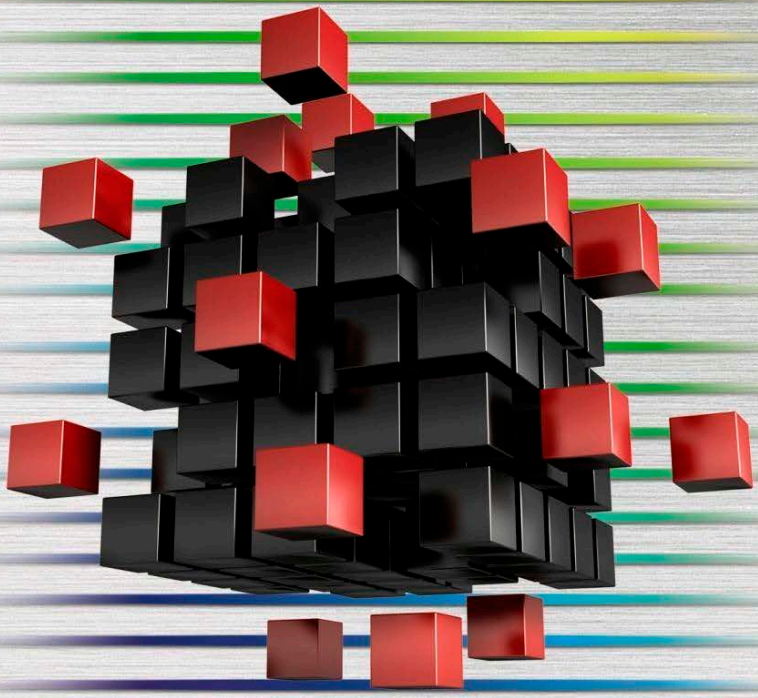
Guía Práctica

JavaScript

Edición 2012

Astor de Caso Parra



ANAYA MULTIMEDIA

JavaScript
Edición 2012

Astor de Caso Parra

ANAYA MULTIMEDIA

GUÍAS PRÁCTICAS

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S.A.), 2012
Juan Ignacio Luca de Tena, 15. 28027 Madrid
Depósito legal: M.38.836-2011
ISBN: 978-84-415-3048-5
Printed in Spain
Impreso en: Closas-Orcoyen, S.L.

"Aunque los perros no programan;
Yembo, formas parte de esta guía."

Agradecimientos

Gracias a todos los lectores de la anterior edición porque van a hacer posible que futuros nuevos lectores puedan aprender tanto como ellos. También quiero dar las gracias a Anaya por confiar en mí de nuevo para desarrollar esta nueva edición.

Por supuesto, gracias a mi familia y amigos más cercanos por apoyarme tanto o más que la vez anterior.

Índice

Introducción

¿De qué trata este libro?

Esta guía práctica está dirigida a personas que, como yo hace unos años, sienten la necesidad de hacer sus páginas Web más útiles o atractivas de lo que son ahora. El HTML a secas se queda pequeño cuando nos surge la necesidad de interactuar con el usuario que visita nuestra página para que nos envíe una sugerencia o sencillamente para que vea lo bonito que queda que los botones cambien de color al pasar sobre ellos. Por esta razón, ha decidido que era el momento de ir un paso más allá y aprender un lenguaje de programación Web como JavaScript comprando este libro, cosa que le agradezco desde este momento.

Mi intención no es llenarle la cabeza con tecnicismos que le dejen una sensación de vacío y de no haber aprendido nada, aunque hay ciertos términos que son imposibles de simplificar. Pese a ello, en la medida de lo posible, intentaré que todas las explicaciones sean claras y concisas, acompañándolas de un ejemplo si eso mejora la compresión.

Navegadores Web

Todo aquel que desarrolle sitios Web se encuentra ante un pequeño muro cuando visualiza el sitio en los distintos navegadores del mercado: las páginas no funcionan igual o ni siquiera se muestran de la misma manera. Un navegador no hace más que interpretar código HTML, ajustándose lo más posible a unos estándares fijados, para después plasmarlo en la pantalla del usuario de forma gráfica. Por desgracia, algunos

de ellos se desvían más que otros de esos estándares y, como consecuencia directa, una misma página puede tener distintas interpretaciones, de modo que recae sobre el desarrollador el hacer que sus páginas sean compatibles con todos o el mayor número posible de navegadores, mediante pequeños ajustes o "trucos".

¿Qué se debe tener en cuenta a la hora de intentar que nuestra página sea lo más compatible posible? Dado que sería una tarea interminable el ajustarse a todos y cada uno de los navegadores y sus versiones, se opta por lo más razonable: ajustar nuestro código a los navegadores más utilizados.

Hasta hace no mucho, existía una clara predominancia del navegador de Microsoft (Internet Explorer), con una cuota alrededor del 60 por ciento, seguido por otros navegadores que poco a poco han ido ganando adeptos. Esto no quiere decir que Internet Explorer fuera el mejor navegador, de hecho es uno de los más criticados por ser de los que más se intentan alejar de los estándares para imponer el suyo propio. En los dos últimos años la cosa ha cambiado bastante y el uso de Internet Explorer ha llegado a bajar del 40 por ciento. Además hay que sumar la aparición de un nuevo tipo de usuario que está en pleno auge: los que usan Smartphones o tabletas.

Veamos entonces una lista con el resto de navegadores que tienen una presencia importante en la Web hoy en día:

- FireFox: Desarrollado por Mozilla, fue el rival más directo de Internet Explorer y actualmente tiene cerca de un 27 por ciento de cuota de mercado. Por tanto, es una opción a tener en cuenta para hacer nuestras páginas compatibles con él.
- Chrome: Navegador introducido por Google que ha alcanzado rápidamente una cuota del 20 por ciento, convirtiéndose en un serio adversario.
- Safari: Estamos ante el navegador de Apple, eterno adversario de Microsoft. Aunque dispone de una versión para Windows, su uso está más centrado en sistemas de la propia compañía: ordenadores Mac y sus *iAmigos* (iPhone, iPad). Su cuota está alrededor del 7 por ciento.
- Opera: Este navegador, desarrollado por Opera Software, cuenta con una discreta cuota que supera ligeramente el 2 por ciento pese a ser un buen producto y estar presente en varios Smartphones en su versión Opera Mini.

Herramientas para programar en JavaScript

Por suerte para nosotros, JavaScript no requiere que instalemos y aprendamos a usar complejos entornos de desarrollo (como pudiera ser Visual Studio o Eclipse) para escribir código en nuestras páginas.

Quizá se sorprenda si le digo que le será suficiente con saber utilizar el bloc de notas de Windows o cualquier otro editor de texto en su forma más básica. También puede que me llame mentiroso porque un conocido suyo le haya comentado que existen editores específicos para JavaScript. Bien, déjeme defenderme;). Es cierto que hay editores avanzados (como JavaScript Plus o Antechinus), pero la ventaja que ofrecen es la de facilitar la tarea de escribir código sin tener que recordar todos los nombres de funciones o propiedades. Reconozco que esta ayuda es de agradecer cuando se trabaja intensivamente con JavaScript, pero para aprender a manejarlo (como será su caso si ha adquirido este libro) le recomiendo utilizar un editor de texto básico para familiarizarse con la sintaxis.

Para ver el resultado de nuestro trabajo no tendremos más que abrir nuestra página con el navegador que tengamos instalado. Sin más.

Notas finales

Ser un "crack" de JavaScript no es algo que se consiga en quince días puesto que conocer el funcionamiento de muchas de sus partes requiere cierta práctica, y más aún si contamos con todas las técnicas que se basan en él (DHTML, AJAX...).

Con este libro podrá construir una sólida base de conocimientos que le permitirá enfrentarse a cualquier reto que JavaScript pudiera proponerle.

Cómo usar este libro

No, no se asuste. A pesar del título de este apartado, no ha comprado un manual de instrucciones sino una guía para aprender JavaScript. El objetivo entonces de este apartado es adelantarle un poco con lo que se va a encontrar en cuanto pase unas páginas.

El motivo que posiblemente le habrá llevado a adquirir esta guía es porque alguien o algo (un sitio Web normalmente) le ha revelado una palabra, una tecnología, un lenguaje nuevo del que desconocía su existencia o sólo lo conocía de oídas: JavaScript. Esto, por supuesto, le ha llamado la atención y ha decidido lanzarse a aprenderlo para aprovechar su potencial y versatilidad aplicándolo a sus páginas. ¡Bien por ello! No le defraudará en absoluto.

Aunque en esta guía se comience desde cero, cuando la termine tendrá conocimientos más que suficientes para empezar a dar un toque de color a sus páginas, y ya verá como con el tiempo querrá más y más. Y lo mejor de todo es que JavaScript le ofrecerá más y más posibilidades, puesto que su potencial no acaba en la última hoja de este libro ni mucho menos, todavía le quedará mucho a lo que sacarle jugo.

Sin embargo, las cosas no se hacen solas (vale, las puede copiar de Internet... pero eso no es aprender JavaScript) y por ello le animo a intentar aportar su granito de arena mientras lee cada capítulo, escribiendo sus propios ejemplos (por muy tontos que sean) y tratando de jugar con las opciones que tengamos a nuestro alcance. Con esto sí podrá llegar a adquirir bastante soltura en este lenguaje y ser capaz de detectar rápidamente cuál es la línea de código que no funciona bien, o incluso poder adaptar a otras necesidades un código que tuviera hecho para otro propósito.

Conocimientos previos

Aunque aquí se van a explicar todos los conceptos y los elementos utilizados en los ejemplos, podría ser ligeramente favorable tener unos conocimientos sobre HTML pues es la base donde opera JavaScript. De todas formas no se va a realizar un uso intensivo de las decenas de etiquetas HTML sino de algunas muy básicas como los encabezados, enlaces o tablas.

Por otro lado, si ha programado antes en cualquier lenguaje, sin duda leerá este libro con más soltura, sin detenerse en ciertos puntos, pero si no lo ha hecho nunca tampoco debe preocuparle puesto que lo que va a aprender a continuación es justamente un lenguaje de programación, y todos los términos y elementos que se utilizan quedan detallados.

Estructura del libro

Se podría decir que este libro es como una escalera, puesto que no podemos llegar al peldaño más alto sin pasar antes por los demás. Por ello se ha intentado ir paso a paso con los capítulos, sin utilizar elementos antes de ser explicados (cosa que a veces no ha resultado fácil), para así cubrir unos conocimientos mínimos necesarios para explicar el siguiente capítulo. Además, cada capítulo siempre tiene unos ejemplos asociados, que los considero fundamentales para comprender mejor algo con lo que nunca se ha tenido contacto antes.

Ahora le dejo que se sumerja en el amplio mundo de JavaScript y espero que le resulte fácil la lectura de esta guía. ¡Nos vemos en el último capítulo!

Código fuente

Para desarrollar los ejemplos, puede optar por introducir de forma manual el código o utilizar los archivos de código fuente que acompañan al libro.

En el sitio Web de Anaya Multimedia: `http://www.anayamultimedia.es/`, diríjase a la sección **Soporte técnico>Complementos**, donde encontrará una sección dedicada a este libro introduciendo el código **2335597**.

1 Introducción a JavaScript

El lenguaje de programación JavaScript se utiliza en las páginas Web para cubrir las carencias que deja el HTML a secas. Incorporando este lenguaje a nuestras páginas logramos incrementar la funcionalidad de las mismas y la interacción con el usuario, lo que se traduce en unas páginas más dinámicas. Algunos ejemplos de su utilidad podrían ser:

- Validar los datos introducidos en un formulario.
- Detectar el navegador que está utilizando el usuario.
- Almacenar información del usuario para que no tenga que volver a ser introducida en la siguiente visita (*Cookies*).

El código que escribamos en nuestras páginas será interpretado directamente por el navegador que estemos usando, de modo que el servidor Web no interpreta ni ejecuta nuestro código. Esto lo convierte en un lenguaje del lado cliente, al contrario de lo que ocurre con lenguajes como PHP o ASP, donde todo se ejecuta en el lado del servidor.

Si repasamos un poco la historia desde la aparición de Internet, nos encontramos con que se empezaron a necesitar una serie de funcionalidades en las páginas a principios de los 90 que el HTML no era capaz de satisfacer por sí solo. Un caso muy claro fue la validación de formularios en el servidor Web, ya que la única manera posible por entonces era enviar los datos al servidor para que éste devolviera una respuesta al usuario incluyendo los errores encontrados, si los hubiera. Otra circunstancia que agravaba este ir y venir de datos es que en esa época la velocidad de conexión a Internet no era muy alta (del orden de decenas de kbps, frente a los mbps de hoy en día), por lo que surgió la necesidad de comprobar los

campos de los formularios en el propio navegador del usuario, es decir, antes de enviar nada al servidor. Con esto se conseguirían principalmente dos cosas:

1. Evitar largas esperas al usuario con la respuesta del servidor para subsanar un error, por pequeño que fuera.
2. Liberar al servidor de realizar este tipo de operaciones, con lo que la respuesta sería más rápida al tener que realizar menos trabajo.

Brendan Eich, durante su trabajo en la empresa Netscape Communications, desarrolló un lenguaje capaz de dar solución a este problema, apareciendo por primera vez en el navegador Netscape 2.0 con un gran éxito. Inicialmente lo llamaron Mocha, más tarde LiveScript y finalmente fue rebautizado como JavaScript cuando Netscape se alió con Sun Microsystems (creadores del lenguaje Java).

> **Nota**: *Es importante no confundir Java con JavaScript ya que, si bien su nombre y sintaxis son similares, su propósito y complejidad son muy distintas.*

Después del lanzamiento de la primera versión de JavaScript, Microsoft (autores del navegador rival de Netscape, Internet Explorer) desarrolló su propio lenguaje del lado del cliente, al que denominaron JScript, que no era más que una copia de JavaScript adaptada a su propio navegador. Más adelante desarrollarían también VBScript, basado en Visual Basic.

> **Nota**: *JavaScript es soportado por cualquier navegador gráfico actual, mientras que JScript y VBScript sólo pueden ser utilizados junto con Internet Explorer.*

El uso de JavaScript se ha ido extendiendo y evolucionando a medida que la complejidad de las páginas lo requería. De este modo podemos encontrarnos con técnicas de desarrollo como:

- DHTML (*Dynamic HTML*): mediante el uso de HTML, CSS y JavaScript esta técnica es capaz de crear efectos visuales y aumentar la interactividad con el usuario. Por ejemplo, podemos crear un menú desplegable de forma que al pasar el puntero del ratón por encima de cada opción del menú aparezcan a su lado las opciones que tiene asociadas.

- AJAX (*Asynchronous JavaScript And XML*): es una técnica capaz de solicitar datos adicionales al servidor en un segundo plano sin recargar o influir en la página que se está mostrando en el mismo momento. Lo normal es que la página se recargue por completo para poder mostrar la nueva información. Por ejemplo, si un usuario escribe las primeras letras del título de una película en un cuadro de texto, se pueden solicitar las coincidencias con un listado completo y mostrárselas al usuario para que elija.

En el capítulo 13 puede encontrar una descripción un poco más amplia sobre estas técnicas, a modo informativo ya que no son el objetivo de esta guía. Le recomiendo que, una vez haya adquirido los conocimientos sobre JavaScript mediante los capítulos siguientes, consulte otros libros de esta editorial, como "CSS, DHTML y AJAX", el cual tuve ocasión de leer y me sirvió de gran ayuda para ampliar mis conocimientos y añadir funcionalidades a mis páginas que antes me parecían muy complejas.

1.1. Versiones de JavaScript y los navegadores

Con el paso del tiempo se han ido implementando nuevas y ampliadas versiones de JavaScript para soportar nuevas funcionalidades, lo cual obliga a actualizar los navegadores, dado que es donde se interpreta el código, y así poder mostrar correctamente una página que utilice estas nuevas versiones.

Por todo esto, el primer problema al que se enfrentaba un programador de código JavaScript es que los usuarios no siempre tenían la última versión de los navegadores y, por tanto, algunas instrucciones podrían no ser reconocidas. Actualmente es raro encontrar un navegador que no soporte al menos la versión 1.5 de JavaScript (véase la tabla 1.1) aunque seguimos teniendo otros problemas a la hora de escribir nuestro código por la existencia de distintos DOM (*Document Object Model*). Al cargar una página, los navegadores crean una jerarquía de objetos, cada uno de los cuales representa un elemento de la misma. Esta jerarquía es la que permite a los desarrolladores acceder y manipular todos los elementos que contiene dicha página (por ejemplo, guardar el teléfono introducido en un formulario). Aunque existe un estándar definido para

crear el DOM, en algunos navegadores o incluso en distintas versiones del mismo navegador, los objetos no siempre se comportan igual.

Siempre que programemos un código JavaScript es conveniente probarlo en distintos navegadores para asegurar su correcto funcionamiento. En otro caso, estaremos dejando que un grupo de visitantes de nuestra página no pueda visualizarla correctamente o incluso que no pueda interactuar con ella.

Tabla 1.1. Versiones de JavaScript y navegadores que las soportan.

Versión JavaScript	*Navegador*
1.0	Netscape (2.0)
1.1	Netscape (3.0)
1.2	Netscape (4.0-4.05)
1.3	Netscape (4.06-4.7x), Internet Explorer (4.0)
1.4	Netscape Server
1.5	Netscape (6.0), Internet Explorer (5.5-8.0), FireFox (1.0), Opera (6.0-9.0), Chrome (1.0), Safari (3.0.5)
1.6	FireFox (1.5)
1.7	FireFox (2.0), Safari (3.x)
1.8	FireFox (3.0), Opera (11.5)
1.8.1	FireFox (3.5)
1.8.2	FireFox (3.6)
1.8.5	FireFox (4.0), Internet Explorer (9.0)

Nota: *La tabla 1.1 refleja los navegadores que tienen un soporte total o muy cercano sobre la versión de JavaScript correspondiente. Aún así, existen versiones más recientes de navegadores que soportan parte de las funcionalidades de versiones superiores de JavaScript.*

Cada versión de JavaScript es compatible hacia atrás con las antiguas, es decir, un código escrito para una versión concreta siempre será ejecutado correctamente en las siguientes

versiones. Por tanto, no es recomendable utilizar características de la última versión si queremos que nuestro código pueda ser ejecutado por el mayor número posible de navegadores.

En su versión 1.3, Netscape Communications presentó su lenguaje a ECMA (*European Computer Manufacturers Association*), organización que se encarga de crear estándares con el fin de facilitar el uso de las Tecnologías de Información. Como resultado de esta estandarización, surgió la especificación ECMAScript, a la cual se ajustó JavaScript en las versiones posteriores.

Tabla 1.2. Versiones ECMAScript adaptadas en JavaScript.

Versión ECMAScript	*Versión JavaScript*
ECMA-262 (2ª edición)	1.3
ECMA-262 (3ª edición)	1.5
ECMA-262 (5ª edición)	1.8.5

1.2. Integración con HTML

Para poder insertar nuestro código JavaScript dentro de una página HTML, disponemos de la etiqueta `<SCRIPT>`. Su contenido será interpretado y/o ejecutado según se vaya cargando la página.

Esta etiqueta debe usarse junto con el atributo `TYPE`, el cual es obligatorio. Anteriormente se utilizaba el atributo `LANGUAGE`, pero hace tiempo que está en desuso. Para el caso de JavaScript, debemos etiquetar nuestro código de la siguiente manera:

```
<SCRIPT TYPE="text/javascript">
   Nuestro código
</SCRIPT>
```

Si el código utiliza alguna característica especial de una versión en concreto, podremos especificarla de la siguiente manera: `TYPE="text/javascript;version=1.6"`. En caso de que el navegador no la soporte, ignorará el código que contenga esa etiqueta.

Cuando tenemos un código muy extenso y queremos reutilizarlo en otras páginas, podríamos pensar que tendríamos que escribirlo repetidas veces haciendo que su mantenimiento sea difícil pero, afortunadamente para nosotros, existe la posibi-

lidad de meterlo en un fichero con extensión ".js" y utilizar además el atributo SRC de la etiqueta <SCRIPT> para incluirlo en todas las páginas que lo necesiten, suponiendo un esfuerzo bastante menor:

```
<SCRIPT TYPE="text/javascript" SRC="mi_codigo.js">
</SCRIPT>
```

En este caso, si escribimos código adicional dentro de la etiqueta <SCRIPT>, éste será ignorado por el navegador dado que estamos usando el atributo SRC.

```
<SCRIPT TYPE="text/javascript" SRC="mi_codigo.js">
Lo que pongamos aquí no se ejecutará
</SCRIPT>
```

Hay dos sitios dentro de una página donde podemos situar nuestro código, el número de veces que queramos:

1. Dentro de la etiqueta <HEAD>. Esta posición se suele utilizar normalmente para cargar ficheros externos que vamos a utilizar en otras partes de la misma página.
2. Dentro de la etiqueta <BODY>. Aquí se coloca habitualmente el código que va a realizar acciones sobre la página, pudiendo hacer referencia a parte del código cargado dentro de <HEAD>.

```
<HEAD>
<SCRIPT TYPE="text/javascript" SRC="mi_codigo.js">
</SCRIPT>
</HEAD>
<BODY>
   <SCRIPT TYPE="text/javascript">
      Aquí hago algunas operaciones
   </SCRIPT>
</BODY>
```

Ya iremos viendo que existen algunas restricciones cuando ponemos nuestro código en el <BODY>.

