos crear dos gráficos distintos: el que se verá normalmente y el que únicamente podrá verse cuando el ratón pase por encima. Si llamamos al primero, por ejemplo, `apagado.gif` y al segundo `encendido.gif` el código necesario para que el truco funcione es:

imagenes.html

```
<HTML>
<HEAD>
  <TITLE>Ejemplo de imagenes</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
    if (document.images) {
      var activado=new Image();
      activado.src="encendido.gif";
      var desactivado= new Image();
      desactivado.src="apagado.gif";
    }
    function activar(nombreImagen) {
      if (document.images) {
        document[nombreImagen].src=activado.src; }
    }
    function desactivar(nombreImagen) {
      if (document.images) {
        document[nombreImagen].src=desactivado.src; }
   }
  </SCRIPT>
</HEAD>
<BODY>
<A HREF="mipagina.html" onMouseOver="activar('prueba');"
  onMouseOut="desactivar('prueba');">
  <IMG NAME="prueba" SRC="apagado.gif" BORDER=0>
</A>
</BODY>
</HTML>
```

Lo primero que hay que indicar es que para que funcione el invento la imagen debe ser un enlace. ¿Por qué? Porque los eventos que controlan si el ratón pasa o no por encima no son

admitidos por la etiqueta <IMG>. También deberemos "bautizar" nuestra imagen usando el atributo NAME="como-se-llame" para permitir al código su identificación posterior.

El ejemplo funciona de la siguiente manera: en principio la imagen que vemos es la desactivada, que es la que indica la etiqueta <IMG>. Al pasar el raton por encima de ella el evento onMouseOver llamará a la función activar llevando como parámetro el nombre de la imagen. Esta función sustituirá en el objeto document el nombre del fichero donde se guarda la imagen por encendido.gif, que es el gráfico activado. Cuando apartemos el ratón de la imagen, será el evento onMouseOut el que se active, llamando a desactivar. Esta función sustituirá el gráfico de nuevo, esta vez por apagado.gif.

Leyendo esta explicación parece que una parte del código sobra. ¿Para qué sirve declarar dos objetos Image para albergar los gráficos? ¿No bastaría con cambiar directamente el nombre de la imagen escribiendo document[nombreImagen].src = 'encendido.gif';? Pues no del todo. Efectivamente funcionaría, pero cada vez que cambiásemos el nombre, el navegador se traería la imagen del remoto lugar donde se encontrara. Es decir, cada vez que pasásemos el ratón por encima de la imagen o nos alejaramos de ella tendríamos que cargar (aunque ya lo hubiésemos hecho antes) el gráfico correspondiente. Es cierto que con la caché del navegador este efecto quedaría algo mitigado, pero siempre es mejor precargar las imágenes usando el objeto Image, ya que así el navegador comenzará a leer el gráfico en cuanto ejecute el código en vez de esperar a que el usuario pase por encima de la imagen con el ratón. El objeto Image tiene como atributos los distintos parámetros que puede tener la etiqueta <IMG>.

Por último, hay que estudiar que significa eso de if (document.images). Los navegadores en que no exista el objeto Image darán un mensaje de error si se lo encuentran por ahí. La solución a este problema consiste en detectar la capacidad del navegador para manipular gráficos preguntándole por la existencia del vector document.images. Así podremos no crear las variables que guardan los gráficos ni ejecutar el código de las funciones para activar y desactivar en el caso de que el navegador no soporte este vector.

Objeto Form

Los formularios siempre han sido la mejor manera de facilitar la comunicación entre los usuarios y los creadores de una web. Sin embargo, la implementación de los mismos en el lenguaje HTML ha provocado ciertos problemas debido a sus carencias. Estos problemas han intentado solventarse con scripts, situados tanto en el servidor como en el navegador.

Los programas albergados en el servidor suelen ser scripts CGI y, por supuesto, no vamos a investigarlos en un curso de JavaScript. Solamente decir que existen infinidad de ellos y que, en general, los buenos proveedores de espacio web tienen alguno disponible para sus usuarios, con instrucciones de uso incluidas.

Los programas que se ejecutan en el navegador suelen estar escritos en JavaScript, y realizan tareas simples de validación. En muchas ocasiones están combinados con scripts CGI que modifican el mensaje generado por un formulario para facilitar su lectura y manejo.

El vector document.forms contiene todos los formularios de un documento. Así, se accedería al primer formulario definido como document.forms[0]. Sin embargo, si usamos el parámetro NAME en la etiqueta HTML:

```
<FORM NAME="miFormulario">
```

entonces podremos acceder al formulario con document.miFormulario, que resulta bastante más cómodo y estable ante la posibilidad de variación del número y posición de formularios en el documento. Estos son los métodos y propiedades del objeto Form:

submit()

Envía el formulario.

reset()

Devuelve los valores de un formulario a su estado inicial.

elements[]

Contiene todos y cada uno de los elementos de los que consta el formulario, es decir, los botones, cajas de textos, listas desplegables, etc. que componen un formulario. Cada elemento puede ser un objeto distinto, por lo que deberemos averiguar de qué tipo son por medio de la propiedad común `type`.

Objetos Text

Cuatro elementos HTML distintos (text, textarea, password y hidden) son, desde el punto de vista del DOM, objetos tan parecidos entre sí que vamos a estudiarlos conjuntamente. En realidad, sólo tienen tres propiedades verdaderamente importantes:

name

Nombre del elemento, indicado en el atributo `NAME` de su etiqueta HTML. Este atributo está presente en todos los objetos que son elementos de un formulario.

type

Tipo del elemento. Al igual que el anterior, esta propiedad está presente en todos los objetos que representan elementos de un formulario. En el caso de los que nos ocupan valdrá siempre `"text"`.

value

Contiene el valor, es decir, el texto tecleado por el usuario. Es a esta propiedad a la que accederemos habitualmente.

Como ejemplo de su uso, vamos a ver ahora el código de nuestro primer formulario con validación, que comprueba si la dirección de correo electrónico que introduce el usuario es correcta:

formularios.html

```
<HTML>
<HEAD>
  <TITLE>Ejemplo de formularios</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
    function validar(direccion) {
       if (direccion.indexOf("@") != -1)
          return true;
       else {
          alert('Debe escribir una dirección válida');
          return false;
       }
    }
  </SCRIPT>
</HEAD>
<BODY>
<FORM NAME="miFormulario"
     METHOD="POST"
     ACTION="mailto:yo@miproveedor.mipais"
     ENCTYPE="text/plain"
     onSubmit="return validar(this.email.value)">
Mandame tu e-mail: <INPUT NAME="email" TYPE="text"><BR>
<INPUT TYPE="submit" VALUE="Mandame tu e-mail">
</FORM>
</BODY>
</HTML>
```

El código es sencillo: el código llamado por el controlador de evento `onSubmit` debe devolver `false` si deseamos que el formulario no sea enviado. Así pues, llamamos a la función que comprueba si hay alguna arroba en el campo `email` del formulario.

La manera de llamar a esta función es quizás lo más complicado. La función `validar` recibe una cadena de caracteres, devolviendo verdadero o falso dependiendo de que haya o no una arroba dentro de ella. El controlador utiliza para llamar a esta función lo siguiente:

```
this.email.value
```

`this` es una referencia estándar. Cuando veamos `this` en algún código en realidad deberemos sustituirlo mentalmente por el nombre del objeto en el que está el código. En este caso, como estamos dentro de `miFormulario`, `this` será equivalente a `document.miFormulario`. `email` es el nombre del campo que queremos investigar y `value` el el atributo que contiene lo que haya tecleado el usuario.

Mándame tu e-mail: Mándame tu e-mail

Objetos Checkbox y Radio

En estos objetos, tanto el atributo HTML `VALUE` como su correspondiente propiedad `value` accesible desde JavaScript no corresponden a nada visible. En ambos casos, el estado del elemento es de tipo lógico: puede estar seleccionado o no. Este valor lógico se contiene en la propiedad `checked`.

Por tanto, para comprobar si está pulsado o no una caja de confirmación o un botón de selección específico deberemos preguntar por dicha propiedad.

Vamos a ver un ejemplo práctico de uso: os voy a hacer un pequeño examen, para ver cuanto JavaScript haz aprendido. Espero que ningún profesor utilice este ejemplo para realizar sus exámenes, ya que cualquiera puede averiguar las respuestas observando el código. Aquí tienes el examen:

La estructura a = b ? c : d; es...