1.3. Sintaxis del lenguaje

La sintaxis de JavaScript es muy parecida a la de otros lenguajes como Java y C, por lo que si ya conoce esos lenguajes se sentirá cómodo escribiendo código. De cualquier modo, lo explicaremos a continuación para que ningún lector tenga problemas a la hora de adentrarse en el mundo JavaScript.

1.3.1. Mayúsculas y minúsculas

JavaScript es capaz de distinguir mayúsculas y minúsculas en los nombres de variables, funciones u otros objetos. Por tanto, no será lo mismo la variable posicion que la variable Posicion o incluso que una tercera llamada POSICION, pudiendo estar definidas todas a la vez y con valores totalmente distintos.

Debemos tener cierto cuidado mientras escribimos nuestro código, ya que estos pequeños matices pueden dar algún quebradero de cabeza, sobre todo las primeras veces que programemos en JavaScript.

Por norma general se usarán las minúsculas para evitar este tipo de conflictos, aunque ya veremos que se usan algunas reglas (recomendadas pero no obligatorias) para mejorar la lectura y comprensión del código.

1.3.2. Comentarios

Los comentarios en el código sirven para ayudar al programador u otra persona a comprender mejor su funcionamiento, o simplemente para dejar anotaciones que nos puedan ser de interés (autor del código, fecha en la que se escribió, etc.).

Todo texto que se escriba como comentario no será interpretado por el navegador, por lo que su contenido es irrelevante para él. Existen dos tipos de comentarios:

1. Línea simple: se identifican escribiendo dos barras (//) al comienzo de la línea, y van seguidas del texto que queramos etiquetar como comentario.
2. Multilínea: si el comentario es muy extenso o queremos presentarlo en varias líneas, entonces debemos escribir una barra junto a un asterisco (/*) al principio del texto y un asterisco junto a una barra (*/) al final del mismo.

```
<SCRIPT TYPE="text/javascript">
   // Esto es un comentario corto de una línea.
   /* En cambio este otro ocupa más espacio y,
   además,no pasa nada si escribo algo de
   código como alto = 5; ya que estamos en
   un comentario. */
   ancho = 7; // Sí se ejecuta el código, pero
              este texto no.
   /* Esto es sólo una línea, pero también
   es válido. */
</SCRIPT>
```

1.3.3. Separación de instrucciones

Para separar las distintas instrucciones que contiene nuestro código tenemos dos opciones:

1. Escribir un punto y coma (;) al final de cada una de las instrucciones.
2. Insertar un salto de línea al terminar cada una de las instrucciones.

De este modo, los siguientes ejemplos funcionarán correctamente en JavaScript:

```
// Código utilizando punto y coma
 <SCRIPT TYPE="text/javascript">
   numero = 1;
   cuadrado = numero * numero;
 </SCRIPT>
 // Código utilizando salto de línea
 <SCRIPT TYPE="text/javascript">
   numero = 1
   cuadrado = numero * numero
</SCRIPT>
```

No siempre es necesario que cada instrucción comience en una línea nueva, pero si ponemos varias instrucciones seguidas en la misma línea entonces estamos obligados a utilizar el punto y coma para separarlas, ya que de otro modo se interpreta todo como una única instrucción, con el consecuente error de sintaxis.

```
// Varias instrucciones utilizando punto y coma ->
Funciona
<SCRIPT TYPE="text/javascript">
   alto = 5; ancho = 7; area = alto * ancho;
</SCRIPT>
// Varias instrucciones sin punto y coma ->
Genera un error
<SCRIPT TYPE="text/javascript">
   alto = 5 ancho = 7 area = alto * ancho
</SCRIPT>
```

> **Nota**: *Es aconsejable acostumbrarse a escribir cada instrucción en una nueva línea y utilizar un punto y coma al final para que la apariencia de nuestro código sea clara y se asemeje a la habitual en otros lenguajes, como Java o C. Así mismo, un correcto tabulado del código ayudará a mejorar su compresión.*

1.4. Navegadores sin soporte JavaScript

Aunque no es algo común, puede darse la posibilidad de que el visitante de nuestra página esté usando un navegador que no soporte JavaScript, o simplemente lo tenga desactivado. En estos casos, todo el código que tengamos escrito no será interpretado ni ejecutado, por lo que el visitante no será capaz de ver correctamente nuestra página o hacer uso de ella.

La causa de esto es que el navegador no reconoce la etiqueta <SCRIPT>, por lo que todo el código que hayamos escrito dentro de ella puede ser ignorado o llegar a ser mostrado como un texto HTML. Para evitar esto último, es bueno marcar el código como un comentario HTML, usando <!-- al principio y --> al final, con la particularidad de escribir el cierre de comentario a su vez como un comentario JavaScript para evitar que el navegador lo interprete como parte de nuestro código. Lo explico a continuación:

```
<SCRIPT TYPE="text/javascript">
<!--
   numero = 4;
   cuadrado = numero * numero;
// -->
</SCRIPT>
```

Adicionalmente, tenemos un `as en la manga´ para advertir al usuario de que su navegador no soporta JavaScript: la etiqueta <NOSCRIPT>. El contenido HTML de esta etiqueta sólo será mostrado si el navegador no puede interpretar JavaScript. Veamos un ejemplo:

```
<HTML>
<BODY>
<SCRIPT TYPE="text/javascript">
<!--
   numero = 4;
   cuadrado = numero * numero;
// -->
</SCRIPT>
   ¡Bienvenido a mi página!
   <NOSCRIPT>Lo siento, pero tu navegador no soporta
   JavaScript :(</NOSCRIPT>
</BODY>
</HTML>
```

Un navegador que soporte JavaScript ejecutará nuestro código y mostrará únicamente el mensaje de bienvenida, mientras que los navegadores que no lo hagan ignorarán el código

y mostrarán tanto el mensaje de bienvenida como el aviso. En lugar de mostrar sólo este aviso, también podemos dar la opción al usuario de dirigirse a una página que hayamos creado con aspecto similar, pero sin usar JavaScript.

```
<NOSCRIPT>
   Lo siento, pero tu navegador no soporta JavaScript
   :(<BR>
   Sin embargo, puedes ver una página similar
   <A HREF="pagina_sin_javascript.html">aquí</A>
</NOSCRIPT>
```

Advertencia: *Si el navegador no soporta la versión de JavaScript que hayamos especificado (por ejemplo la 1.6), tampoco se mostrará el texto contenido en <NOSCRIPT> ya que sí es capaz de reconocer la etiqueta <SCRIPT>.*

2

Variables, tipos de datos y expresiones regulares

2.1. Variables

Una variable es un espacio de memoria donde almacenamos temporalmente un dato que utilizamos en las operaciones dentro de nuestro código como, por ejemplo, el resultado de una suma. Ese dato podrá ser recuperado en cualquier momento e incluso ser modificado.

En el código, cada variable estará identificada por un nombre a nuestra elección y que podrá estar formado por caracteres alfanuméricos, el carácter guión bajo o subrayado (_) y el símbolo de dólar ($). La única restricción es que no puede comenzar por un número. También es importante destacar que se distingue entre mayúsculas y minúsculas, por lo que la variable `suma` no será la misma que la variable `SUMA`.

Tabla 2.1. Ejemplos de nombres de variables.

Nombre válido	*Nombre no válido*
dos_numeros	2_numeros
nombre1	nombre-1
Documento	%Documento
nombreApellidos	nombre&apellidos
$dinero	€dinero

También debe tener en cuenta que el nombre de una variable no puede coincidir con el de una palabra reservada de JavaScript (ver apéndice A), ni con el de otro elemento que hayamos definido (funciones u objetos).

Aunque no hay obligación para ello, es una buena práctica usar minúsculas en los nombres de variables excepto si se compone de varias palabras, en cuyo caso escribiremos en mayúsculas la primera letra de cada palabra siguiente a la primera. Por ejemplo: `nombre`, `miNombre`, `miNombreCompleto`. Con el uso de esta sencilla regla estaremos haciendo nuestro código más limpio y fácil de seguir por otro programador o nosotros mismos a la hora de retomarlo.

2.1.1. Declaración de variables

Para utilizar una variable en nuestro código debe estar indicado en alguna parte de éste cuál es su nombre y, si procede, el valor que le queremos dar. A esto se le conoce como, declaración de una variable.

En la mayoría de lenguajes de programación es obligatorio declarar cada variable antes de utilizarla en el código pero en JavaScript se puede hacer sobre la marcha según las vayamos necesitando. No obstante, como queremos hacer un código ejemplar, es bueno que nos acostumbremos a declarar las variables antes de darles uso. Para ello existen dos maneras de hacerlo:

1. Declaración explícita: Consiste en usar la sentencia `var` seguida del nombre de la variable. Esta opción es la recomendada para añadir claridad al código.
2. Declaración implícita: Es cuando escribimos directamente el nombre de la variable sin una sentencia previa. En este caso debe ir acompañada obligatoriamente de un valor si no queremos obtener un error de declaración.

```
<SCRIPT TYPE="text/javascript">
   // Declaración explícita
   var miEdad;
   // Declaración implícita (errónea al no tener
   // un valor)
   miEdad;
   // Declaración implícita (correcta)
   miEdad = 26;
</SCRIPT>
```

También es posible declarar explícitamente varias variables a la vez, separando sus nombres por comas:

```
<SCRIPT TYPE="text/javascript">
   // Declaración explícita múltiple
   var nombre, apellidos, edad;
</SCRIPT>
```


Opcionalmente, podemos asignar un valor a la variable al mismo tiempo que se declara, lo cual se conoce como inicialización, a través del operador de asignación = (explicado más adelante). Esta inicialización es ineludible si hacemos declaraciones implícitas, como hemos visto hace un momento.

```
<SCRIPT TYPE="text/javascript">
   // Declaración explícita con inicialización
   var alto = 5;
   // Declaración implícita con inicialización
   ancho = 12;
   // Declaración explícita múltiple con inicialización
   var alto = 5, ancho = 12, profundo = 7;
   // Declaración implícita múltiple con inicialización
   hora = 14, minutos = 35, segundos = 41;
   // Declaración explícita múltiple con y sin
   // inicialización
   var calle, numero = 7, piso = 3;
</SCRIPT>
```

Si declaramos la misma variable más de una vez y la inicializamos con valores, prevalecerá el último que hayamos asignado.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de la misma variable dos veces (alto
   // valdrá 8)
   var alto = 5;
   var alto = 8;
</SCRIPT>
```

Recordemos la importancia de las mayúsculas y minúsculas puesto que estaremos definiendo variables distintas.

```
<SCRIPT TYPE="text/javascript">
   // Declaración con mayúsculas y minúsculas
   var alto = 5;
   var Alto = 8;
</SCRIPT>
```

2.1.2. ¿Y las constantes?

Una constante es un dato cuyo valor no puede ser modificado después de haber sido inicializado. Esto es útil cuando necesitamos trabajar con valores fijos como, por ejemplo, el número Pi (3,14).

Aunque es posible declarar constantes en JavaScript a través de la sentencia `const`, ésta sólo es soportada desde la versión 1.5 de JavaScript y principalmente por los navegadores basados

en Mozilla (como FireFox), aunque se prevé que tenga total integración con la versión 2.0 de JavaScript. Los navegadores que no la soporten ignorarán todas las declaraciones que hagamos utilizando `const` haciendo que aparezcan errores cuando se intente utilizar una de esas constantes en nuestro *script*.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de constantes
   const pi = 3.14;
   const valorMaximo = 256;
</SCRIPT>
```

Si está acostumbrado a programar en otros lenguajes es posible que lo siguiente le resulte extraño pero, para asegurar el funcionamiento de nuestro código en los distintos navegadores, definiremos nuestras constantes como una variable (¡a grandes males, grandes remedios!).

Para poder diferenciarlas claramente de las variables normales, podemos escribir su nombre completamente en mayúsculas y separaremos las palabras que la formen con guiones bajos o subrayados (_).

```
<SCRIPT TYPE="text/javascript">
   // Declaración de constantes como variables
   var PI = 3.14;
   var VALOR_MAXIMO = 256;
</SCRIPT>
```

> **Advertencia**: *Las constantes definidas con la sentencia `var` no dejan de ser variables, por lo que su valor sigue siendo modificable a través del código. Tenga cuidado de no alterar el valor de una constante.*

2.2. Tipos de datos

En nuestro código vamos a utilizar información de distinta clase, como textos o números. Estas clases son las que se conocen como tipos de valores o tipos de datos, de forma que cada variable o constante que definamos pertenecerá a uno de ellos.

Una de las muchas particularidades de JavaScript, en su afán por hacer más cómoda la programación con este lenguaje, consiste en no tener que definir el tipo de una variable al declararla, será el propio JavaScript el que determine a cuál corresponde en función del valor asignado. Gracias a esto, una variable puede cambiar su tipo de datos a lo largo de un *script* sin que esto provoque errores, aunque es poco común hacer este tipo de acciones.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var edad;
   // La variable es un número…
   edad = 30;
   // y ahora un texto
   edad = "treinta";
</SCRIPT>
```

> **Nota**: *En algunos de los ejemplos que veremos a lo largo del libro se utiliza la instrucción `alert`, cuyo funcionamiento aún no hemos explicado, pero de momento le adelanto que nos permite mostrar el valor de una variable en la pantalla. De esta forma podrá escribir los ejemplos en su ordenador y comprobar los resultados: `alert(edad);`*

A continuación veremos todos los tipos de datos que maneja JavaScript.

2.2.1. Números

Como viene siendo habitual, JavaScript se diferencia de nuevo de otros lenguajes de programación ya que sólo tiene un tipo de valor numérico. Por tanto, todo número que escribamos, ya sea entero o real, será considerado del mismo tipo. También lo serán los números reales escritos en notación científica mediante el indicador de exponente `E` o `e`. Veamos algunos ejemplos:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de datos de tipo numérico
   var altura = 5;
   var precio = 12.65;
   var temperatura = -3.0;
   var PI = 3.141592;
   var seiscientos = 6E2;
   var sieteMilimetros = 7e-3;
   var MILLON_Y_MEDIO = 1.5e6;
</SCRIPT>
```

> **Nota**: *El separador decimal para los números reales es el punto (.) y no la coma (,).*

Además, los números enteros pueden ser representados en otros sistemas distintos al decimal (base 10):

- Sistema octal (base 8): Utiliza dígitos del 0 al 7 para representar un número. Para ello debemos escribir un cero (0) delante del número. Este sistema está obsoleto, aunque JavaScript 1.5 mantiene un soporte para él con tal de mantener la compatibilidad hacia atrás.
- Sistema hexadecimal (base 16): Consta de 16 dígitos: los comprendidos entre el 0 y el 9 y las letras entre la A y la F (minúsculas o mayúsculas). Para representarlo tenemos que escribir un cero y una equis (0x) antes que el número.

Tabla 2.2. Ejemplos de números en distintas bases.

Base	*Número*	*Equivalencia en base 10*
10	23	23
8	027	23
16	0x17	23
10	127	127
8	0177	127
16	0x7F	127
16	0x7f	127

2.2.2. Lógicos

Este tipo, conocido también como *boolean* o booleano, tan sólo abarca dos valores posibles: verdadero (`true`) y falso (`false`), que nos servirán principalmente para tomar decisiones sobre realizar una serie de acciones o no en nuestro código. Un ejemplo con una situación cotidiana sería: Si llueve entonces me llevo el paraguas, en otro caso lo dejo en casa. Nuestra variable booleana sería `llueve`, y, en función de su valor (`true` o `false`), realizaría una acción u otra (llevarme el paraguas o dejarlo en casa).

```
<SCRIPT TYPE="text/javascript">
   // Declaración de datos de tipo lógico o booleano
   var esVerdadero = true;
   var esFalso = false;
   var COMPROBAR = true;
</SCRIPT>
```


Advertencia: *Tenga de nuevo en cuenta la importancia del uso de mayúsculas y minúsculas, ya que JavaScript no reconocerá valores del estilo True o TRUE como un booleano.*

2.2.3. Cadenas

Otra clase de información muy común en los *scripts* es la cadena de caracteres o *string*. Este tipo consta de un texto con cualquier carácter (letras, números y símbolos).

Las cadenas o *strings* se escriben en el código entre comillas dobles (") o simples (') pero nunca combinadas. Lo más habitual es utilizar las comillas dobles para representar un texto.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de datos de tipo cadena o string
   var idioma = "español";
   var dni = '12345678A';
   var direccion = "C/ Mayor, 5";
   var FICHERO = "mi_codigo.js";
   // Declaración errónea de datos de tipo cadena o string
   var pais = "España';
 </SCRIPT>
```

Caracteres especiales

A veces es necesario incluir en el texto de una variable ciertos caracteres, como una comilla simple ('), el tabulador o el salto de línea, que deben escribirse de una forma concreta para que JavaScript pueda reconocerlos.

Para identificar esos caracteres especiales utilizaremos lo que se denomina carácter de escape, que no es más que una barra inversa o *backslash* (\). Este carácter de escape irá colocado delante del carácter que se quiera incluir expresado mediante un código (véase la tabla 2.3).

Tabla 2.3. Lista de caracteres especiales.

Código del carácter	*Modo de empleo*	*Descripción*
"	\"	Comilla doble
'	\'	Comilla simple
\	\\	Barra inversa o backslash
n	\n	Nueva línea

Código del carácter	Modo de empleo	Descripción
t	\t	Tabulador
r	\r	Retorno de carro
b	\b	Retroceso o backspace
f	\f	Avance de página
v	\v	Tabulador vertical

Existe también la posibilidad de incluir cualquier carácter UNICODE (`http://www.unicode.org`) mediante la sentencia de escape `\uxxxx`, donde `xxxx` representan los dígitos hexadecimales de un carácter. Por ejemplo, los dígitos `00A9` corresponden al símbolo de copyright (©), el `0009` al tabulador y el `0022` a la comilla doble. La realidad es que rara vez se usan, siendo los más comunes los contenidos en la tabla 2.3.

Si utilizamos el carácter de escape con otro carácter fuera de los mencionados el resultado será dicho carácter, es decir, se representa a sí mismo. De esta forma, si escribiésemos `\c` en la cadena, aparecería el carácter "c".

Veamos algunos sencillos ejemplos junto con el resultado visto por pantalla:

```
<SCRIPT TYPE="text/javascript">
   // Ejemplo 1 de caracteres especiales en cadenas
   var cadena = "El fichero está en C:\\Documentos";
   // Mostramos su valor
   alert(cadena);
</SCRIPT>
```

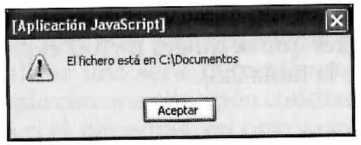


Figura 2.1. Ejemplo 1 de caracteres especiales en cadenas.

```
<SCRIPT TYPE="text/javascript">
   // Ejemplo 2 de caracteres especiales en cadenas
   var cadena = "Este libro se titula \"Guía de
   JavaScript\"";
   // Mostramos su valor
   alert(cadena);
</SCRIPT>
```

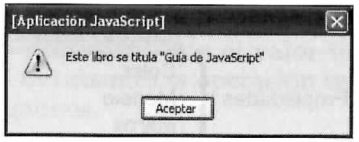



Figura 2.2. Ejemplo 2 de caracteres especiales en cadenas.