- un bucle
- una operación aritmética
- una condición

El atributo `window.status` contiene:

- el valor de la barra de estado
- el valor por defecto de la barra de estado
- ciertas características de la ventana

El evento *load* ocurre:

- cuando termina de cargarse una página
- cuando termina de cargarse un gráfico
- cuando empieza a cargarse una página

¿Que *array* contiene los gráficos de una página?

- document.graphics

- document.images
- no hay ninguno

El método `alert` sirve para:

- hacer sonar un pitido de alarma
- lanzar una ventana con información
- decirle al navegador que hay un problema con el código

Quiero saber mi nota

Comprueba que cada elemento del formulario de tipo `radio` seleccionado tiene el atributo `VALUE` igual a `"bien"`. Teniendo en cuenta que el vector `elements` contiene todos los elementos de un formulario, el código resultante es el siguiente:

```
function averiguarNota(examen) {
  var resultado=0;
  for (i=0;i<examen.elements.length;i++)
     if ((examen.elements[i].type=="radio") &&
        (examen.elements[i].value=="bien") &&
        (examen.elements[i].checked))
          resultado++;
  alert("Acertaste "+resultado+" de 5 preguntas.");
  return false;
}
```

y cada pregunta tiene la siguiente estructura:

```
<BR>HTML en castellano es
<BR><INPUT TYPE="radio" NAME="Respuesta1" VALUE="mal">
  una porquería
<BR><INPUT TYPE="radio" NAME="Respuesta1" VALUE="bien">
  el mejor
<BR><INPUT TYPE="radio" NAME="Respuesta1" VALUE="mal">
  algo poco reseñable
```

Luego la función es llamada desde el evento submit y ya está. Podría haberse hecho también desde el evento click de un botón... siempre hay más de una manera de hacer las cosas.

Objetos Select y Option

El objeto `Select` es, con mucho, el más complicado. Esto es debido a que contiene en su interior un vector de objetos `Option`. Dispone de dos propiedades interesatnes:

selectedIndex

Empezando a contar a partir de cero, indica qué opción está seleccionada. Si hay varias seleccionadas, indica la opción con el índice más bajo.

options[]

Vector que contiene los objetos `Option` correspondientes a todas y cada una de las opciones.

El objeto `Option`, por otro lado, dispone de otras dos propiedades a estudiar (aparte de las comunes, como `type` o `value`):

selected

Valor lógico que será verdadero si la opción está seleccionada.

text

Texto que acompaña a la opción.

Como ejemplo, vamos a ver una lista desplegable que nos permita navegar por diversas páginas. Cada etiqueta `OPTION` tendrá como parámetro `VALUE` la dirección de la página web e incluiremos un controlador del evento `onChange` (que se ejecuta cuando el usuario cambia la opción escogida en la lista) que llamará a una rutina que explicamos más adelante:

select.html

```
<HTML>
<HEAD>
  <TITLE>Ejemplo de Select</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
    function irA(menu){
      window.location.href = menu.options[menu.selectedIndex].value;
    }
  </SCRIPT>
</HEAD>
<BODY>
<FORM name="formulario">
  <SELECT NAME="menu" SIZE=1 onChange ="irA(this)">
    <OPTION VALUE="">Visitar
    <OPTION VALUE="http://www.ole.es">&iexcl;Ol&eacute;!
    <OPTION VALUE="http://www.ozu.es">Oz&uacute;
    <OPTION VALUE="http://www.ozu.com">Otro oz&uacute;
    <OPTION VALUE="http://www.es.lycos.de">Lycos
    <OPTION VALUE="http://www.metabusca.com">Metabusca
  </SELECT>
</FORM>
</BODY>
</HTML>
```

La función `irA` simplemente utiliza la opción elegida para obtener por medio de su valor la dirección de la página a la que debe acudir.

Otros objetos

El modelo de objetos del documento define varios objetos, por así decirlo, "menores", que no tienen relación con nada físico de la página en la que estamos. Es decir, no guardan relación con las etiquetas HTML que estén en ellas escritas. Son los siguientes:

Objeto History

Se accede a este objeto por medio de la referencia `document.history` y contiene todas las direcciones que se han visitado en la sesión actual. Aunque no permite acceder a ellas (para que no podamos cotillear demasiado al usuario), dispone de varios métodos para sustituir el documento actual por alguno que el usuario ya haya visitado:

back()

Volver a la página anterior. Es muy sencillo de utilizar:

```
<A HREF="javascript:window.history.back()">
```

Y, para variar, si deseas probar no tienes más que pulsar aquí.

forward()

Ir a la página siguiente.

go(donde)

Ir a `donde` se indique, siendo `donde` un número tal que `go(1)=forward()` y `go(-1)=back()`.