```
<SCRIPT TYPE="text/javascript">
   // Ejemplo 3 de caracteres especiales en cadenas
   var cadena = "Primera línea\nSegunda línea\nTer\
   u00A9era línea";
   // Mostramos su valor
   alert(cadena);
</SCRIPT>
```

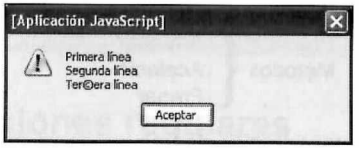


Figura 2.3. Ejemplo 3 de caracteres especiales en cadenas.

2.2.4. Objetos

Aunque este libro contiene un capítulo dedicado en exclusiva al manejo de objetos en JavaScript, aquí explicaré lo que es un objeto, puesto que se trata de un tipo más.

A grandes rasgos, un objeto es una estructura que engloba una serie de datos (propiedades) y las funcionalidades necesarias para manejarlo (métodos).

Para que tenga una pequeña idea de cómo imaginarse un objeto, veámoslo con un ejemplo: pensemos en un coche (¡ese con el que tanto sueña!), el cual tiene un conjunto de características como el color, el modelo y la marca y una serie de funcionalidades como arrancar, acelerar y frenar. En este contexto, el objeto sería el coche, las propiedades sus características y los métodos sus funcionalidades asociadas.

Ahora bien, como no todos los coches son iguales necesitamos poder especificar las características individuales de cada uno. A cada uno de ellos se le denomina instancia. Para el caso que nos ocupa, podríamos definir una instancia del objeto "coche" asignando valores concretos a sus propiedades: color "Rojo", modelo "Punto" y marca "Fiat". Los métodos que tendrá esta instancia serán los mismos que los del objeto: arrancar, acelerar y frenar.

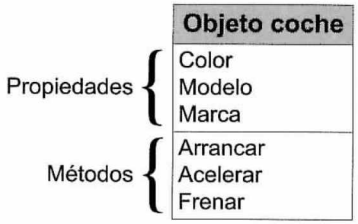


Figura 2.4. Representación del objeto "coche".

Figura 2.5. Representación de una instancia del objeto "coche".

Para que le vaya resultando familiar, le confesaré que cuando asignamos un valor de cualquier tipo (números, booleanos y cadenas) a una variable, ésta se convierte en realidad en una instancia de un objeto con sus propiedades y métodos propios. No se preocupe si esto le choca un poco ahora, puesto que lo explicaré en capítulos posteriores con más detalle y veremos algunos ejemplos que, espero, no le dejen con los ojos abiertos de sorpresa.

2.2.5. Valores especiales

Las variables en JavaScript pueden tomar una serie de valores especiales cuando estamos trabajando con ellas:

- Indefinido (`undefined`): Este valor indica que la variable aún no ha tomado ningún valor.
- Nulo (`null`): Nos permite definir o comprobar que una variable está vacía.
- No numérico (`NaN`, *Not a Number*): Una variable recibe este valor si almacena el resultado de alguna operación matemática no válida (división por cero, por ejemplo).
- Infinito (`Infinity`): Nos indica que una variable numérica ha tomado un valor demasiado alto (infinito)

como resultado de la operación que hemos tratado de hacer. También existe el valor infinito negativo (`-Infinity`) cuando la operación se realiza con números negativos.

Tabla 2.4. Ejemplos con valores especiales.

Operación	*Valor*
var numero;	undefined
var ahorros = null;	null
var division = 0/0;	NaN
var fraccion = 5/0;	Infinity
var fraccion = -1/0;	-Infinity
var suma = numero + 5;	NaN (numero no estaba inicializado)

2.3. Expresiones regulares

Una expresión regular, o patrón, nos permitirá buscar coincidencias dentro de un texto (por ejemplo, todas las palabras que tengan la letra T) o comprobar que una cadena tiene un formato concreto (el DNI de una persona).

Esto nos será especialmente útil cuando queramos realizar de golpe una misma operación sobre un grupo determinado de palabras como, por ejemplo, cambiar a mayúsculas todas las palabras que contengan la cadena "JavaScript". Con los métodos tradicionales tendríamos que buscar en el texto palabra por palabra hasta encontrar la que nos interesa y después cambiarla a mayúsculas.

En un capítulo posterior veremos cómo trabajar con estos patrones dentro de nuestro código a través del objeto `RegExp`.

2.3.1. Escribir una expresión regular

En JavaScript las expresiones regulares se identifican colocando unas barras (`/`) al principio y al final de la cadena que la constituyen.

Tenemos dos opciones a la hora de definir una expresión regular para encontrar las palabras que nos interesan: usar un patrón simple o uno con caracteres especiales.

Patrón simple

Consiste en escribir un conjunto de caracteres para después encontrar coincidencias directas con él, es decir, nuestro conjunto de caracteres deberá aparecer dentro de la palabra en el mismo orden en que están escritos.

Supongamos que tenemos el texto "el pájaro saltó del nido" y el patrón `/el/`. Si aplicásemos el patrón, obtendríamos que las palabras "el" y "del" lo cumplen mientras que el resto no entrarían en el saco.

Patrón con caracteres especiales

Para hacer más útiles las expresiones regulares, podemos utilizar una serie de estructuras y caracteres especiales que nos permitirán crear patrones más complejos y adaptables a nuestras necesidades. Para que le sea más fácil comprender el funcionamiento de cada uno, los vamos a agrupar en base a su funcionalidad.

Caracteres de repetición

Gracias a estos caracteres podremos indicarle a nuestro patrón que se busquen coincidencias teniendo en cuenta que algunas partes de la cadena pueden repetirse un número determinado de veces. Pasemos a ver las distintas opciones que tenemos para ello:

- Asterisco (`*`): Nos permite indicar que el carácter que le precede puede aparecer cero o más veces.
- Más (`+`): En este caso indica que el carácter que le precede puede repetirse una o más veces.
- Interrogación (`?`): Igual que los anteriores, pero indicando que el carácter puede aparecer ninguna o una vez.
- `{n}`: Siendo `n` un entero positivo, esta estructura nos permite indicar que el carácter que le precede se debe repetir exactamente el número de veces especificado por `n`.
- `{n,}`: Similar al anterior, pero esta vez se especifica que el carácter debe aparecer al menos `n` veces.
- `{n, m}`: Esto supone una ampliación de los anteriores con la que, teniendo `n` y `m` como enteros positivos, podemos indicar que el carácter que preceda a esta estructura debe repetirse un número de veces entre las definidas por `n` y `m`.

Tabla 2.5. Ejemplos de patrones con caracteres de repetición.

Patrón	*Significado que lo cumplirían*	*Valores*
/hola*/	"hol" seguido de cero o más aes.	hol, hola, holaaaaa
/hola+/	"hol" seguido de una o más aes.	hola, holaaaaa
/hola?/	"hol" seguido de cero o una "a".	hol, hola
/hola{2}	/"hol" seguido de exactamente dos aes.	holaa
/hola{2,}/	"hol" seguido de dos o más aes.	holaa, holaaaaa
/hola{2,4}/	"hol" seguido de dos a cuatro aes.	holaa, holaaa, holaaaa
/ho+la*/	"h" seguido de una o más oes, seguido de "l" y seguido de cero o más aes.	hol, hooool, hola, hoolaaa

Advertencia: *Hay que remarcar que en los ejemplos de la tabla 2.5 las palabras que contengan más letras a continuación del patrón encontrado serán igualmente válidas, es decir, en estos ejemplos no se obliga a que la palabra termine estrictamente con el último carácter del patrón, sino que puede haber más letras después. Pongamos como ejemplo el patrón `/hola+/`. Si nuestra palabra fuese "holaaas!" nuestro patrón encontraría coincidencia con ella puesto que lo contiene ("holaaa") y de este modo la parte restante no afectaría ("s!").*

Caracteres especiales

Funciona de forma exacta a lo visto en el apartado anterior, donde explicamos los caracteres especiales de las cadenas. En el caso de las expresiones regulares, los caracteres que se utilizan tienen bastantes coincidencias con los de las cadenas aunque hay otros más complejos:

- Punto (`.`): Si lo escribimos dentro de un patrón, coincidirá con cualquier carácter simple excepto con el de salto de línea.

- `\n`: Coincide con el carácter de nueva línea (o salto de línea).
- `\r`: El retorno de carro.
- `\t`: Carácter de tabulado.
- `\v`: El de tabulado vertical.
- `\f`: Avance de página.
- `\uxxxx`: Coincide con el carácter UNICODE que tiene como código cuatro dígitos hexadecimales representados por `xxxx`.
- `\b`: Coincide con un separador de palabra, como puede ser el espacio o el carácter de nueva línea.
- `\B`: Coincide con un carácter que no sea separador de palabra.
- `\cX`: Coincide con un carácter de control en una cadena siendo `X` dicho carácter, es decir, equivale a la combinación de teclas **Control-X**.
- `\d`: Coincide con un dígito entre cero y nueve.
- `\D`: Coincide con un carácter que no sea un dígito.
- `\s`: Coincide con un único carácter de separación (espacio, tabulado, avance de página o nueva línea).
- `\S`: Coincide con un único carácter que no sea de separación.
- `\w`: Coincide con cualquier carácter alfanumérico (letras de la "a" a la "z", minúsculas o mayúsculas y números del cero al nueve) o el subrayado.
- `\W`: Coincide con cualquier carácter no alfanumérico ni subrayado.
- `\0` (cero): Coincide con el carácter nulo o de fin de cadena. No puede tener ningún dígito a continuación (`\03` por ejemplo).

En los ejemplos de la tabla 2.6 se han escrito los caracteres no representables como texto entre corchetes ([]).

Tabla 2.6. Ejemplos de patrones con caracteres especiales.

Patrón	*Significado*	*Valores que lo cumplirían*
/.n/	Un carácter seguido de una "n".	**un**, **en**
/hola\n/	"hola" seguido de un salto de línea.	**hola[NUEVA LÍNEA]**

Patrón	*Significado*	*Valores que lo cumplirían*
/java\tscript/	"java" seguido de un tabulado y de "script".	**java script**
/java\bscript/	"java" seguido de un separador y de "script".	**java[NUEVA LÍNEA] script**, **java script**
/java\Bscript/	"java" no seguido de un separador y de "script".	**java-script**, **java0script**, **java*script**
/java\dscript/	"java" seguido de un dígito y de "script".	**java0script**, **java6script**
/java\sscript/	"java" seguido de un carácter de separación y de "script".	**java[NUEVA LÍNEA] script**, **java script**, **java script**
/java\wscript/	"java" seguido de un carácter alfanumérico o subrayado y de "script".	**java5script**, **javaX script**, **javaxscript**, **java_script**
/javascript\0/	"javascript" seguido de un carácter nulo.	JavaScript**[NULO]**
/\d\d-\d\d-\d\d\d\d/	una fecha: dos dígitos, guión, dos dígitos, guión y cuatro dígitos.	18-01-2009, 04-12-1997
/\w\w\w\w/	cuatro caracteres alfanuméricos o subrayados (una contraseña por ejemplo).	java, JaVa, J4va, 3_js

Agrupación de valores

En ocasiones puede surgirnos la necesidad de crear patrones que puedan encontrar una coincidencia con palabras que son muy similares y sólo se diferencian en una pequeña parte, lo cual nos obligaría a crear un patrón por cada una de esas diferencias. Por eso nos viene `como anillo al dedo´ el uso de estas estructuras para así fusionar todo bajo un mismo patrón.

- [xxx]: Coincide con uno de los caracteres que están entre los corchetes. También es posible especificar un rango de caracteres usando un guión. Por ejemplo [abc] es lo mismo que [a-c], o [01234] sería equivalente a [0-4]. La única regla para usar este guión es que esos valores deben ser contiguos, es decir, no podremos decir que el rango es del cero al cuatro, excepto el tres.
- [^xxx]: El uso del circunflejo (^) dentro de un conjunto de caracteres hace que coincida con cualquier carácter salvo los que estén indicados en el conjunto. A esto se le conoce como conjunto complementado o negado. Aquí también es posible definir un rango con el guión.
- Barra vertical (x | y): Puede coincidir con x o y, pero no con los dos. Puede ir dentro de corchetes para acortar mejor los valores.
- [\b]: Coincide con el carácter de retroceso o *backspace*. No lo confunda con el carácter especial \b (separador de palabra).

Tabla 2.7. Ejemplos de patrones con agrupación de valores.

Patrón	*Significado*	*Valores que lo cumplirían*
/[ue]n/	"u" o "e" seguido de "n".	**un**, **en**
/[^ue]n/	Cualquier carácter excepto "u" o "e" seguido de "n".	a**n**, F**n**, 1**n**...
/cara|cruz/	"cara" o "cruz".	**cara**, **cruz**
/gat[a|o]/	"gat" seguido de "a" u "o".	gat**a**, gat**o**

Caracteres de posición

Estos caracteres nos permiten especificar en qué parte de la línea debe existir la coincidencia dentro de nuestro patrón.

- Circunflejo (^): Hace que nuestro patrón tenga que coincidir desde el inicio de la línea. Debe ir colocado siempre al inicio del patrón.
- $: Cumple la misma función que el anterior pero haciendo que nuestro patrón coincida hasta la parte final de la línea. Se debe añadir al final de nuestra expresión regular.

Tabla 2.8. Ejemplos de patrones con caracteres de posición.

Patrón	*Significado*	*Valores que lo cumplirían*
/^B/	"B" al inicio de la línea.	**B**uenos días, **B**ocadillo de tortilla...
/^Bu/	"Bu" al inicio de la línea.	**Bu**enos días, **Bu**enas tardes...
/o$/	"o" al final de la línea.	Hace viento, Altavoz amplificado...
/to$/	"to" al final de a línea.	Hace vien**to**, Eso está al**to**...

> **Nota**: *Si necesita buscar alguno de los caracteres que se emplean para construir los patrones, es tan sencillo como utilizar el carácter de escape antes de escribirlo. De modo que quedarían expresados como \^, \[o \$, por ejemplo.*

Todos estos caracteres especiales pueden combinarse e incluso un mismo patrón puede escribirse de distintas formas. Por ejemplo, suponiendo que queremos crear un patrón para encontrar un número de dos dígitos, todas las siguientes expresiones regulares serían equivalentes:

```
/[0123456789][0123456789]/
/[0-9][0-9]/
/[0-9]{2}/
/\d\d/
/\d{2}/
```

2.3.2. Usar paréntesis dentro de una expresión regular

Si encerramos entre paréntesis cualquier parte de una expresión regular causaremos que la cadena coincidente se recuerde para poder recuperar su valor más tarde. Esto es útil cuando queremos utilizar los patrones para hacer reemplazos en un texto utilizando parte del que ya existe, como por ejemplo buscar / (vasos) / y sustituirlo por "posavasos" ("vasos" es el valor recordado). Podemos recordar tantos valores como queramos, utilizando una pareja de paréntesis para cada uno de ellos.

Así mismo, podemos aplicar alguno de los caracteres para patrones complejos a la parte delimitada por paréntesis, de forma que su funcionalidad se aplicaría a todo el conjunto. Si tenemos el patrón `/(ja)+/` obtendremos coincidencias con una o varias repeticiones del texto "ja": "ja" o "jajaja".

Para poder recuperar estas coincidencias, tendremos que escribir la siguiente expresión:

- `\x`: Donde `x` es un entero positivo que hace referencia al número de cadena que hemos marcado para ser recordado, contando de izquierda a derecha.

Imaginemos que tenemos el texto "doble clic". Si le aplicamos el patrón `/(\w+) (\w+)/` obtendremos coincidencias con ambas palabras, puesto que buscaríamos uno o más caracteres alfanuméricos, seguido de un espacio en blanco y otra vez uno o más caracteres alfanuméricos. Con esto, las partes de caracteres alfanuméricos serán recordadas para poder ser recuperadas mediante \1 (doble) y \2 (clic) respectivamente.

2.3.3. Modificadores

Una expresión regular puede llevar modificadores o *flags* para definir algunas características a tener en cuenta a la hora de buscar coincidencias. Podemos aplicar a un patrón tantos modificadores como queramos y deben escribirse justo al final, a continuación de la última barra (/) que delimita el patrón.

- g: Fuerza que se sigan buscando coincidencias después de encontrar la primera. Esto puede resultar útil cuando queremos reemplazar la misma cadena varias veces.
- i: Elimina la distinción entre mayúsculas y minúsculas.
- m: Permite usar varios caracteres de posición (^ y $) en el patrón.
- s: Incluye el salto de línea en el comodín punto (.).
- x: Fuerza que los espacios sean ignorados.

Tabla 2.9. Ejemplos de patrones con modificadores.

Patrón	*Significado*	*Valores que lo cumplirían*
/javascript/i	Palabra "javascript" sin distinguir mayúsculas y minúsculas.	**javascript, JavaScript, JAVASCRIPT**

Patrón	*Significado*	*Valores que lo cumplirían*
/(ja)+/gi	Palabra "ja" repetida una o más veces, sin distinguir mayúsculas y minúsculas y todas sus ocurrencias.	**ja**, **Ja**, **JaJa**, **JAJAJA**, **JAjaJAja**...

No se preocupe si no consigue escribir correctamente una expresión regular a la primera. Estas sentencias necesitan práctica y aún así hay ocasiones en las que se vuelve complicado dar con el fallo. Sin embargo, son tremendamente útiles cuando están bien definidas.

2.4. Expresiones regulares útiles

A continuación le dejo algunos patrones, en una forma bastante básica, que se suelen utilizar con frecuencia, aunque le aconsejo que cree los suyos propios para adquirir una mayor práctica.

Tabla 2.10. Expresiones regulares de uso frecuente.

Expresión regular	*Descripción*
/\d{9}/	Número de teléfono.
/\d{8}[a-zA-Z]/	Número de DNI.
/\d{2}-\d{2}-\d{4}/	Fecha (día-mes-año).
/([0][1-9]|1[0-2]):[0-5]\d:[0-5]\d/	Hora (hora:minutos:segundos en formato de 12 horas).
/([01]\d|2[0-3]):[0-5]\d:[0-5]\d/	Hora (hora:minutos:segundos en formato de 24 horas).
/\w+@\w+\.\w{2,3}/	Dirección de correo electrónico.

3

Operadores y conversión entre tipos

3.1. Operadores en JavaScript

Los operadores nos permiten realizar cálculos o acciones sobre las variables o datos que maneja un *script*, devolviendo otro valor como resultado de dicha operación. A esos datos utilizados se les da el nombre de operandos.

JavaScript nos proporciona una buena lista de operadores para poder jugar con los datos.

3.1.1. Operador de asignación

Este operador, identificado por el símbolo igual (=), asigna al operando situado a su izquierda el valor del operando que esté a su derecha, pudiendo ser este último el resultado de un cálculo o una expresión.

De este modo, si escribimos en nuestro código `var miDato = 5;` el operador asignará el valor 5 a la variable `miDato`. También sería posible escribir `var miDato = otroDato;` copiando así el valor que tuviera la variable `otroDato` en la variable `miDato`.

Existen más operadores de asignación pero, como en realidad son formas abreviadas de algunos operadores que veremos a continuación, prefiero presentárselos en su momento como operadores abreviados.

3.1.2. Operadores aritméticos

Estos operadores nos permiten realizar cálculos matemáticos sencillos. Para ello, toman valores numéricos como operandos y devuelven un único valor, también numérico.

Operadores suma, resta y signo

Estos tres operadores se identifican mediante dos símbolos: más (+) y menos (-). Según la posición en que los utilicemos realizarán una operación u otra:

- Si los colocamos entre dos operandos, se realiza la suma o resta entre ellos.
- Si lo colocamos junto a un operando, se cambia el signo de éste.

Veamos esas distintas operaciones y la forma de escribirlas:

Tabla 3.1. Ejemplos de los operadores suma, resta y signo.

Operación	*Ejemplo*	*Resultado*
Suma	var miDato = 3 + 2;	miDato vale 5
Resta	var miDato = 3 - 2;	miDato vale 1
Signo positivo	var miDato = +7;	miDato vale 7
Signo positivo	var miDato = 7;	miDato vale 7 (operador no obligatorio)
Signo negativo	var miDato = -7;	miDato vale -7

Se pueden realizar tantas combinaciones como queramos con estos operadores.

Tabla 3.2. Ejemplos de uso múltiple de los operadores suma, resta y signo.

Operación	*Ejemplo*	*Resultado*
Suma y resta	var miDato = 3 + 2 - 4;	miDato vale 1
Resta y suma	var miDato = 3 – 2 + 9;	miDato vale 10
Suma múltiple	var miDato = 1 + 1 + 1 + 1;	miDato vale 4
Suma positivos	var miDato = 3 + +2;	miDato vale 5
Resta positivos	var miDato = 3 - +2;	miDato vale 1
Suma positivos	var miDato = 3 + -2;	miDato vale 1 y negativos
Suma positivos y negativos	var miDato = -3 + 2 + 6;	miDato vale 5
Suma y resta positivos y negativos	var miDato = 3 - 2 + -1;	miDato vale 0
Suma y resta positivos y negativos	var miDato = 3 + 2 - -1;	miDato vale 6

Operadores incremento y decremento

Como extensión de los operadores anteriores, existe la posibilidad de utilizarlos para sumar (incremento) o restar (decremento) una unidad al valor de un operando, escrito de una forma más breve. Para realizar una de estas operaciones, debemos usar ++ para hacer un incremento y -- para un decremento.

Tabla 3.3. Incremento y decremento.

Operación	*Ejemplo*	*Equivalencia*
Incremento	miDato++;	miDato = miDato + 1;
Decremento	miDato --;	miDato = miDato - 1;

Veamos un ejemplo de cada uno con un pequeño *script*:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var miDato = 3;
   // Incremento
   miDato++;
   // Mostramos su valor
   alert(miDato);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var miDato = 3;
   // Decremento
   miDato--;
   // Mostramos su valor
   alert(miDato);
</SCRIPT>
```

En el primer ejemplo, la variable `miDato` valdrá 4 al hacer el `alert`, mientras que en el segundo valdrá 2.

Operadores suma, resta y signo

Estos tres operadores se identifican mediante dos símbolos: más (+) y menos (-). Según la posición en que los utilicemos realizarán una operación u otra:

- Si los colocamos entre dos operandos, se realiza la suma o resta entre ellos.
- Si lo colocamos junto a un operando, se cambia el signo de éste.

Veamos esas distintas operaciones y la forma de escribirlas:

Tabla 3.1. Ejemplos de los operadores suma, resta y signo.

Operación	*Ejemplo*	*Resultado*
Suma	var miDato = 3 + 2;	miDato vale 5
Resta	var miDato = 3 - 2;	miDato vale 1
Signo positivo	var miDato = +7;	miDato vale 7
Signo positivo	var miDato = 7;	miDato vale 7 (operador no obligatorio)
Signo negativo	var miDato = -7;	miDato vale -7

Se pueden realizar tantas combinaciones como queramos con estos operadores.

Tabla 3.2. Ejemplos de uso múltiple de los operadores suma, resta y signo.

Operación	*Ejemplo*	*Resultado*
Suma y resta	var miDato = 3 + 2 - 4;	miDato vale 1
Resta y suma	var miDato = 3 – 2 + 9;	miDato vale 10
Suma múltiple	var miDato = 1 + 1 + 1 + 1;	miDato vale 4
Suma positivos	var miDato = 3 + +2;	miDato vale 5
Resta positivos	var miDato = 3 - +2;	miDato vale 1
Suma positivos	var miDato = 3 + -2;	miDato vale 1 y negativos
Suma positivos y negativos	var miDato = -3 + 2 + 6;	miDato vale 5
Suma y resta positivos y negativos	var miDato = 3 - 2 + -1;	miDato vale 0
Suma y resta positivos y negativos	var miDato = 3 + 2 - -1;	miDato vale 6

Operadores incremento y decremento

Como extensión de los operadores anteriores, existe la posibilidad de utilizarlos para sumar (incremento) o restar (decremento) una unidad al valor de un operando, escrito de una forma más breve. Para realizar una de estas operaciones, debemos usar ++ para hacer un incremento y -- para un decremento.

Tabla 3.3. Incremento y decremento.

Operación	*Ejemplo*	*Equivalencia*
Incremento	miDato++;	miDato = miDato + 1;
Decremento	miDato --;	miDato = miDato - 1;

Veamos un ejemplo de cada uno con un pequeño *script*:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var miDato = 3;
   // Incremento
   miDato++;
   // Mostramos su valor
   alert(miDato);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var miDato = 3;
   // Decremento
   miDato--;
   // Mostramos su valor
   alert(miDato);
</SCRIPT>
```

En el primer ejemplo, la variable `miDato` valdrá 4 al hacer el `alert`, mientras que en el segundo valdrá 2.

Además, estos operadores se pueden colocar antes o después del operando, generando resultados distintos:

- Si los colocamos antes (prefijo), el valor del operando se modifica antes de leer su contenido. A esto se le conoce como pre-incremento y pre-decremento.
- Si lo colocamos después (sufijo), el valor del operando se modifica después de realizar obtener su contenido. A esto se le conoce como post-incremento y post-decremento.

Puede que esto resulte un poco confuso según se lee, así que preste atención al siguiente *script*:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var num1 = 3;
   var num2 = 3;
   // Mostramos el valor con pre-incremento
   alert(++num1);
   // Mostramos el valor con post-incremento
   alert(num2++);
</SCRIPT>
```

A primera vista podríamos decir que el resultado obtenido en ambos casos es el mismo pero, ¿qué ocurre si ejecutamos ese código? Aunque le parezca mentira, el primer `alert` nos mostrará un 4 mientras que el segundo mostrará un 3. ¿¡Cómo!? ¿¡Por qué!?

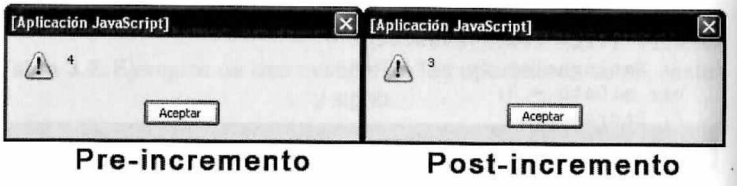


Figura 3.1. Ejemplo con incrementos.

En el primer caso, la variable `num1` es incrementada antes de que `alert` nos muestre su valor. Sin embargo, en el segundo caso la variable `num2` es incrementada después de ser mostrada. Curioso, ¿verdad? Pruebe ahora con el siguiente script:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var numero = 3;
```



```
   // Mostramos el valor con post-incremento
   alert(numero++);
   // Mostramos el valor con otro post-incremento
   alert(numero++);
</SCRIPT>
```

¿Qué cree que ocurrirá? Si analizamos con calma el código veremos que el primer `alert` nos mostrará un 3, incrementando seguidamente el valor de la variable `numero`. Por ello, el segundo `alert` mostrará un 4 y después volverá a incrementar la variable.

Vamos ahora con un último ejemplo para reforzar nuestros conocimientos sobre el funcionamiento de estos operadores tan traviesos:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var numero = 3;
   // Mostramos el valor con pre-incremento
   alert(++numero);
   // Mostramos el valor con post-incremento
   alert(numero++);
</SCRIPT>
```

¡Buf! ¿Y ahora que saldrá? Calma... vayamos siempre paso a paso.

Con el primer `alert` se incrementará la variable `numero` antes de ser mostrada, por lo que veremos un 4. A continuación volvemos a mostrar la variable, que será incrementada después, así que veremos... ¡otro 4!

Operadores multiplicación y división

Las operaciones de multiplicación y división se representan con un asterisco (*) y una barra (/) respectivamente. Para utilizarlos basta con ponerlos entre dos operadores, tal y como hemos visto con la suma y la resta.

Tabla 3.4. Ejemplos de los operadores multiplicación y división.

Operación	*Ejemplo*	*Resultado*
Multiplicación	var miDato = 6 * 2;	miDato vale 12
División	var miDato = 6 / 2;	miDato vale 3

Por supuesto, también se pueden encadenar varias operaciones con estos operadores: `var miDato = 6 / 2 * 3`, dando como resultado un 9.

Operador resto o módulo

Este operador, a través del símbolo de porcentaje (%), permite obtener el resto resultante de la división de dos operandos. Este resto también es conocido como el módulo del dividendo sobre el divisor.

Tabla 3.5. Ejemplos del operador resto o módulo.

Ejemplo	*Resultado*
var miDato = 6 % 2;	miDato vale 0
var miDato = 7 % 2;	miDato vale 1

Figura 3.2. Representación del módulo.

Operadores aritméticos abreviados

Como le adelanté al explicar el operador de asignación, en JavaScript es posible escribir de forma abreviada el uso de cualquiera de estos operadores aritméticos para hacer nuestro código más rápido de escribir. Solamente interviene un único operando, situado a la derecha. De este modo podremos hacer el cálculo (suma, división, etc.) y su asignación de un solo golpe obteniendo el mismo resultado. En la tabla 3.6 podrá ver estos operadores abreviados y su equivalencia en forma "larga".

Tabla 3.6. Ejemplos de operadores aritméticos abreviados.

Ejemplo	*Equivalencia*
var miDato += 5;	var miDato = miDato + 5;
var miDato -= 5;	var miDato = miDato - 5;
var miDato *= 5;	var miDato = miDato * 5;
var miDato /= 5;	var miDato = miDato / 5;
var miDato %= 5;	var miDato = miDato % 5;

También sería posible usar una variable como operando de este operador.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var numero1 = 3;
   var numero2 = 5;
   // Suma con operador abreviado
   numero1 += numero2;
   // Mostramos el valor (numero1 vale 8)
   alert(numero1);
</SCRIPT>
```

3.1.3. Operador sobre cadenas

Este apartado será bastante corto, puesto que sólo existe un operador que actúe sobre las cadenas con el que se consigue unir o concatenar dos cadenas. Se representa con el mismo símbolo del operador aritmético de suma: más (+).

Esta coincidencia no supone un problema para JavaScript que realizará una operación u otra en función del tipo de los operandos: suma si son números y concatena si son cadenas.

Tabla 3.7. Ejemplos del operador sobre cadenas.

Ejemplo	*Resultado*
var miDato = "Java" + "Script";	miDato vale "JavaScript"
var miDato = "Capítulo " + "3";	miDato vale "Capítulo 3"
var miDato = "Im " + "pre" + "sio" + "nante";	miDato vale "Impresionante"

Si intentamos concatenar números con cadenas, JavaScript interpreta todos los operandos como si fuesen cadenas de texto.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var x = 6;
   // Unión de número con cadena (resultado "10 metros")
   alert(10 + " metros");
   // Unión de cadena con número (resultado "Capítulo 3")
   alert("Capítulo " + 3);
   // Unión de cadena con variable número (resultado
   // "Capítulo 6")
   alert("Capítulo " + x);
</SCRIPT>
```

3.1.4. Operadores lógicos

Estos operadores se utilizan para realizar operaciones con dos únicos resultados posibles: verdadero (`true`) o falso (`false`). Esto nos será muy útil para tomar decisiones dentro de nuestros *scripts*.

Los operandos que se utilicen junto con uno de estos operadores serán evaluados siempre como un valor booleano, para así poder devolver otro valor del mismo tipo.

> **Nota:** *Si tiene curiosidad desde ya por saber cómo se evalúa cualquier tipo de variable como un booleano, puede echar un vistazo a algunos ejemplos en la tabla 7.1 del capítulo 7.*

Operador NO o de negación (NOT)

Este operador se representa con el símbolo de fin de exclamación (`!`) y nos permite invertir el valor del operando que le acompañe.

Tabla 3.8. Funcionamiento del operador lógico NO o de negación.

Valor operando	*Ejemplo*	*Resultado*
var x = false;	var y = !x;	y vale true
var x = true;	var y = !x;	y vale false

Operador Y (AND)

Este operador se representa con dos símbolos *ampersand* (`&&`) y devuelve un valor `true` si sus dos operandos también lo son, o un valor `false` en otro caso.

Tabla 3.9. Funcionamiento del operador lógico Y.

Valor operando 1	*Valor operando 2*	*Ejemplo*	*Resultado*
var x = false;	var y = false;	var z = x && y;	z vale false
var x = true;	var y = false;	var z = x && y;	z vale false
var x = false;	var y = true;	var z = x && y;	z vale false
var x = true;	var y = true;	var z = x && y;	z vale true

Operador O (OR)

Este operador se representa con dos barra verticales (`||`) y devuelve un valor `true` si al menos uno de sus dos operandos lo son. En cualquier otro caso nos da un valor `false`.

Tabla 3.10. Funcionamiento del operador lógico O.

Valor operando 1	*Valor operando 2*	*Ejemplo*	*Resultado*
var x = false;	var y = false;	var z = x || y;	z vale false
var x = true;	var y = false;	var z = x || y;	z vale true
var x = false;	var y = true;	var z = x || y;	z vale true
var x = true;	var y = true;	var z = x || y;	z vale true

3.1.5. Operadores condicionales o de comparación

Los operadores condicionales nos servirán para comparar los valores de sus dos operandos, devolviendo un valor booleano (`true` o `false`). Junto con los operadores lógicos, son también muy útiles a la hora de tomar decisiones en nuestro código.

Operadores de igualdad y desigualdad

El operador de igualdad (==) y desigualdad (`!=`) nos permiten comprobar si los valores de dos operandos son iguales o no, independientemente del tipo que sean dichos operandos. El resultado obtenido es `true` cuando la igualdad o desigualdad se cumple y `false` en caso contrario.

> **Advertencia:** *No confundir el operador de asignación (=) con el de igualdad (==), ya que su función es bien distinta.*

Tabla 3.11. Funcionamiento de los operadores de igualdad y desigualdad.

Valor operando 1	*Valor operando 2*	*Ejemplo*	*Resultado*
var x = 5;	var y = 5;	var z = x == y;	z vale true
var x = 5;	var y = 5;	var z = x != y;	z vale false
var x = 9;	var y = 5;	var z = x == y;	z vale false

Valor operando 1	Valor operando 2	Ejemplo	Resultado
var x = 9;	var y = 5;	var z = x != y;	z vale true
var x = "hola";	var y = "hola";	var z = x == y;	z vale true
var x = "hola";	var y = "hola";	var z = x != y;	z vale false
var x = "libro";	var y = "hola";	var z = x == y;	z vale false
var x = "libro";	var y = "hola";	var z = x != y;	z vale true
var x = 5;	var y = "5";	var z = x == y;	z vale true
var x = 5;	var y = "5";	var z = x != y;	z vale false

En los dos últimos ejemplos de la tabla 3.11 el número 5 y la cadena "5" tienen el mismo valor para JavaScript por la conversión de tipos que éste realiza (cosa que veremos más adelante en este capítulo).

Por otro lado, tenemos los operadores de igualdad estricta (===, tres signos de igual) y desigualdad estricta (!==), que funcionan de la misma manera que los que acabamos de explicar, pero con la particularidad de que también comprueban si los tipos de sus operandos coinciden o no. Por tanto, se obtiene un valor `true` cuando el valor y el tipo cumplen la igualdad o desigualdad, y valor `false` en caso contrario. Veamos los mismos ejemplos que la tabla 3.11 pero aplicando estos nuevos operadores.

Tabla 3.12. Funcionamiento de los operadores de igualdad y desigualdad estricta.

Valor operando 1	Valor operando 2	Ejemplo	Resultado
var x = 5;	var y = 5;	var z = x === y;	z vale true
var x = 5;	var y = 5;	var z = x !== y;	z vale false
var x = 9;	var y = 5;	var z = x === y;	z vale false
var x = 9;	var y = 5;	var z = x !== y;	z vale true
var x = "hola";	var y = "hola";	var z = x === y;	z vale true
var x = "hola";	var y = "hola";	var z = x !== y;	z vale false
var x = "libro";	var y = "hola";	var z = x === y;	z vale false
var x = "libro";	var y = "hola";	var z = x !== y;	z vale true
var x = 5;	var y = "5";	var z = x === y;	z vale false
var x = 5;	var y = "5";	var z = x !== y;	z vale true

Conforme a la tabla 3.12, puede comprobar cómo hemos obtenido los mismos resultados salvo en los dos últimos ejemplos donde, con estos operadores estrictos, sí se ha tenido en cuenta el tipo de datos para los valores 5 y "5" (número y cadena respectivamente).

Operadores mayor y menor

El operador mayor (>) nos devuelve un valor `true` si el operando de la izquierda tiene un valor estrictamente mayor que el de la derecha. El operador menor (<), por el contrario, nos devuelve `true` si el valor izquierdo es estrictamente menor que el derecho.

> **Nota:** *Si ambos valores coincidiesen, se devolvería `false` ya que un valor no es estrictamente mayor o menor que el otro.*

En el caso de tener operandos numéricos creo que el resultado de la comparación está claro, pero si tenemos cadenas la cosa no es tan obvia. Para determinar si una cadena es mayor o menor que otra, se comparan letra a letra (empezando por la izquierda) hasta encontrar una diferencia entre las dos cadenas. Para comparar dos letras se siguen estas reglas:

1. Las mayúsculas son menores que las minúsculas ("A" es menor que "a").
2. Las primeras letras del abecedario son menores que las últimas ("a" es menor que "c").
3. Los números (expresados como cadena) son menores que las letras ("6" es menor que "A").

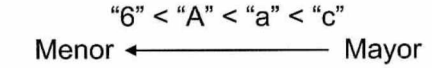


Figura 3.3. Orden en la comparación de cadenas.

Tabla 3.13. Funcionamiento de los operadores mayor y menor.

Valor operando 1	Valor operando 2	Ejemplo	Resultado
var x = 5;	var y = 9;	var z = x > y;	z vale false
var x = 5;	var y = 9;	var z = x < y;	z vale true
var x = 9;	var y = 5;	var z = x > y;	z vale true

Valor operando 1	*Valor operando 2*	*Ejemplo*	*Resultado*
var x = 9;	var y = 5;	var z = x < y;	z vale false
var x = 5;	var y = 5;	var z = x > y;	z vale false
var x = 5;	var y = 5;	var z = x < y;	z vale false
var x = "hola";	var y = "Hola";	var z = x > y;	z vale true ("h" mayor que "H")
var x = "hola";	var y = "adios";	var z = x > y;	z vale true ("h" mayor que "a")
var x = "hola";	var y = "adios"; ("a" "h")	var z = x < y;	z vale false menor que
var x = "hola";	var y = "hoy";	var z = x > y;	z vale false ("l" menor que "y")
var x = "hola";	var y = "hoy";	var z = x < y;	z vale true ("y" mayor que "l")

También existe una variación de estos operadores que añade la operación de igualdad a la comparación para el caso de encontrarnos con valores iguales. Con ellos podremos determinar si un operando es mayor, o igual (>=), o menor, o igual (<=) que el otro.

Tabla 3.14. Funcionamiento de los operadores mayor, o igual y menor o igual.

Valor operando 1	*Valor operando 2*	*Ejemplo*	*Resultado*
var x = 5;	var y = 9;	var z = x >= y;	z vale false
var x = 5;	var y = 9;	var z = x <= y;	z vale true
var x = 5;	var y = 5;	var z = x >= y;	z vale true
var x = 5;	var y = 5;	var z = x <= y;	z vale true

3.1.6. Operadores sobre bits o binarios

En este apartado trataremos los operadores que utilizan operandos expresados en sistema binario (unos y ceros). Si esto le `suena a chino´, no se preocupe que por el mismo precio también incluimos una pequeña explicación de este sistema.

Introducción al sistema binario

Antes de comenzar a listar los operadores, creo que debemos hacer esta pequeña introducción al mundo del sistema binario para que pueda comprender bien todas las operaciones y ejemplos.

Por si le `suena a chino´, un bit (*Binary digIT*) es un dígito del sistema binario (base 2), que únicamente toma los valores cero (`0`) o uno (`1`). La agrupación de bits representa un número.

Para representar un número decimal en binario debemos dividir primeramente ese número entre la base (2 en este caso) y después dividir repetidamente los cocientes resultantes entre la base (2), hasta que dicho cociente sea cero (no se pueden hacer más divisiones). Los restos de todas estas divisiones serán los que formen los bits del número binario. Veamos un rápido ejemplo de cómo convertir el número decimal 13 en binario:

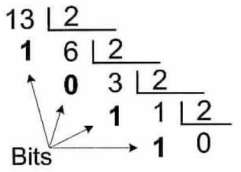


Figura 3.4. Conversión de decimal a binario.

Por tanto, el número 13 en binario se representa con los bits 1101 (se empieza por el último resto obtenido).

Para realizar la conversión inversa (binario a decimal), debe ver la posición de los bits como exponentes de la base (2), siendo cero la posición del bit situado más a la derecha representando la potencia 2^0. El número decimal se obtiene sumando todas las potencias mientras que el valor del bit (1 ó 0) correspondiente indica si debe incluirse esa potencia en la suma. Veamos esto mejor con un ejemplo, convirtiendo a decimal el número binario 1101 anterior:

Bits	1	1	0	1
Posición bit (exponente)	3	2	1	0
Potencia	2^3	2^2	-	2^0
Valor potencia	8	4	0	1

Figura 3.5. Conversión de binario a decimal.

Si sumamos los valores de las potencias (8 + 4 + 0 + 1) obtenemos el número decimal equivalente, que es 13. La potencia 2^1 no se incluye ya que el valor del bit en esa posición es 0.

Debe saber que para poder representar, tanto los números negativos como positivos, se reserva el bit situado más a la izquierda para indicar el signo (0 para los positivos y 1 para los negativos). De modo que para el caso del número binario 1101 (13 en decimal) su representación con signo sería **0**1101.

Desgraciadamente, tal y como maneja JavaScript los números binarios, interpretar un número positivo o negativo no es tan sencillo como mirar el bit situado más a la izquierda y calcular el decimal equivalente con los bits restantes (-13 no es **1**1101). Para convertir un positivo en negativo o lo contrario existen varios métodos, pero JavaScript utiliza el que se conoce como complemento a dos, que consiste en aplicar al número dos operaciones:

1. Invertir el valor de cada bit que forma el número, es decir, cambiar los unos por ceros y viceversa. Esta operación se conoce como negación o complementación y la veremos como un operador de JavaScript.
2. Se suma 1 al resultado de la operación anterior.

Procedamos a ver cómo se representa entonces el número decimal -13 a partir de su positivo en binario (01101):

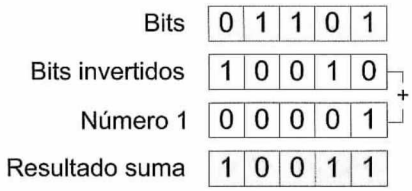


Figura 3.6. Conversión de número binario positivo en negativo.

De este modo vemos que el número decimal -13 se representa en binario como 10011 en complemento a dos.

Si va a utilizar los operadores binarios con frecuencia le aconsejo que se documente de forma más amplia acerca del manejo de número binarios, puesto que aquí se ha expuesto lo más básico para que pueda comprender los ejemplos.