Objeto Location

Se accede a este objeto por medio de la referencia `document.location` y contiene información sobre la dirección de la página actual en varias propiedades.

href

Permite el acceso a la dirección de la página actual. Si lo cambiamos, pues cambiaremos de página.

protocol

Protocolo de la página actual. Habitualmente `http`.

host

Máquina donde se alberga la página actual. En el caso de la que estás leyendo, sería `html.programacion.net`.

pathname

Camino de la página actual. En nuestro ejemplo, será `/js/otrosobjetos.htm`.

hash

Si hemos accedido a una página por medio de un ancla, contiene una almohadilla seguida de ese ancla. Por ejemplo, `#location`.

search

Puede que hallas notado que muchas páginas (especialmente en los motores de búsqueda) tienen unas direcciones inmensas con una estructura como `pagina.asp?busqueda=HTML+en+castellano&tipo=Y` o engendros semejantes. Esta propiedad permite acceder a esa última parte de la dirección (a partir de la interrogación, inclusive). Puede ser útil para pasar parámetros de una página a otra.

Objeto Navigator

Se accede a él por medio de la referencia `navigator` y nos permite averiguar varias características del navegador que usamos. Por ejemplo:

appName

Nombre del navegador.

appVer

Número principal de versión.

language

Idioma del mismo.

platform

Plataforma donde esta ejecutándose.

No podemos sobreescribir estos atributos, pero sí leerlos.

Objeto Screen

Como cabía esperarse, se puede acceder a este objeto por medio de la referencia... chachaaaan `iscreen`! Nos permitirá conocer la configuración de la pantalla del usuario. Al igual que en el anterior objeto, todos sus atributos son de sólo lectura. Conviene indicar que este objeto sólo está disponible desde las versiones 4.0 de ambos navegadores.

height

Altura de la resolución de la pantalla.

width

Anchura de la resolución de la pantalla.

pixelDepth

Número de bits por pixel.

Así, por ejemplo, te puedo decir que en este momento tu pantalla está configurada para 800x600x4294967296 colores. Y el código que he utilizado para averiguarlo es el siguiente:

```
if (window.screen)
  texto=screen.width + "x" + screen.height + "x" +
    Math.pow(2,screen.colorDepth) + " colores.";
else
  texto="quien sabe cuantos colores, que necesito " +
    "que tengas Communicator o IE4 para averiguarlo.";
document.write(texto);
```

Puede verse que antes de acceder al objeto, investigo si éste existe, mostrando un mensaje de circunstancias en caso de que no sea así.

Nuevas características

Vamos a estudiar algunos elementos del lenguaje que no estaba definidos en su primera versión pero, tras ser introducidos en la versión 1.1, se han incorporado al estándar ECMAScript. Por ejemplo, en esta versión se define el método para incluir una expresión JavaScript en los parámetros de una etiqueta HTML. La forma de hacerlo es considerando dicha expresión (entre llaves) como un caracter especial HTML, encerrándolo, por tanto, entre un `&` y un `;`. Por ejemplo:

```
<HR WIDTH=&{anchoLinea+"%";};>
```

que, en el supuesto de que la variable `anchoLinea` sea igual a 50 nos daría el siguiente resultado:

Ficheros .js

Un fichero .js es un archivo donde podremos guardar funciones y variables globales que podrán leerse desde cualquier página HTML. Gracias a ellos podremos evitar el tener que duplicar funciones que se necesiten en más de un documento. Podremos incorporarlos a nuestras páginas de esta manera:

externo.html

```
<HTML>
<HEAD>
  <TITLE>Mi Página</TITLE>
  <SCRIPT LANGUAGE="Javascript" SRC="funciones.js">
  <!--
    alert('Error con el fichero js');
  // -->
  </SCRIPT>
</HEAD>
<BODY>
Lo que sea.
</BODY>
</HTML>
```

El código que incluyamos entre un <SCRIPT SRC> y un </SCRIPT> sólo se ejecutará en caso de que la lectura del fichero.js falle.

Por otra parte, hay que indicar que el fichero en cuestión contendrá sólo código en Javascript, no etiquetas HTML de ningún tipo, ni siquiera las de apertura y cierre de `SCRIPT`.