> **Nota**: *Las operaciones con números binarios se realizan bit a bit y no con números completos, por lo que el resultado será la combinación de cada operación individual a nivel de bit.*

Por último, los operadores binarios de JavaScript representan cada número con 32 bits (31 para el número y 1 para el signo). En caso de que un número no ocupe los 31 bits, se repite el bit de signo en esos bits no usados para que el complemento a dos siga funcionando. De este modo el número decimal 13 (01101 en binario) tendrá 28 bits a 0 a la izquierda (el bit de signo y 27 réplicas) quedando como 00000000000000000000 00000001101. En el caso del decimal -13 (10011) se tendrán 28 bits a 1 (signo y 27 réplicas) quedando expresado como 11111 11111111111111111111110011.

Para darles más claridad a los ejemplos, se omiten las réplicas del signo siempre que se pueda.

Operador de negación o complementación (NOT)

Este primer operador binario, como hemos avanzado hace unas líneas, nos permite invertir el valor de cada bit que forma un número, cambiando los unos por ceros y viceversa. El símbolo que lo representa es la tilde o virgulilla (~) y debe ir colocada delante del número.

> **Advertencia**: *Este operador no calcula el valor negativo de un número, sino su complemento (bits invertidos). Los números negativos en JavaScript se obtienen mediante el complemento a dos.*

En el siguiente ejemplo, veremos cuál es el resultado de complementar el número decimal 13:

Bit signo	Bit 3	Bit 2	Bit 1	Bit 0	Número decimal
0	1	1	0	1	13
1	0	0	1	0	**-14**

Como puede ver, el valor obtenido no es -13 sino el complemento de 13, que es -14. Ahora podrá darse cuenta de por qué hay que sumar 1 al utilizar el complemento a dos para obtener el negativo de un número.

Operador Y (AND)

El comportamiento de este operador es el mismo que el operador lógico Y (&&), salvo que éste se represente con un único *ampersand* (&) y nos devuelva un 1 cuando los dos bits que se comparan son también un 1.

Tabla 3.15. Funcionamiento del operador binario Y.

Valor bit 1	*Valor bit 2*	*Resultado*
0	0	0
1	0	0
0	1	0
1	1	1

Recuerde que las operaciones se hacen bit a bit y el resultado de cada operación individual se almacena en la misma posición. Veamos como ejemplo el resultado que ofrece la operación `5&3`:

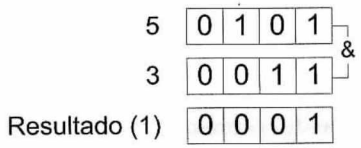


Figura 3.7. Ejemplo del operador binario Y.

Como ha podido observar, el resultado de aplicar el operador Y a los bits de la derecha ha dado como resultado un 1, mientras que en el resto ha sido un 0. Esto nos ha dejado una secuencia nueva de bits que representa el número uno.

Operador O (OR)

Al igual que ocurre con el operador binario Y (`&`), este también se comporta como su análogo lógico (`||`), representándose con una barra vertical (`|`) y devolviendo un 1 cuando alguno de los dos bits que se comparan es también un 1.

Tabla 3.16. Funcionamiento del operador binario O.

Valor bit 1	*Valor bit 2*	*Resultado*
0	0	0
1	0	1
0	1	1
1	1	1

Veamos como ejemplo el resultado que nos presenta la operación `5 | 3`:

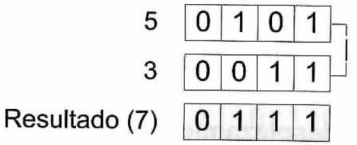


Figura 3.8. Ejemplo del operador binario O.

Esta vez, el resultado de la operación nos ha dejado un 0 en la posición de la izquierda y el resto con 1. Esto nos da el número decimal 7.

Operador O exclusivo (XOR)

Este operador se representa mediante el acento circunflejo (^) y nos devuelve un 1 cuando únicamente uno de los bits también es un 1.

Tabla 3.17. Funcionamiento del operador binario O exclusivo.

Valor bit 1	*Valor bit 2*	*Resultado*
0	0	0
1	0	1
0	1	1
1	1	0

Veamos como ejemplo el resultado que nos presenta la operación `5^3`:

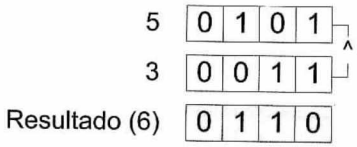


Figura 3.9. Ejemplo del operador binario O exclusivo.

Tanto el bit de la izquierda como la derecha se han convertido en un 0, dejando un 1 en las posiciones intermedias dando lugar al número 6.

Habrá notado que los resultados no tienen mucha lógica desde el punto de vista de números decimales (5 | 3 devuelve 7). No se preocupe ya que rara vez se analiza el resultado de esta forma. Normalmente se mira con detenimiento el resultado obtenido en las posiciones de los bits (por ejemplo, si hay un 1 en la posición 3).

Operadores de desplazamiento

Estos operadores van acompañados de dos operandos: el número cuyos bits se van a desplazar y la cantidad de posiciones que se quieren desplazar esos bits, representado por un número entero. La dirección en la que debe realizarse el desplazamiento la indica el operador, de modo que tenemos los siguientes:

- Desplazamiento a la izquierda (<<): se encarga de desplazar los bits hacia la izquierda el número de posiciones indicado, rellenando con ceros los espacios de la derecha resultantes de este desplazamiento.
- Desplazamiento a la derecha con propagación de signo (>>): desplaza los bits hacia la derecha tantas posiciones como hayamos indicado, descartando los bits que sobran por la derecha y conservando el signo. Para conservar el signo lo que se hace es repetir el bit de signo por la izquierda tantas veces como desplazamientos hayamos hecho.
- Desplazamiento a la derecha con relleno de ceros (>>>): realiza la misma operación que el anterior pero rellenando con ceros los espacios de la izquierda y sin conservar el signo. El resultado en los números positivos es el mismo que con el operador >>.

Veamos ahora unos ejemplos con cada operador, para comprender su funcionamiento::

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var numero = 11;
   // Desplazamiento a la izquierda
   alert(numero << 2);
   // Asignación de valor
   numero = -11;
   // Desplazamiento a la izquierda
   alert(numero << 2);
<SCRIPT>
```

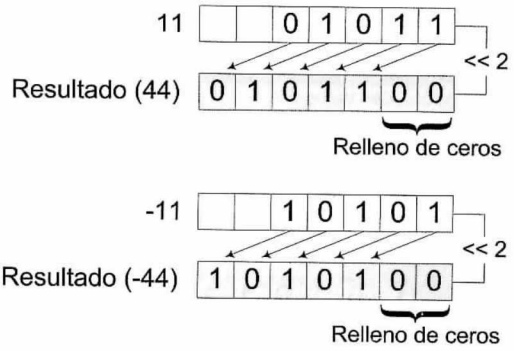



Figura 3.10. Ejemplos de desplazamiento a la izquierda.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var numero = 11;
   // Desplazamiento a la derecha con propagación
   // de signo
   alert(numero >> 2);
   // Asignación de valor
   numero = -11;
   // Desplazamiento a la derecha con propagación
   // de signo
   alert(numero >> 2);
<SCRIPT>
```

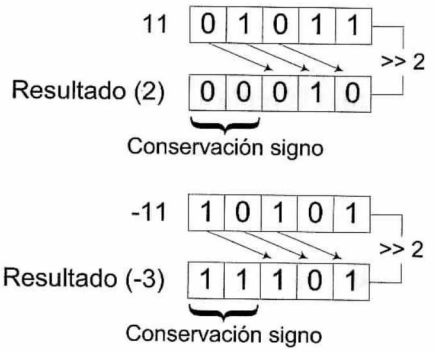


Figura 3.11. Ejemplos de desplazamiento a la derecha con propagación de signo.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var numero = 11;
```

```
    // Desplazamiento a la derecha con relleno
    // de ceros
    alert(numero >>> 2);
    // Asignación de valor
    numero = -11;
    // Desplazamiento a la derecha con relleno
    // de ceros
    alert(numero >>> 2);
</SCRIPT>
```

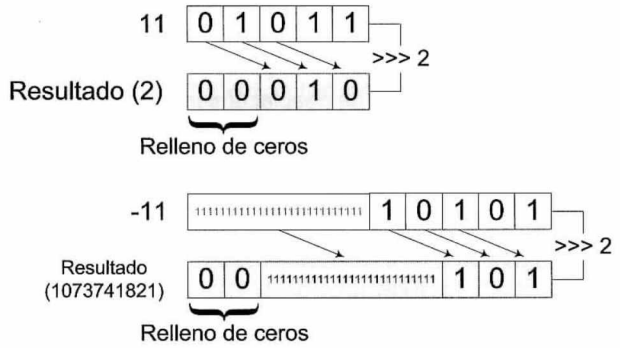


Figura 3.12. Ejemplos de desplazamiento a la derecha con relleno de ceros.

Puede que le choque esa fila de unos del último ejemplo. La explicación está en la breve introducción al sistema binario, escrita hace unas páginas, donde se dijo que estos operadores manejan los números como conjuntos de 32 bits y que se omitiría la réplica del signo siempre que fuera posible para que su lectura fuera más fácil. En el ejemplo, el número -11 representado con todos sus bits queda como 11111111111111111111111111110101 (27 réplicas de signo). Al desplazar los bits dos posiciones a la derecha y rellenar con ceros obtenemos el número binario 0011111111111111 1111111111111101.

Operadores sobre bits abreviados

Como ya ocurría con los operadores aritméticos, existe también la posibilidad de escribir los operadores sobre bits de una forma abreviada, de modo que la operación binaria y su asignación pueden realizarse a través de un solo operador de asignación, como puede ver en la tabla 3.18 que mostramos a continuación.

Tabla 3.18. Ejemplos de operadores sobre bits abreviados.

Ejemplo	*Equivalencia*
var a &= 5;	var a = a & 5;
var a |= 5;	var a = a | 5;
var a ^= 5;	var a = a ^ 5;
var a <<= 2;	var a = a << 2;
var a >>= 2;	var a = a >> 2;
var a >>>= 2;	var a = a >>> 2;

3.1.7. Operadores especiales

JavaScript posee una serie de operadores que no encajan en las categorías anteriores ya que no realizan un cálculo sobre los datos.

Operador condicional

Estamos frente al único operador que tiene tres operandos, escribiéndose de la siguiente forma:

```
<SCRIPT TYPE="text/javascript">
    // Operador condicional
    operando_1? operando_2 : operando_3;
</SCRIPT>
```

El operando 1 es en realidad una condición que corresponde a una expresión que puede valer `true` o `false`. En base a él, este operador devuelve el operando 2 si dicha condición se cumple, o el operando 3 en caso contrario.

Para añadir mayor claridad al código, la expresión del operando 1 se suele encerrar entre paréntesis.

Veamos cómo utilizar este operador dentro de un *script*:

```
<SCRIPT TYPE="text/javascript">
    // Declaración de variables
    var edad = 16;
    var mayoriaEdad;
    // Uso operador condicional
    mayoriaEdad = (edad >= 18)? "sí" : "no";
    // Mostramos su valor
    alert(mayoriaEdad);
</SCRIPT>
```

Si escribimos este código, y lo ejecutamos en nuestro navegador, veremos cómo la variable `mayoriaEdad` recibe el valor `"no"` ya que la expresión `(edad >= 18)` toma valor `false`.

Operador coma

Este operador (,) se coloca entre dos operandos con la única finalidad de que ambos sean evaluados y que el valor del segundo sea devuelto por el operador.

Veremos, en el siguiente capítulo, su uso dentro de los bucles `for`, ya que es el sitio donde se emplea principalmente.

Operador void

Cuando se usa este operador junto con una expresión, lo que se consigue es que dicha expresión sea ejecutada pero sin devolver ningún valor, es decir, la operación se realiza pero el resultado no puede ser almacenado ya que se descarta.

La expresión que evalúa este operador puede ser escrita a continuación del mismo o usando paréntesis, de manera que `void miExpresion` es lo mismo que `void(miExpresion)`.

Veamos qué pasa si lo utilizamos dentro de un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var suma;
   // Uso operador void
   suma = void(1 + 2);
   // Mostramos su valor
   alert(suma);
   // Uso operador suma
   suma = 1 + 2;
   // Mostramos su valor
   alert(suma);
</SCRIPT>
```

El resultado de ejecutar este código es que la expresión `1 + 2` es evaluada, pero su valor no es devuelto a la variable `suma`, por lo que en el primer `alert` ésta tiene valor `undefined` dado que todavía no ha sido inicializada. Sin embargo, en el segundo `alert` vemos que tiene valor 3 como resultado de la operación de suma realizada justo antes.

Operador typeof

Mediante este operador podremos conocer el tipo de dato del operando que pongamos a continuación. Este operando corresponde a una expresión, que no es evaluada, y su tipo se devuelve como una cadena de texto.

Al igual que ocurre con el operador `void`, el operando puede ir encerrado entre paréntesis o no, siendo lo mismo `typeof miOperando` que `typeof(miOperando)`.

En la tabla 3.19 podremos ver los tipos que puede devolver este operador, junto a un pequeño ejemplo.

Tabla 3.19. Tipos devueltos por el operador typeof.

Tipo devuelto	*Operando esperado*	*Ejemplo*
"number"	Cualquier número (entero o real)	typeof(5) o typeof(1 + 2)
"boolean"	Valor booleano	typeof(true) o typeof(1 < 2)
"string"	Cadena de texto	typeof("hola") o typeof("ho" + "la")
"function"	Función u objeto predefinido	typeof(alert)
"object"	Objeto no predefinido	typeof(miObjeto)
"undefined"	Operando no inicializado	typeof(miVariable-Nueva)

Nota: *Si quiere comprobar el tipo de un operando tenga en cuenta que está manejando cadenas. Por tanto debe escribir algo como* `typeof(5) == "number"`.

Operadores sobre objetos

Por último, existen otros operadores en JavaScript que se utilizan para manejar objetos. En este capítulo simplemente los nombraremos, ya que se explicarán con más detalle en el capítulo correspondiente a los objetos.

- El punto (.): Sirve para obtener el valor de una propiedad o ejecutar un método del objeto que le precede.
- Los corchetes ([ ]): Sirven para acceder al valor de una propiedad o método del objeto que le precede.
- `new`: Nos permite crear una instancia del objeto que indiquemos.
- `delete`: Se utiliza para borrar una propiedad de una instancia.
- `in`: Nos permite saber si una propiedad existe en el objeto que indiquemos.
- `instanceof`: Nos dice si la instancia especificada corresponde al objeto que indiquemos.
- `this`: Sirve para referirse a un objeto como sustituto de su nombre.

3.1.8. Precedencia de los operadores

Todos los operadores que hemos visto hasta ahora pueden utilizarse conjuntamente y las veces que se necesiten. Consecuentemente, es necesario establecer un orden de ejecución o precedencia entre ellos ya que, por ejemplo, la expresión `5 * 2 - 1` puede ser interpretada de distintas maneras:

1. Si consideramos que primero se ejecuta la resta (`2 - 1`) y, con ese resultado, se ejecuta después la multiplicación (`5 * 1`), entonces tenemos el valor 5 como resultado.
2. Si ejecutamos primero la multiplicación (`5 * 2`) y a continuación, con ese resultado, la resta (`10 - 1`), entonces tendremos como resultado el valor 9.

¿Cuál es la interpretación válida? Si miramos la tabla 3.20 nos daremos cuenta de que el resultado correcto es 9, ya que la multiplicación tiene mayor precedencia que la resta.

Tabla 3.20. Precedencia de operadores.

Precedencia	*Operadores*
1	. []
2	() new
3	! ~ - (signo menos) + (signo más) ++ -- typeof void delete
4	* / %
5	+ (suma) - (resta)
6	<< >> >>>
7	< <= > >= in instanceof
8	== != === !==
9	&
10	^
11	|
12	&&
13	||
14	?:
15	= += -= *= /= %= &= ^= |= <<= >>= >>>=
16	,

Aquí puede jugar un papel importante el uso de los paréntesis, ya que con ellos podremos alterar el orden de precedencia de los operadores. Por ejemplo, si escribimos la expresión `(5 * 2) - 1` obtendremos como resultado un 9, mientras que si dejamos la expresión como `5 * (2 - 1)` tendremos un 5. Esto es así porque primero se ejecuta la parte que está entre paréntesis al tener mayor precedencia que el resto de operadores.

3.2. Conversión entre tipos

Como habrá podido ver en algunos de los ejemplos del apartado anterior, a veces es posible combinar dos operandos de distinto tipo sin obtener ningún tipo de error por ello. ¿Cómo sabe JavaScript que, por ejemplo, la cadena "5" puede ser interpretada de la misma manera que el número 5? Esto es posible gracias a la conversión interna entre tipos que posee JavaScript, pudiendo ser de dos tipos: implícita y explícita.

3.2.1. Conversión implícita

Esta conversión se realiza automáticamente por JavaScript cuando detecta que los operandos que recibe un operador tienen distinto tipo. Esta conversión no siempre es posible ("hola" no tiene equivalencia como número) así que en ocasiones el resultado es impredecible (¡JavaScript hace lo que puede!).

El único operador que tiene un comportamiento más especial es la suma (+) puesto que ya hemos visto que puede utilizarse tanto para sumar números como para concatenar cadenas. Si en una misma suma utilizamos valores numéricos y cadenas, JavaScript siempre lo va a interpretar como una concatenación de cadenas. Pongamos como ejemplo el siguiente código:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var cadena = "6";
   var numero = 4;
   // Primera operación
   alert(cadena + numero);
   // Segunda operación
   alert(numero + cadena);
   // Tercera operación
   alert(numero + numero);
```

```
   // Cuarta operación
   alert(cadena + cadena);
</SCRIPT>
```

En las dos primeras operaciones, el resultado será una cadena ("64" y "46" respectivamente). En la tercera, como todos los operandos son números, el resultado será un número (8). Finalmente, en la última operación, obtendremos de nuevo una cadena ("66").

En el resto de operaciones aritméticas, las cadenas intentarán ser convertidas en un número para poder devolver un valor numérico. Por ejemplo:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var cadena = "6";
   var numero = 2;
   // Primera operación
   alert(cadena - numero); // mostrará un 4
   // Segunda operación
   alert(numero * cadena); // mostrará un 12
   // Tercera operación
   alert(cadena / numero); // mostrará un 3
</SCRIPT>
```

Otro pequeño caso especial ocurre cuando se intenta transformar un valor booleano en un número. Para este tipo de datos, los valores `true` se interpretan como el número 1 y los `false` como un 0.

3.2.2. Conversión explícita

Dado que hay ocasiones en que la conversión implícita no ofrece un resultado predecible ni deseado, es preciso que intervenga la mano del programador para realizar estas conversiones cuando sea necesario. Para ello, dispone de las funciones predefinidas `parseInt` y `parseFloat`, las cuales están detalladas en el capítulo 5, dedicado a las funciones, aunque le dejaré un pequeño ejemplo para que vea cómo influye en el resultado:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var cadena = "6";
   var numero = 4;
   // Primera operación
   alert(parseInt(cadena) + numero);
   // Segunda operación
   alert(numero + parseInt(cadena));
</SCRIPT>
```


En las dos operaciones, gracias a la intervención de la conversión explícita, el resultado es el mismo: el número 10. Si no utilizásemos estas funciones, JavaScript aplicaría la conversión implícita explicada en el apartado anterior, dándonos como resultados las cadenas "64" y "46".

Para terminar, me gustaría recordarle que en JavaScript las variables no tienen un tipo definido, por lo que puede almacenar el resultado de una conversión en la misma variable sin provocar ningún error en su código.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var cadena = "6";
   // Conversión explícita
   cadena = parseInt(cadena);
   // Mostramos el valor (número 6)
   alert(cadena);
 </SCRIPT>
```

4 Estructuras de control

Los ejemplos vistos hasta ahora han sido bastante "sosillos" ya que nos hemos limitado a ejecutar instrucciones en orden secuencial. No se desanime, JavaScript no nos va a decepcionar y por ello nos brinda una colección de estructuras de control con las que podremos adaptar nuestro código a las múltiples necesidades que nos vayan surgiendo, permitiéndonos ser un poco más dueños de las instrucciones que se van a ejecutar mediante la toma de decisiones (por ejemplo, si una variable sobrepasa un valor) o repetir una operación varias veces con la inclusión de bucles (realizar la suma de los números que van del 1 al 30).

Todas las instrucciones que se quieran ejecutar dentro de una estructura de control deben ir encerradas entre llaves ({ }).

```
<SCRIPT TYPE="text/javascript">
   estructura-de-control {
      // Mis instrucciones
   }
</SCRIPT>
```

Truco: *En JavaScript no es obligatorio usar las llaves si nuestro bloque consta de una única instrucción.*

Además, toda estructura de control puede ir anidada dentro de cualquier otra estructura, de forma que sería posible, por ejemplo, tomar una decisión dentro de un bucle o ejecutar un bucle dentro de otro (veremos que es útil para recorrer los datos de una tabla).

Pasemos ahora al detalle de cada una de estas estructuras, agrupadas por categorías: condicionales, de bucle y, por último, de manipulación de objetos.

4.1. Estructuras condicionales

Este tipo de estructuras evalúan una condición que, de cumplirse, permitirán que se ejecute su bloque de instrucciones asociado. En caso de no cumplirse dicha condición el bloque tampoco se ejecutará, por tanto es como si esa porción de código no existiera.

4.1.1. Sentencia if - else

La sentencia `if` ejecuta su bloque de instrucciones si la condición que la sigue se cumple, esto es, si la condición es verdadera. La sintaxis básica sería esta:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia if
   if (condición) {
      // Mi bloque de instrucciones
   }
</SCRIPT>
```

Ilustrado con un pequeño ejemplo, quedaría así:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia if con condición verdadera
   if (5 > 2) {
      // Mi bloque de instrucciones
   }
   // Sentencia if con condición falsa
   if (5 > 9) {
      // Mi bloque de instrucciones
   }
</SCRIPT>
```

Opcionalmente podremos usar la sentencia `else` combinada con una sentencia `if` para ejecutar otro bloque en caso de que la condición no se cumpla. El resultado de la unión de ambas es que siempre se ejecute uno (y sólo uno) de los dos bloques. Veamos la sintaxis si utilizamos ambas sentencias:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia if - else
   if (condición) {
      // Mi bloque de instrucciones (condición verdadera)
   } else {
      // Otro bloque de instrucciones (condición falsa)
   }
</SCRIPT>
```

Quizá vea más clara esta explicación mediante un ejemplo:


```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var edad = 19;
   // Sentencia if - else
   if (edad >= 18) {
      alert("Soy mayor de edad");
   } else {
      alert("No soy mayor de edad");
   }
</SCRIPT>
```

Si ejecutamos este *script* veremos en nuestra pantalla el mensaje "Soy mayor de edad". Esto es así puesto que la condición `(edad >= 18)` se cumple, quedando descartado el bloque de la sentencia `else`. Si la edad fuese, por ejemplo, 13 entonces veríamos el mensaje "No soy mayor de edad".

Si recuerda el operador condicional explicado en el capítulo anterior, verá que existe una gran similitud con esta estructura de control. Fíjese cómo podría escribirse el ejemplo anterior usando solamente el operador condicional:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var edad = 19;
   var mensaje;
   // Operador condicional
   mensaje = (edad >= 18)? "Soy mayor de edad" : "No soy
   mayor de edad";
   // Mostrar el mensaje
   alert(mensaje);
</SCRIPT>
```

El resultado obtenido es exactamente el mismo. Únicamente necesitamos almacenar primero el texto que devuelve el operador dentro de una variable en vez de mostrarlo directamente.

También es posible encadenar varias sentencias `if - else` si necesitamos evaluar distintas condiciones.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var semaforo = "verde";
   // Sentencias if - else
   if (semaforo == "rojo") {
      alert("Stop");
   } else if (semaforo == "ámbar") {
      alert("Precaución");
   } else if (semaforo == "verde") {
      alert("Adelante");
   }
</SCRIPT>
```

En este *script* se irán comprobando una a una todas las condiciones hasta que se encuentre alguna que concuerde con el valor de la variable `semaforo`. En nuestro caso se cumplirá la última condición haciendo que se muestre el mensaje "Adelante".

En estos casos la sentencia `else` final también es opcional, aunque es muy útil para ejecutar unas instrucciones por defecto, es decir, cuando no se cumple ninguna de las condiciones previas.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var semaforo = "blanco";
   // Sentencias if - else
   if (semaforo == "rojo") {
      alert("Stop");
   } else if (semaforo == "ámbar") {
      alert("Precaución");
   } else if (semaforo == "verde") {
      alert("Adelante");
   } else {
      alert("¡Este color no es de un semáforo!");
   }
</SCRIPT>
```

También sería posible anidar sentencias `if-else` dentro de otras:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var precio = 100;
   // Sentencias if - else anidadas
   if (precio > 50) {
      if (precio < 75) {
         alert("Normal");
      } else {
         alert("Caro");
      }
   } else {
      alert("Barato");
   }
   // Resto del código
</SCRIPT>
```

Ejecutando este código, entraríamos inicialmente dentro del bloque del primer `if`, ya que la condición se cumple (`precio > 50`), descartando por tanto el bloque de la sentencia `else`. Dentro de ese primer `if`, nos encontramos de nuevo con otro `if` que evalúa si el valor de la variable `precio` es menor que

75, cosa que esta vez no se cumple. Por tanto, se ejecutará finalmente el bloque de la sentencia `else` anidada, dando como resultado el mensaje "Caro". El *script* seguiría ejecutando el código que hubiese a continuación del `if-else` principal dado que se ha entrado en un bloque de instrucciones.

4.1.2. Sentencia switch - case

Esta otra sentencia evalúa una expresión que puede tener múltiples valores (opciones) posibles.

La sentencia principal es el `switch`, que va acompañada de la expresión a evaluar. Dentro de ella habrá sub-bloques, identificados por la palabra reservada `case`, para cada uno de los posibles valores que puede tomar la expresión evaluada, y que contendrán las instrucciones a ejecutar si el valor de la expresión coincide con él. Cada uno de estos sub-bloques `case` no se delimita con las llaves que hemos visto hasta ahora, sino que se debe hacer con dos puntos (`:`) al comienzo y la sentencia `break` al final.

Una vez ejecutadas las instrucciones de un `case`, se finaliza la sentencia `switch` ignorando el resto de opciones. Recapitulando, la sintaxis queda así:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia switch-case
   swicth (expresión) {
      case valor1:
         // Bloque de instrucciones para valor1
         break;
      case valor2:
         // Bloque de instrucciones para valor2
         break;
      ...
   }
</SCRIPT>
```

Para aclarar el funcionamiento, veamos esta nueva sentencia mediante el ejemplo del semáforo que utilizamos anteriormente:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var semaforo = "verde";
   // Sentencia switch
   switch (semaforo) {
      case "rojo":
         alert("Stop");
```

```
        break;
      case "ámbar":
         alert("Precaución");
         break;
      case "verde":
         alert("Adelante");
         break;
   }
</SCRIPT>
```

Un poco más claro, ¿verdad? El resultado de ejecutar este código es exactamente el mismo que con las sentencias `if-else`: se mostraría el mensaje "Adelante".

Al igual que ocurre con el `if`, también es posible indicar un conjunto de instrucciones que se ejecuten por defecto cuando la expresión no coincide con ninguna de las opciones. Esta operación se hace añadiendo un nuevo sub-bloque con la palabra reservada `default` que va normalmente en la última posición.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var semaforo = "blanco";
   // Sentencia switch
   switch (semaforo) {
      case "rojo":
         alert("Stop");
         break;
      case "ámbar":
         alert("Precaución");
         break;
      case "verde":
         alert("Adelante");
         break;
      default:
         alert("¡Este color no es de un semáforo!");
      break;
   }
</SCRIPT>
```

La sentencia `break` esconde un pequeño secreto: en realidad es opcional. Si la omitimos, estamos dejando sin delimitar el bloque `case`, por lo que el *script* se seguiría ejecutando hacia abajo hasta que se llegue al final del `switch` o se encuentre una sentencia `break`, aunque eso implique "colarnos" dentro del bloque de otro `case`. Veamos qué pasa si omitimos los `break` en el ejemplo del semáforo:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var semaforo = "ámbar";
   // Sentencia switch
   switch (semaforo) {
      case "rojo":
         alert("Stop");
      case "ámbar":
         alert("Precaución");
      case "verde":
         alert("Adelante");
   }
</SCRIPT>
```

¿Qué mensaje aparece en su pántalla? En principio esperamos que sólo aparezca el mensaje "Precaución" correspondiente al valor "ámbar" pero, sin embargo, también aparece también el mensaje "Adelante" a pesar de que el valor de la variable no es "verde".

Por tanto, debemos tener un poco de cuidado delimitando correctamente los bloques `case` si no queremos obtener un resultado que no es el esperado. No obstante, la omisión de la sentencia `break` puede tener sentido si en su código necesita ejecutar las mismas instrucciones para un conjunto de valores. Veamos un caso práctico:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var mes = "mayo";
   // Sentencia switch
   switch (mes) {
      case "enero":
      case "marzo":
      case "mayo":
      case "julio":
      case "agosto":
      case "octubre":
      case "diciembre":
         alert(mes + " tiene 31 días");
         break;
      case "abril":
      case "junio":
      case "septiembre":
      case "noviembre":
         alert(mes + " tiene 30 días");
         break;
      case "febrero":
         alert(mes + " tiene 28 o 29 días");
         break;
   }
</SCRIPT>
```

Fíjese que los `case` que no llevan bloque de instrucciones dan la sensación de estar agrupados, consiguiendo que al entrar en uno de ellos se llegue hasta la instrucción que hay antes del `break` correspondiente. Es como si dijéramos: si la variable `mes` es igual a "abril", "junio", "septiembre" o "noviembre", entonces el mensaje es "tiene 30 días". Por consiguiente, el mensaje que obtendremos en el ejemplo será "mayo tiene 31 días".

4.2. Estructuras de bucle

Estas estructuras ejecutan un conjunto de instrucciones repetidamente mientras se cumpla una condición. En el momento que deja de cumplirse, el bucle se da por terminado. A cada repetición de un bucle se le denomina iteración.

> **Advertencia**: *Tenga especial cuidado en asegurarse de que en algún momento la condición dejará de cumplirse o provocará un bucle infinito (más conocido como "dejar el ordenador tostado").*

4.2.1. Sentencia for

Este tipo de bucle ejecuta las instrucciones de su bloque recorriendo un rango de valores de forma secuencial hasta que la condición deja de cumplirse (valor falso). Su sintaxis es la siguiente:

```
<SCRIPT TYPE="text/javascript">
      // Sentencia for
      for (expresión inicial; condición; expresión
      actualización) {
            // Mi bloque de instrucciones
      }
</SCRIPT>
```

- Expresión inicial: Se utiliza para inicializar una o varias variables que sirvan a modo de contadores, aunque puede contener expresiones más complejas. Representa el primer valor de todo el rango que se va a utilizar dentro del bucle.
- Condición: Evalúa en cada iteración si el valor de una variable (normalmente la usada como contador) cumple la expresión especificada. Si no lo hace (valor falso), significa que el bucle finaliza.

- Expresión actualización: Se utiliza para actualizar el valor del contador declarado en la expresión inicial, con el fin de hacer falsa la condición en algún momento. Se puede usar cualquier tipo de expresión, aunque lo más común es hacer uso del operador de incremento (++).

Veamos un rápido ejemplo para aclarar el funcionamiento de esta estructura.

```
<SCRIPT TYPE="text/javascript">
   // Sentencia for
   for (var x=1; x<10; x++) {
      alert(x);
   }
</SCRIPT>
```

La mecánica que sigue la ejecución de un bucle `for` es la siguiente:

1. La variable `x` se declara e inicializa con valor 1 (expresión inicial). Esto será nuestro contador.
2. Se comprueba si se cumple la condición (el valor de `x` es menor que 10). Si el resultado es `false` el bucle se da por terminado, en otro caso se entra en el bloque de instrucciones.
3. Se ejecuta el bloque de instrucciones, mostrando en nuestro ejemplo el valor de la variable `x` cada vez (desde 1 hasta 9).
4. Se incrementa el valor de la variable `x` en una unidad (expresión actualización) y se vuelve al paso 2.

También es posible utilizar dos o más contadores dentro de un bucle `for` mediante el operador coma (,). Veamos un *script* que nos muestra la relación, un tanto trivial, entre dos números:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia for
   for (var x=1, y=2; x<=10; x++, y++) {
      alert(x + " es menor que " + y);
   }
</SCRIPT>
```

Aquí hemos inicializado las dos variables con un valor distinto en la expresión inicial e incrementado sus respectivos valores en la expresión de actualización. Para recorrer los valores del 1 al 9, sólo hemos comprobado la variable `x` dentro de la condición.

Hemos mencionado que lo más común es una usar una de las "variables contador" para establecer cuándo finaliza el bucle, pero esto no tiene por qué ser necesariamente así:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var seguir = true;
   // Sentencia for
   for (var x=1; seguir; x++) {
      alert(x);
      if (x >= 9) {
         seguir = false;
      }
   }
</SCRIPT>
```

El resultado de este *script* es el mismo que vimos al principio de este apartado, pero esta vez estamos comprobando cuándo finaliza el bucle en base al valor de la variable `seguir`, que cambia en el `if` anidado dentro del bucle `for`. Si la variable es `false`, el bucle termina, y en otro caso se continúa ejecutando. La condición de parada también se podría haber escrito como: `seguir == true` o `seguir != false`.

> **Nota**: *Si la condición es falsa desde el principio el bloque de instrucciones no se ejecutará nunca. Por ejemplo:* `for (var x=10; x<5; x++)`.

Tal como comentamos acerca de la expresión de actualización, se puede utilizar cualquier expresión siempre y cuando haga cumplir la condición de salida en algún momento. El siguiente ejemplo es una variante del primero que vimos (mostrar los números del 1 al 9). En lugar de incrementar en una unidad el contador, vamos a hacer que se haga en 2 unidades, de modo que mostrará los números impares entre el 1 y el 9:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia for
   for (var x=1; x<10; x=x+2) {
      alert(x);
   }
</SCRIPT>
```

Como podrá ver, hemos usado una expresión de suma para definir la forma de actualizar la variable que hace de contador (`x`). También lo podríamos haber escrito con el operador abreviado `x+=2`.

4.2.2. Sentencia do - while

Esta sentencia ejecuta una serie de instrucciones, al menos una vez, hasta que la condición que se evalúa toma valor falso.

La sintaxis es la siguiente:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia do-while
   do {
      // Mi bloque de instrucciones
   } while (condición)
</SCRIPT>
```

Como puede fijarse, no hay una condición para ejecutar por primera vez el bloque de instrucciones, así que debe tener esto en cuenta a la hora de decidir cuándo usar este tipo de bucles para no encontrarse con resultados inesperados. Vamos con un ejemplo de toma de contacto:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var x = 1;
   // Sentencia do-while
   do {
      alert(x);
      x++;
   } while (x < 10)
</SCRIPT>
```

La ejecución de este tipo de estructura sigue estos pasos:

1. Se ejecuta el bloque de instrucciones, mostrando el valor de la variable `x` (desde 1 hasta 9) y realizando un post-incremento de la misma para no crear un bucle infinito.
2. Se comprueba la condición para determinar si debe ejecutarse una nueva iteración del bucle. En caso de no cumplirse (`x` vale 10) se da por finalizado y en otro caso se vuelve al paso 1.

> **Advertencia:** *La expresión inicial (`var x = 1`) debe escribirse antes de comenzar el bucle y la expresión de actualización (`x++`) debe ponerse dentro del bloque de instrucciones.*

Para acortar el código una pizca, se podría realizar el post-incremento dentro de la función `alert`, aunque esto puede restar claridad a nuestro código.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var x = 1;
   // Sentencia do-while
   do {
      alert(x++);
   } while (x < 10)
</SCRIPT>
```

4.2.3. Sentencia while

Esta última sentencia de bucle ejecuta un bloque de instrucciones repetidamente mientras la condición sea verdadera. A diferencia de la estructura `do-while`, aquí se comprueba la condición antes de realizar la primera iteración, por lo que no se ejecuta el bloque una primera vez antes de ello. Su sintaxis es esta:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia while
   while (condición) {
      // Mi bloque de instrucciones
   }
</SCRIPT>
```

Su funcionamiento es muy similar al `do-while`, así que lo explicaremos a través de un ejemplo:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var x = 1;
   // Sentencia while
   while (x < 10) {
      alert(x);
      x++;
   }
</SCRIPT>
```

1. Se comprueba la condición para saber si debe ejecutarse el bloque de instrucciones. Cuando no se cumpla (el valor de x es 10) se dará por finalizado el bucle.
2. El bloque de instrucciones es ejecutado, mostrando el valor de la variable x (desde 1 hasta 9) y realizando un post-incremento de la misma. Seguidamente se vuelve al paso 1.

También se deben escribir las expresiones de inicialización y actualización en las mismas zonas que vimos con el `do-while` para que el bucle funcione correctamente.

4.2.4. Sentencias break y continue

Estas dos sentencias tendrán utilidad únicamente dentro de los bucles y el `switch`. Veamos qué hace cada una de ellas.

Break

La sentencia `break` sirve para finalizar prematuramente un bucle o, como ya hemos visto, para delimitar un bloque `case` dentro de una sentencia `switch`. Observemos ahora cómo influye su uso dentro de un bucle:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var x = 1;
   // Sentencia while
   while (x < 10) {
      if (x == 5) {
         break;
      }
      x++;
   }
   alert("La mano tiene " + x + " dedos.");
</SCRIPT>
```

En este *script* se interrumpe la repetición del bucle `while` cuando la variable x toma el valor 5, de manera que el resto de instrucciones dentro del bloque son ignoradas y se continúa la ejecución por la instrucción que hay a continuación del bucle, en este caso un `alert`.

Continue

Esta otra sentencia nos permitirá saltar una iteración de un bucle, ignorando además todas las instrucciones que hubieran por debajo. Tras ese salto, la expresión de actualización es ejecutada y la condición de salida es comprobada antes de hacer la iteración correspondiente. Vamos a comprender esto con un ejemplo:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia for
   for (var x=1; x < 10; x++) {
      if (x == 4) {
         continue;
      }
      alert("La plaza de parking " + x + " está libre.");
   }
</SCRIPT>
```

El resultado que conseguimos es que se muestren los distintos valores de la variable x desde el 1 hasta el 9 excepto el 4, ya que es el que provoca el salto en el bucle.

Si el código fuese este:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia for
   for (var x=1; x < 10; x++) {
      if (x == 9) {
         continue;
      }
      alert("La plaza de parking " + x + " está libre.");
   }
</SCRIPT>
```

Se nos mostrarían mensajes diciendo que las plazas de la 1 a la 8 están libres, saltaría a la 9 debido al `continue` y acto seguido saldría del bucle ya que tras actualizar `x` su valor no cumpliría la condición.

4.3. Estructuras de manipulación de objetos

En este último apartado veremos que JavaScript también nos ofrece algunas estructuras de control para acceder a las propiedades y métodos de un objeto de una forma más cómoda.

Como se le hará difícil comprender su funcionamiento sin conocer cómo definir un objeto en JavaScript, aplazaremos la explicación de estas estructuras hasta el capítulo correspondiente. De todos modos, para que le vayan resultando familiares, se trata de las estructuras `for-in` y `with`.

5 Funciones

Es muy común a la hora de programar que nos surja la necesidad de ejecutar un conjunto de acciones de forma habitual o simplemente nos convenga que se ejecuten de forma independiente para dar mayor claridad al código. Pues bien, la solución a nuestro problema son las funciones, que se encargan de agrupar una serie de operaciones dentro de un mismo bloque para ejecutarlas cuando y cuantas veces queramos. De esta forma, podemos tener por ejemplo una función que se encargue siempre de sumar dos números, con lo que nos evitaremos repetir esa operación de suma en distintas partes del código.

Después de leer este apartado sabrá cómo crear sus propias funciones, pero debe saber que también existen muchas funciones que ya están definidas en JavaScript y le serán de mucha utilidad. A lo largo de los capítulos iremos viendo muchas de ellas.

5.1. Declaración de funciones

JavaScript tiene la palabra reservada `function` para poder identificar las funciones que creemos. Veamos mejor un sencillo ejemplo para explicar la sintaxis que debemos usar:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function sumar(){
      var miSuma = 9 + 2;
   }
</SCRIPT>
```

Primero tenemos la palabra `function` (indispensable), después indicamos el nombre que queremos que tenga la función (`sumar`) seguida obligatoriamente de los paréntesis y, final-

mente, escribimos las acciones que queremos que realice nuestra función (realizar una suma simple) encerradas entre llaves ({ }) formando un bloque. Dentro de este bloque podemos declarar variables y realizar las operaciones que nos venga en gana.

Para utilizar o llamar a una función que hayamos definido es suficiente con escribir su nombre seguido de los paréntesis. Con esto conseguimos que se ejecuten las acciones que contenga:

```
<SCRIPT TYPE="text/javascript">
   // Llamada a la función sumar
   sumar();
</SCRIPT>
```

En cuanto a la nomenclatura de las funciones, se siguen casi las mismas reglas y restricciones que en el caso de las variables:

1. Nombre en minúsculas, excepto la primera letra de cada palabra siguiente a la primera. Esto es una recomendación que le hago, no es en absoluto obligatorio.
2. Se pueden utilizar caracteres alfanuméricos y algunos símbolos, como el guión bajo o subrayado (_).
3. El nombre no puede coincidir con el de una palabra reservada, como `function`, ni con el de otro elemento que hayamos definido.

Aunque pueda parecer que variables y funciones no se van a diferenciar a simple vista, las funciones se pueden identificar rápidamente por el uso de los paréntesis. También podemos hacer aún mayor esta diferencia si al declarar funciones utilizamos nombres que representen acciones: `sumar` mejor que `suma` o `sumarTresNumeros` en vez de `sumaTresNumeros`.

> **Nota**: *Una vez más, tenga especial cuidado con el uso de mayúsculas y minúsculas, pues las llamadas a funciones como `Sumar` o `SUMAR` no son lo mismo.*

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función con nombre corto
   function sumar(){
      // Mis acciones
   }
   // Declaración de función con nombre largo
   function sumarTresNumeros(){
      // Mis acciones
   }
</SCRIPT>
```


Las funciones pueden ser declaradas en cualquier parte del *script*, pero si tenemos varias etiquetas `<SCRIPT>` en nuestra página, deben estar en una anterior a la parte donde se hagan las llamadas. En los siguientes ejemplos veremos formas posibles de definir y llamar correctamente una función:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función en la misma etiqueta
   function sumar(){
      // Mis acciones
   }
   // Llamada a la función de la misma etiqueta
   sumar();
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Llamada a la función de la misma etiqueta
   sumar();
   // Declaración de función en la misma etiqueta
   function sumar(){
      // Mis acciones
   }
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de función en otra etiqueta
   function sumar(){
      // Mis acciones
   }
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Llamada a la función de otra etiqueta anterior
   sumar();
</SCRIPT>
```

Ahora veremos un ejemplo en el que la definición y la llamada se hacen de forma incorrecta, haciendo que se llame a una función que para JavaScript no está definida:

```
<SCRIPT TYPE="text/javascript">
   // Llamada a la función de otra etiqueta posterior (error)
   sumar();
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de función en otra etiqueta
   function sumar(){
      // Mis acciones
   }
</SCRIPT>
```

Dentro del bloque de una función (lo encerrado entre llaves) se pueden definir y usar variables de todo tipo e incluso llamar a otras funciones.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function sumar(){
      // Mis acciones
   }
   // Uso de variables y otra función ya definida
   function miFuncion(){
      var numero1 = 5;
      var nombre = "Pepe";
      var otroNumero = numero1;
      sumar();
   }
</SCRIPT>
```

Truco: *Para evitar errores en las llamadas a funciones es recomendable declarar todas al principio de nuestra página dentro de <HEAD>, bien una a una o todas de golpe a través de un fichero externo (.js).*

5.2. Parámetros

Ahora que ya sabemos definir nuestras propias funciones, vamos a dar un paso más. Como habrá podido observar, la utilidad de nuestras funciones hasta ahora no es que sea muy alta ya que las acciones que se realizan dentro usan variables con valores fijos que no podemos variar al hacer la llamada (nos puede interesar sumar números distintos en diferentes llamadas a la función `sumar`). Gracias a los parámetros le podremos indicar a una función los valores con los que queremos operar. A esto se le conoce como pasar o mandar parámetros.

5.2.1. Definición de parámetros

Para definir los parámetros de una función basta con poner el nombre que le identificará, dentro de los paréntesis que usamos al definir una función. Los parámetros pueden interpretarse como variables dentro de las funciones, pero no es necesario declararlas previamente ya que el hecho de que sea un parámetro implica su declaración.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función con un parámetro
   function sumar(numero){
      var miSuma = numero + 2;
   }
</SCRIPT>
```


Para pasar un parámetro a una función debemos colocar un valor entre los paréntesis que se usa en las llamadas a funciones. Al hacer esto, el parámetro se inicializa con dicho valor. Vamos con un ejemplo de todo esto:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var cinco = 5;
   // Declaración de función con un parámetro
   function sumar(numero){
      var miSuma = numero + 2;
   }
   // Llamada a la función con un dato
   sumar(3); // El resultado sería 5 (3 + 2)
   // Llamada a la función con una variable
   sumar(cinco); // El resultado sería 7 (5 + 2)
</SCRIPT>
```

Advertencia: *El nombre de nuestros parámetros no puede coincidir con el de ninguna de las variables declaradas dentro de la función.*

Hay que resaltar que los parámetros de las funciones se pasan por valor, es decir, aunque usemos un parámetro dentro de la función para almacenar otro resultado, el valor de la variable original usada en la llamada no varía. Procedamos a ver esto con un ejemplo:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var cinco = 5;
   // Declaración de función con un parámetro
   function sumar(numero){
      // Modificamos el parámetro
      numero = 4;
      // El resultado de la suma será 6
      var miSuma = numero + 2;
   }
   // Llamada a la función con una variable
   sumar(cinco); // El resultado sería 6 (4 + 2)
   // Otra llamada (cinco sigue valiendo 5)
   sumar(cinco); // El resultado sería 6 (4 + 2)
</SCRIPT>
```

Ahora ya podemos decirle a nuestra función qué número queremos sumar con el 2. Esto está bien pero, ¿y si queremos decirle a la función que sume nuestro número con otro que no sea el 2? La solución, a continuación.

5.2.2. Múltiples parámetros

Por suerte, JavaScript ha pensado en todo y nos permite definir en una función tantos parámetros como necesitemos simplemente con separar cada nombre con una coma (,). Asimismo, la llamada y paso de parámetros a la función debe hacerse de forma análoga. Veamos un ejemplo, extensión del anterior:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var cinco = 5, diez = 10;
   // Declaración de constante
   var MAXIMO = 25;
   // Declaración de función con dos parámetros
   function sumar(numero1, numero2){
      var miSuma = numero1 + numero2;
   }
   // Declaración de función con tres parámetros
   function sumarTresNumeros(num1, num2, num3){
      var miSuma = num1 + num2 + num3;
   }
   // Llamadas a la función con dos valores
   sumar(3, 6); // El resultado sería 9
   sumar(cinco, diez); // El resultado sería 15
   // Llamada a la función con tres valores
   sumarTresNumeros(MAXIMO, 3, diez); // El resultado
                                      // sería 38
   sumarTresNumeros(0, diez, cinco); // El resultado
                                     // sería 15
</SCRIPT>
```

La llamada a una función con parámetros debe hacerse pasando tantos datos como parámetros tenga.

Si se diese el caso de omitir alguno de los parámetros en la llamada a una función, éstos se inicializarán con un valor indefinido (`undefined`) y el resultado de las acciones será impredecible.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función con dos parámetros
   function sumar(numero1, numero2){
      var miSuma = numero1 + numero2;
   }
   // Llamada a la función con un solo valor (error)
   sumar(7);
</SCRIPT>
```

En este *script*, el parámetro `numero2` tomará el valor `undefined`, haciendo que el resultado de la suma (`7 + undefined`) sea `NaN`.

Debo señalar que como las variables o parámetros en JavaScript no tienen un tipo definido, es posible pasar un dato de un tipo que no se espera dentro de la función. Por ejemplo, podemos pasar una cadena como parámetro de la función `sumar` sin problemas aunque el resultado no sería el deseado.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función con dos parámetros
   function sumar(numero1, numero2){
      var miSuma = numero1 + numero2;
   }
   // Llamada a la función con tipo no esperado
   sumar("salu", 2); // El resultado sería la cadena
                     // "salu2"
</SCRIPT>
```

5.2.3. Parámetros obligatorios y opcionales

Para terminar con este punto, me gustaría ponerle al tanto de que es posible hacer funcionar los parámetros como obligatorios u opcionales. Esto es, si tenemos declarada una función con dos parámetros podemos prepararla para que no ejecute su bloque si no se le han pasado todos los parámetros (los hacemos obligatorios) o que lo ejecute tanto con dos parámetros, uno o incluso ninguno (los hacemos opcionales) pudiendo hasta asignarles valores por defecto.

Desgraciadamente, y siento dejarle con la miel en los labios, a estas alturas del libro no es posible explicarle esta funcionalidad ya que se requiere el manejo de objetos (concretamente el objeto `arguments`), los cuales no hemos aprendido a manejar, pero prometo hacerlo más adelante. De momento, puedo dejarle unos ejemplos de cómo se haría este control, omitiendo el código correspondiente a los objetos:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función con dos parámetros
   // obligatorios
   function sumar(numero1, numero2){
      // Hacemos obligatorios los parámetros
      if (no hemos recibido numero 1 &&
          no hemos recibido numero2){
            // No hago nada
      }else{
            var miSuma = numero1 + numero2;
      }
   }
```

```
  // Llamadas a la función
  sumar(2, 5); // El resultado sería 7
  sumar(2); // La función no haría nada
  sumar(); // La función no haría nada
</SCRIPT>
<SCRIPT TYPE="text/javascript">
  // Declaración de función con un parámetro opcional
  function sumar(numero1, numero2){
     // Asignamos un valor por defecto
     if (no hemos recibido numero2){
        numero2 = 7;
     }
     var miSuma = numero1 + numero2;
  }
  // Llamadas a la función
  sumar(2, 5); // El resultado sería 7
  sumar(2); // El resultado sería 9 (usando valor por
            // defecto)
  sumar(); // El resultado sería NaN (numero1 es
           // necesario)
</SCRIPT>
<SCRIPT TYPE="text/javascript">
  // Declaración de función con dos parámetros
  // opcionales
  function sumar(numero1, numero2){
     // Asignamos varios valores por defecto
     if (no hemos recibido numero1){
        numero1= 7;
     }
     if (no hemos recibido numero2){
        numero2= 1;
     }
     var miSuma = numero1 + numero2;
  }
  // Llamadas a la función
  sumar(2, 5); // El resultado sería 7 (ningún valor
               // por defecto)
  sumar(2); // El resultado sería 3 (usando valor por
            // defecto)
  sumar(); // El resultado sería 8 (usando valores por
           // defecto)
</SCRIPT>
```

Cuando los parámetros son opcionales, sólo pueden serlo aquellos situados a la derecha del último parámetro que sea necesario para realizar las acciones del bloque. Me explico. Si tenemos la siguiente función:

```
<SCRIPT TYPE="text/javascript">
  // Declaración de función con un parámetro opcional
  function sumar(numero1, numero2){
     // Asignamos un valor por defecto
     if (no hemos recibido numero2){
        numero2 = 7;
     }
     var miSuma = numero1 + numero2;
  }
</SCRIPT>
```

No podremos hacer que el parámetro `numero1` sea opcional y `numero2` necesario, ya que la llamada `sumar(,3)` provocaría un error. En cambio, `numero1` puede ser necesario y `numero2` opcional puesto que la llamada `sumar(3)` es completamente válida.

5.3. Valores de retorno

Las funciones nos proporcionan una funcionalidad más: la posibilidad de devolvernos un valor como resultado de las acciones ejecutadas en su bloque. Para ello se utiliza la palabra reservada `return` junto con el valor que queremos devolver.

Para recoger el valor devuelto por la función hay que escribir una asignación haciendo que el operando de la derecha sea una llamada a una función.

Completemos un ejemplo anterior de la función `sumar`, haciendo que nos devuelva el resultado de la suma y recogiendo su valor en una variable:

```
<SCRIPT TYPE="text/javascript">
  // Declaración de variables
  var cinco = 5;
  var resultado;
  // Declaración de función
  function sumar(numero1, numero2){
     return numero1 + numero2;
  }
  // Recogida de valores
  resultado = sumar(3, 6);
  alert(resultado); // Mostrará 9
  resultado = sumar(cinco, 10);
  alert(resultado); // Mostrará 15
</SCRIPT>
```

En el momento en que se devuelve el valor la función terminará su ejecución (se sale de la función), ignorando todo el código que hubiese por debajo.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var cinco = 5;
   var resultado;
   // Declaración de función
   function sumar(numero1, numero2){
      return numero1 + numero2;
      alert("¡Estoy en la función!");
   }
   // Llamadas a la función
   resultado = sumar(3, 6);
   alert(resultado); // Mostrará 9
</SCRIPT>
```

En este ejemplo únicamente veremos el mensaje con el valor de la variable `resultado` dado que el otro mensaje se encuentra después del `return`, así que no se ejecutará por haber salido de la función.

También es posible utilizar la sentencia `return` sin ningún valor haciendo que la función termine sin devolver ningún resultado. Esto resulta útil cuando queremos evitar ejecutar parte del bloque de instrucciones, por ejemplo, en función del valor de los parámetros recibidos.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function acelerar(velocidad){
      if (velocidad <= 120) {
         return;
      }
      alert(velocidad + "km/h: Ojo con el acelerador!");
   }
   // Llamada a la función
   acelerar(100);
   acelerar(120);
   acelerar(130);
</SCRIPT>
```

Si analizamos detenidamente la función `acelerar` del ejemplo, vemos que si la velocidad pasada como parámetro es menor o igual a 120 se ejecuta el `return` y, por tanto, la función finaliza sin realizar acción alguna. Sin embargo, en cuanto la velocidad es mayor a 120 se nos muestra un mensaje.

Al utilizar el return sin un valor para devolver dentro de una función, ¿qué pasaría si intentásemos recoger el resultado de la llamada de esa función? No hace falta abrir un debate, lo veremos en un ejemplo:


```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var resultado;
   // Declaración de función
   function noDevolverValor(){
      return;
   }
   // Llamada a la función
   resultado = noDevolverValor();
   alert(resultado);
</SCRIPT>
```

Si ejecutamos este *script* veremos que `resultado` tiene un valor `undefined` puesto que no se le ha asignado ningún valor con la llamada a la función `noDevolverValor`.

5.3.1. Múltiples valores de retorno

A veces, dentro del bloque de una función, nos puede interesar tener más de un valor posible de retorno, pero únicamente será posible devolver uno de ellos ya que, recordemos, la sentencia `return` nos finaliza la función. Veamos el caso de querer devolver el mayor de dos números.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var resultado;
   // Declaración de función
   function detectarMayor(numero1, numero2){
      if (numero1 > numero2){
         return numero1;
      }else{
         return numero2;
      }
   }
   // Llamadas a la función
   resultado = detectarMayor(3, 6); // resultado vale 6
   alert(resultado);
   resultado = detectarMayor(10, 8); // resultado vale 10
   alert(resultado);
</SCRIPT>
```

En este *script* la variable `resultado` almacena el mayor de los dos números que es devuelto por la función después de realizar la comparación entre ambos.

Por último, debe saber que no es obligatorio recoger el valor devuelto por una función, pudiendo hacer que sirva directamente como valor de un parámetro o simplemente se puede dejar que se pierda.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var resultado;
   // Declaración de función
   function sumar(numero1, numero2){
      return numero1 + numero2;
   }
   // Recogida del valor de retorno en variable
   resultado = suma(3, 5); // resultado vale 8
   alert(resultado);
   // Recogida del valor de retorno como parámetro
   // directo sin almacenarlo
   alert(sumar(3, 5));
   // Pérdida del valor de retorno
   sumar(3, 5);
</SCRIPT>
```

5.4. Funciones predefinidas

JavaScript incluye una serie de funciones ya definidas que pueden ser usadas con cualquier variable, valor directo u objeto. Gracias a muchas de estas funciones podremos realizar una conversión explícita de tipos de datos.

5.4.1. Función Number

Esta función permite expresar cualquier valor como un número. En caso de que la conversión no sea posible, se devuelve el valor `NaN` (Not a Number).

Tabla 5.1. Ejemplos con la función Number.

Valor	*Conversión*
5	5
-3	-3
1.95	1.95
"1.95"	1.95
"1,95"	NaN
true	1
false	0
"hola"	NaN

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(Number("1.95")); // 1.95
   alert(Number(true)); // 1
   alert(Number("hola")); // NaN
</SCRIPT>
```

5.4.2. Función String

Esta función permite convertir cualquier valor en una cadena. Aquí siempre es posible la conversión por lo que el valor devuelto es una cadena en todos los casos.

Tabla 5.2. Ejemplos con la función String.

Valor	*Conversión*
5	"5"
-3	"-3"
1.95	"1.95"
"1,95"	"1,95"
true	"true"
false	"false"
"hola"	"hola"
NaN	"NaN"

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(String(1.95)); // "1.95"
   alert(String(true)); // "true"
   alert(String("hola")); // "hola"
</SCRIPT>
```

5.4.3. Función isNaN

Esta función comprueba si un valor no es un número, por lo que devolverá `false` en caso de ser un número y `true` en cualquier otro caso. Antes de comprobar el valor, se aplica automáticamente la conversión implícita de tipos.

Tabla 5.3. Ejemplos con la función isNaN.

Valor	*Resultado*
5	false
-3	false
1.95	false
"1.95"	false
"1,95"	true
true	false
false	false
"hola"	true
NaN	true

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(isNaN(1.95)); // false
   alert(isNaN(true)); // false
   alert(isNaN("hola")); // true
</SCRIPT>
```

5.4.4. Función isFinite

Esta función comprueba si un valor es un número finito. Si el valor es `NaN`, `Infinity` o `-Infinity` se devolverá `false` y en el resto de los casos se devolverá `true`. También se realiza la conversión implícita de tipos antes de comprobar el valor.

Tabla 5.4. Ejemplos con la función isFinite.

Valor	*Resultado*
5	true
1.95	true
"1.95"	true
"1,95"	false (NaN tras conversión implícita)
true	true
false	true
"hola"	false (NaN tras conversión implícita)
NaN	false
Infinity	false
-Infinity	false

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(isFinite(1.95)); // true
   alert(isFinite(true)); // true
   alert(isFinite("hola")); // false
</SCRIPT>
```

5.4.5. Función parseInt

Esta función permite transformar cualquier cadena en un número entero, siguiendo estas reglas:

1. Si la cadena contiene varios números separados, sólo se convierte el primero.
2. Los espacios al principio y final de la cadena son omitidós, pero se tienen en cuenta los espacios intermedios, que actuarán como separadores de números.
3. Si el primer carácter de la cadena o la cadena completa no se puede convertir, se devuelve `NaN`.

> **Nota:** *Si intenta convertir un número con decimales, éstos se omitirán ya que la función sólo convierte el valor a un entero.*

Tabla 5.5. Ejemplos con la función parseInt.

Valor	*Conversión*
"5"	5
"-3"	-3
"1.95"	1
"1,95"	NaN (no convertible)
"true"	NaN (no convertible)
"false"	NaN (no convertible)
"hola"	NaN (no convertible)
"1 2 3"	1
" 1.5 2 3"	1
"1 b 3"	1
"a 2 3"	NaN (primer carácter no convertible)

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(parseInt("1.95")); // 1
   alert(parseInt("true")); // NaN
   alert(parseInt("1 b 3")); // 1
</SCRIPT>
```

Esta función cuenta además con un segundo parámetro opcional para indicar la base (entre 2 y 36) en la que se encuentra el valor de la cadena a convertir. El resultado de la conversión siempre estará expresado en base 10.

Si no se especifica este parámetro se siguen los siguientes criterios:

1. Si la cadena empieza por "0x", la base es 16.
2. Si la cadena empieza por "0", la base es 8 (en desuso).
3. Para el resto de valores, la base es 10.

Tabla 5.6. Ejemplos con la función parseInt especificando una base.

Valor	*Base*	*Conversión en base 10*
"9"	10	9 (decimal)
"11"	8	9 (octal)
"1A"	16	26 (hexadecimal)
"011"	2	3 (binario)
"9"	-	9 (se asume base 10)
"011"	-	9 (se asume base 8)
"0x1A"	-	26 (se asume base 16)
"0x1A"	10	NaN (no convertible en base 10)
"0xHola"	16	NaN (no convertible)

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(parseInt("9", 10)); // 9
   alert(parseInt("1A", 16)); // 26
   alert(parseInt("9")); // 9
   alert(parseInt("0x1A")); // 26
   alert(parseInt("0xHola", 16)); // NaN
</SCRIPT>
```


5.4.6. Función parseFloat

Permite convertir cualquier cadena en un número real, mediante las mismas reglas que sigue `parseInt` con un parámetro (no hay segundo parámetro opcional en esta función).

Nota: *El separador decimal en JavaScript es el punto (.).*

Tabla 5.7. Ejemplos con la función parseFloat.

Valor	*Conversión*
"5"	5
"-3"	-3
"1.95"	1.95
"1,95"	NaN (no convertible)
"true"	NaN (no convertible)
"false"	NaN (no convertible)
"hola"	NaN (no convertible)
"1 2.5 3"	1
" 1 2 3"	1
"1.5 b 3"	1.5
"a 2 3"	NaN (primer carácter no convertible)

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(parseFloat("1.95")); // 1.95
   alert(parseFloat("true")); // NaN
   alert(parseFloat("1.5 b 3")); // 1.5
</SCRIPT>
```

5.4.7. Función escape

Esta función se encarga de codificar los caracteres especiales (espacio, acentos, signos de puntuación...) de una cadena, devolviéndola en formato UNICODE (`http://unicode.org`). Cada carácter codificado irá con la forma `%xx`, siendo `xx` su equivalente hexadecimal. Esta función es útil para que una cadena pueda ser leída sin problemas, especialmente cuando se trata de transacciones HTTP (envíos de formularios, cookies...) donde el espacio en blanco, por ejemplo, no puede transmitirse sin codificar ya que provocaría un error.

Tabla 5.8. Ejemplos con la función escape.

Valor	*Codificación*
"dos palabras"	"dos**%20**palabras"
"¿hola?"	"**%BF**hola**%3F**"
"¡Mi código!"	"**%A1**Mi**%20**c**%F3**digo**%21**"

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(escape("dos palabras")); // "dos%20palabras"
   alert(escape("¿hola?")); // "%BFhola%3F"
   alert(escape("¡Mi código!")); // "%A1Mi%20c%F3digo%21"
</SCRIPT>
```

Sí, es difícil descifrar a simple vista qué es lo que habíamos escrito antes de codificar algunos de los caracteres. Pero mientras JavaScript lo entienda no tiene por qué sufrir.

La función `escape` no codifica los siguientes caracteres especiales, por lo que los deja tal cual están:

1. Asterisco (*)
2. Arroba (@)
3. Signo de resta (-)
4. Guión bajo o subrayado (_)
5. Signo de suma (+)
6. Punto (.)
7. Barra o *slash* (/)

Dado que los caracteres especiales son muy usados en el paso de parámetros entre páginas Web (formularios, por ejemplo), está recomendado el uso de las funciones `encodeURI` o `encodeURIComponent` en su lugar ya que funcionan de forma muy similar. Además, como `escape` no trabaja apropiadamente con caracteres fuera del código ASCII (`http://www.ascii.cl/es/`) se recomienda usar esas funciones alternativas desde JavaScript 1.5 en adelante.

5.4.8. Función unescape

Como puede imaginar, esta función realiza el proceso inverso a la que acabamos de ver: descodifica los caracteres convertidos con la función `escape`, volviéndolos a dejar como estaban.

Tabla 5.9. Ejemplos con la función unescape.

Valor	*Descodificación*
"dos%20palabras"	"dos palabras"
"%BFhola%3F"	"¿hola?"
"%A1Mi%20c%F3digo%21"	"¡Mi código!"

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(unescape("dos%20palabras")); // "dos palabras"
   alert(unescape("%BFhola%3F")); // "¿hola?"
   alert(unescape("%A1Mi%20c%F3digo%21")); // "¡Mi código!"
</SCRIPT>
```

Al igual que ocurre con la función `escape`, está recomendado el uso de las funciones `decodeURI` o `decodeURIComponent`.

5.4.9. Función eval

Esta función, de gran utilidad en muchos casos, interpreta una cadena y la ejecuta como si se tratase de una porción de código JavaScript.

En caso de que haya más de una instrucción en la cadena, separadas por punto y coma (`;`), `eval` ejecuta cada una de ellas por separado y en el orden que aparecen escritas.

Tabla 5.10. Ejemplos con la función eval.

Valor	*Resultado*
"2+2"	4 (devuelve la suma).
"suma(2, 2);"	4 (devuelve la ejecución de la función, que debe existir).
"var num=3; suma(2, num);"	5 (declara la variable y devuelve la ejecución de la función).
"suma(2, num);"	Error (num no está definida).
"suma(2, num); var num=3;"	Error (num se define después).
"hola"	Error (interpreta hola como una variable).

Ejemplos con un *script*:

```
<SCRIPT TYPE="text/javascript">
   // Llamadas
   alert(eval("2+2")); // 4
   alert(eval("var num=3; sumar(2, num);")); // 5
   alert(eval("hola")); // Error
</SCRIPT>
```

> **Nota**: *Use la función `eval` sólo si está manejando cadenas. También es capaz de interpretar código JavaScript, pero estará haciendo un uso indebido. Por ejemplo: `eval(2+2);` o `eval(sumar(2, 2));`*

5.5. Ámbito o alcance de las variables

Se le llama ámbito o alcance de una variable al lugar donde está disponible después de haber sido declarada, cosa que ocurre en todos los lenguajes de programación. En el caso de JavaScript, dado que se declaran las variables dentro de una página Web, su disponibilidad queda reducida a dicha página, no pudiendo acceder a las de otra página.

Dentro de una página hay dos tipos de ámbitos: local y global.

5.5.1. Ámbito local

Si declaramos una variable en el bloque de una función e intentamos usarla fuera de ella, obtendremos un error ya que es como si nunca hubiera sido declarada.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de una función con una variable local
   function validar(numero){
      var vale;
      if (numero < 25){
         vale = true;
      }else{
         vale = false;
      }
      return vale;
   }
   // Llamadas a función
   alert(validar(5)); // true
   alert(validar(30)); // false
```



```
   // Uso de variable local
   alert(vale); // Error, variable no definida
</SCRIPT>
```

En este ejemplo comprobamos cómo el valor de la variable local `vale` es correcto sólo si es utilizado dentro de la propia función, que es la única que tiene acceso a ella, mientras que si intentamos obtener su valor directamente obtendremos un error al no estar definida fuera de la función.

> **Nota**: *Los parámetros de una función son variables locales de la misma.*

Por otro lado, las variables definidas dentro de otro tipo de bloques (estructuras de control) tienen un comportamiento un tanto peculiar puesto que no son locales en el sentido estricto de la palabra. La variable estará definida en todo momento, por lo que es accesible desde fuera del bloque, pero pueden ocurrir dos cosas al intentar leer su valor:

1. Si el bloque es ejecutado (se cumple su condición de entrada), entonces el valor que se asigne a la variable estará igualmente disponible fuera de él.
2. Si no se ejecuta nunca el bloque (la condición no se cumple la primera vez), la variable es accesible pero con un valor `undefined`.

```
<SCRIPT TYPE="text/javascript">
   // Estructura con una variable local
   if (5 < 25){
      var vale = true;
   }
   // Uso de variable local
   alert(vale); // true
</SCRIPT>
```

En este primer ejemplo, la variable `vale` es declarada de forma local dentro del bloque `if`. Como la condición de entrada se cumple (`5 < 25`) entonces su valor es accesible fuera del bloque. Veamos el caso contrario.

```
<SCRIPT TYPE="text/javascript">
   // Estructura con una variable local
   if (30 < 25){
      var vale = true;
   }
   // Uso de variable local
   alert(vale); // undefined
</SCRIPT>
```

En esta otra ocasión, como la condición de entrada no se cumple (`30 < 25`) hará que la variable local `vale` no tome el valor `true`. Entonces, puesto que la variable sí está definida para JavaScript, nos encontramos con un ejemplo con un valor no definido.

Para evitar este tipo de situaciones "extrañas", intentaremos declarar previamente todas las variables que se usen dentro de una estructura de control y, siempre que sea posible, asignarle un valor inicial por defecto.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable con valor inicial
   var vale = false;
   // Uso de variable en estructura
   if (30 < 25){
      vale = true;
   }
   // Uso de variable
   alert(vale); // false
</SCRIPT>
```

5.5.2. Ámbito global

Cuando la variable está definida fuera de un bloque, la hacemos accesible desde cualquier parte del código, incluso dentro de las funciones y estructuras de control.

Veamos un ejemplo con variables tanto locales como globales para que el concepto de ámbito quede más claro:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable global
   var MAXIMO = 25;
   // Declaración de una función con una variable local
   function validar(numero){
      var vale;
      if (numero < MAXIMO){
         vale = true;
      }else{
         vale = false;
      }
      return vale;
   }
   // Uso de variables de distintos ámbitos
   alert(validar(5)); // true
   alert(validar(30)); // false
   alert(vale); // Error, variable no definida
   alert(MAXIMO); // 25
</SCRIPT>
```


De este modo conseguimos lo siguiente:

1. La constante global `MAXIMO` es accesible dentro de la función y la comparación con el parámetro `numero` se hace de forma correcta.
2. El valor de la variable local `vale` de la función `validar` es recogido correctamente en las llamadas a la misma.
3. Al intentar acceder directamente al valor de `vale` fuera de la función `validar`, obtenemos un error ya que es de ámbito local y, por tanto, sólo accesible dentro de la función.
4. El valor de la constante `MAXIMO` también puede ser mostrado si la utilizamos en otra parte del código.

En el apartado anterior comentábamos las situaciones "extrañas" que se pueden crear al declarar una variable local dentro de una estructura de control y la forma de evitarlas:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable con valor inicial
   var vale = false;
   // Uso de variable en estructura
   if (30 < 25){
      vale = true;
   }
   // Uso de variable
   alert(vale); // false
</SCRIPT>
```

Si miramos de nuevo el código y lo juntamos con el concepto de ámbito recién explicado, nos podremos dar cuenta de que lo que hicimos en realidad fue darle un ámbito global a la variable `vale`.

5.5.3. Prioridad de las variables

Si dentro de una función se declara una variable local con el mismo nombre que una variable global, tiene preferencia la local a la hora de ser utilizada dentro de esa función. Veamos algunos ejemplos:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable global
   var numero = 5;
   // Declaración de una función con una variable local
   function darNumero(){
      var numero = 10;
      return numero;
```

```
    }
    // Uso de variables con mismo nombre
    alert(numero + "\n" + darNumero() + "\n" + numero);
</SCRIPT>
```

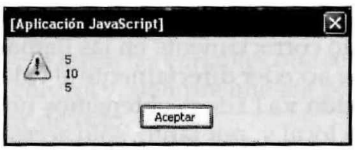


Figura 5.1. Ejemplo 1 de prioridad de variables.

Si observa la figura 5.1 verá que el resultado de nuestro ejemplo cumple la regla de prioridad: la llamada a la función `darNumero` no altera el valor de la variable global (5) ya que usa la local con el mismo nombre.

Aunque ya tenemos la buena costumbre de declarar nuestras variables siempre con la sentencia `var`, tenga presente que el omitirla al declarar la variable local del ejemplo anterior hará que sí se modifique la variable global.

```
<SCRIPT TYPE="text/javascript">
    // Declaración de variable global
    var numero = 5;
    // Declaración de una función sin una variable local
    function darNumero(){
        numero = 10;
        return numero;
    }
    // Uso de variables con mismo nombre
    alert(numero + "\n" + darNumero() + "\n" + numero);
</SCRIPT>
```

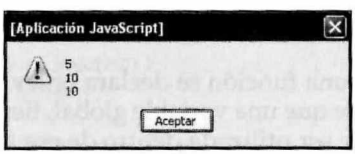


Figura 5.2. Ejemplo 2 de prioridad de variables.

La pequeña diferencia respecto al código anterior (omisión de la sentencia `var`) provoca que no declaremos ninguna variable local en la función, así que se utiliza la variable `numero` de ámbito global a la hora de asignar el valor 10.

6

Programación orientada a objetos

La programación estructurada (la que hemos visto hasta ahora) tiene los datos y las funciones separados sin una relación entre ellos, ya que lo único que se busca es procesar una serie de datos para obtener un resultado. Básicamente se podría decir que sólo se escriben funciones que procesan datos, dado que con esta forma de programar se tiende a pensar primero en la forma de funcionar de los procedimientos o funciones, dejando en un segundo plano las estructuras de datos que se manejan dentro de esas funciones.

La programación orientada a objetos (POO) introduce una nueva estructura (los objetos) que contiene los datos y las funciones que manejan éstos, estando así ambas partes relacionadas. Esto hace más fácil equiparar los objetos a cómo son las cosas en la vida real ya que cualquier cosa o entidad se compone de distintas piezas o partes y forman un todo. Haciendo un símil, se diría que las piezas (objetos) tienen sus propias características (datos y funciones) y al juntarse con otras (escribir un código) crean un elemento con una utilidad determinada (el programa o *script*).

Los datos que contiene un objeto son conocidos como propiedades o atributos, y las funciones como métodos.

Si hacemos memoria del apartado 2.2.4 del capítulo 2, recordaremos que se explicó un ejemplo de objeto con un coche. Las propiedades de ese objeto serían su conjunto de características (color, marca y modelo) y los métodos las distintas funcionalidades que ofrece el coche (arrancar, acelerar y frenar).

Cada objeto puede diferenciarse de otro por su estructura (el objeto coche no se parece al objeto mesa) o simplemente por los valores de sus propiedades. Siguiendo con el ejemplo, no todos los coches son iguales en sus características (color rojo,

azul, verde...) pero todos sí ofrecen las mismas funcionalidades (se entiende que hablamos de funcionalidades básicas, no vamos a adentrarnos en el equipamiento extra). Con esta observación aparece otro concepto importante: las instancias. Todas parten de la misma estructura base pero sus propiedades toman distintos valores.

6.1. Definición de un objeto (constructor)

Cualquier lenguaje de programación actual nos permitirá crear y definir nuestros propios objetos para poder adaptar el código a nuestras necesidades. Además cada lenguaje ofrece una colección más o menos amplia de objetos predefinidos que podremos usar igualmente sin necesidad de quebrarnos la cabeza inventándolos por nuestra cuenta.

Una vez que tenemos claro las propiedades y métodos que van a formar nuestro objeto, definiremos su constructor, cuyo objetivo es crear e inicializar las instancias que hagamos de él. Para conseguir esto disponemos de dos opciones en JavaScript:

1. Utilizar llaves ({}), dentro de las cuales definiremos todas las propiedades y métodos. La sintaxis sería esta:

   ```
   var miObjeto = {propiedades, métodos};
   ```

   En realidad esto no es un constructor en el sentido estricto de la palabra puesto que no podremos crear instancias a partir de esta estructura, sino que estaríamos creando una instancia directamente. De modo que podríamos decir que actúan como un constructor "de un solo uso".
2. Utilizar la sentencia `function`, definiendo dentro de ella las propiedades y métodos. Se escribiría de la siguiente manera:

   ```
   function MiObjeto() {
      // Definición de propiedades
      // Definición de métodos
   }
   ```

Como nomenclatura a seguir, le recomiendo que ponga en mayúsculas la primera letra de cada palabra, incluida la primera. De este modo será más fácil diferenciarlos de las funciones. Fíjese que se ha seguido la nomenclatura de variables

en el ejemplo del constructor mediante llaves, ya que realmente se crea una instancia, mientras que `function` nos deja una estructura lista para crear instancias.

En las secciones que siguen veremos cómo usar cada tipo de constructor.

6.2. Trabajar con objetos

Antes de meternos más en faena con los objetos, voy a presentarle los distintos operadores y sentencias que tiene JavaScript para poder acceder a los distintos elementos que constituyen los objetos.

- El punto (`.`): sirve para obtener el valor de una propiedad o ejecutar un método del objeto que le precede. Se utiliza como sigue:

  ```
  var color = miCoche.color;
  miCoche.arrancar();
  ```

- Los corchetes (`[]`): sirven para lo mismo que el punto, pero para acceder a una propiedad o método debemos escribirlo como un *string*.

  ```
  var color = miCoche["color"];
  miCoche["arrancar"]();
  ```

- `new`: este operador nos permite crear una instancia del objeto que indiquemos.

  ```
  var miCoche = new Coche();
  ```

- `delete`: se utiliza para borrar una propiedad de una instancia, dejando de estar accesible. Los métodos no se pueden eliminar de una instancia.

  ```
  delete miCoche.color;
  ```

- `in`: nos permite saber si una propiedad existe en el objeto que indiquemos, devolviendo un valor `true` si existe, y `false` si no. El nombre de la propiedad debe estar escrito como un *string*.

  ```
  alert("color" in Coche);
  ```

- `instanceof`: nos dice si la instancia especificada corresponde al objeto que indiquemos.

  ```
  alert(miCoche instanceof Coche);
  ```

- `this`: este operador sirve para referirse a un objeto como sustituto de su nombre. Su funcionamiento lo veremos a continuación para que pueda comprenderlo totalmente.

6.2.1. Definición de propiedades en un objeto

Según el tipo de constructor que hayamos empleado, las propiedades se definen de una manera u otra. Aquí veremos el procedimiento a seguir para cada uno de ellos.

En el caso de haber usado las llaves, las propiedades se escriben como pares de la forma propiedad:valor y separando cada uno de esos pares con una coma.

```
var miCoche = {marca: "Fiat", modelo: "Punto", color:
"Rojo"};
```

Si por el contrario hemos utilizado la sentencia `function`, entonces debemos hacer uso de la palabra reservada `this` para hacer referencia a elementos del propio objeto. La sintaxis básica sería esta:

```
function Coche() {
   // Definición de propiedades
   this.marca = "Fiat";
   this.modelo = "Punto";
   this.color = "Rojo";
}
```

Esto está bien pero tiene un inconveniente: ¡los valores que se asignan son fijos! Si creásemos instancias del objeto `Coche` todas tendrían los mismos datos. En el primer caso (uso de llaves) sólo podremos solventarlo asignando distintos valores a cada instancia que creemos directamente con las llaves, pero tendremos que repetir una y otra vez todos los nombres de las propiedades.

```
var miCoche1 = {marca: "Fiat", modelo: "Punto", color:
"Rojo"};
var miCoche2 = {marca: "Honda", modelo: "Civic", color:
"Azul"};
var miCoche3 = {marca: "Seat", modelo: "Ibiza", color:
"Negro"};
```

Sin embargo, si utilizamos la sentencia `function` lo tendremos mucho más fácil puesto que podremos usar los parámetros como los valores que se asignarán a las propiedades.


```
function Coche(valorMarca, valorModelo, valorColor) {
   // Definición de propiedades
   this.marca = valorMarca;
   this.modelo = valorModelo;
   this.color = valorColor;
}
// Definimos instancias
var miCoche1 = new Coche("Fiat", "Punto", "Rojo");
var miCoche2 = new Coche("Honda", "Civic", "Azul");
var miCoche3 = new Coche("Seat", "Ibiza", "Negro");
```

No habría ningún problema en caso de que los parámetros tengan el mismo nombre que las propiedades, ya que la sentencia `this` ayuda a diferenciar cuándo nos referimos a la propiedad propia del objeto y cuándo al parámetro.

```
function Coche(marca, modelo, color) {
   // Definición de propiedades
   this.marca = marca;
   this.modelo = modelo;
   this.color = color;
}
// Definimos instancias
var miCoche = new Coche("Fiat", "Punto", "Rojo");
var elCocheDeMiVida = new Coche("Lamborghini",
"Murciélago", "Blanco");
```

6.2.2. Definición de métodos

De igual modo que las propiedades, también podemos definir los métodos que tendrá nuestro objeto y las acciones que ejecutará cuando sea llamado. Para esto tenemos dos opciones: definir la función a la vez que el método o asignarle una función ya existente.

Definir funciones a la vez que un método

Esto lo haremos con la sentencia `function`, que ya conocemos, pero sin asignar un nombre de función puesto que lo que haremos es almacenar el contenido de dicha función dentro de la propiedad que representa al método. Veámoslo con ejemplos:

```
// Objeto definido con llaves
var miCoche = {arrancar: function() { alert("brummm"); }};
// Objeto definido con function
function Coche() {
   // Definición de métodos
   this.arrancar = function() { alert("brummm"); };
}
```

En ambos casos hemos definido el método `arrancar` de la misma forma: asignando una función a la propiedad del objeto que simboliza el método. Lo único que cambia es cómo realizamos esa asignación: con dos puntos (`:`) en un caso y con el operador de asignación (=) en el otro. Para llamar a cualquiera de los métodos basta con acceder a él como una función, junto con el operador punto (`.`) por estar dentro de un objeto:

```
// Método de objeto definido con llaves
miCoche.arrancar();
// Método de objeto definido con function
var miCoche = new Coche();
miCoche.arrancar();
```

Fíjese que para la segunda llamada es necesario crear previamente una instancia del objeto para poder acceder a las propiedades y métodos.

En el caso de que necesitásemos pasar parámetros a los métodos, tendremos que declarar nuestros métodos de una forma casi idéntica a como lo haríamos con una función normal.

```
// Objeto definido con llaves
var miCoche = {acelerar: function(velocidad) {
alert("acelerando hasta " + velocidad); }};
// Objeto definido con function
function Coche() {
   // Definición de métodos
   this.acelerar = function(velocidad) {
alert("acelerando hasta " + velocidad); };
}
```

Las llamadas a métodos con parámetros es exactamente igual que para una función normal.

```
// Método de objeto definido con llaves
miCoche.acelerar(100);
// Método de objeto definido con function
var miCoche = new Coche();
miCoche.acelerar(100);
```

Asignar funciones existentes a un método

Este otro sistema consigue el mismo resultado que el anterior, pero únicamente se diferencia en que las funciones que se asignan a los métodos ya se encuentran definidas fuera del objeto. Esto puede resultar más interesante, sobre todo desde el punto de vista del mantenimiento del código, ya que es más fácil localizar las funciones para hacer alguna modificación.

```
// Definición de función
function arrancarCoche(){
   alert("brummm");
}
// Asignar función a método
var miCoche = {arrancar: arrancarCoche};
function Coche() {
   // Definición de métodos
   this.arrancar = arrancarCoche;
}
```

Mediante esta técnica, cuando se haga una llamada al método `arrancar` del objeto, en realidad se estará ejecutando el código correspondiente a la función `arrancarCoche`. Las llamadas a los métodos no varían respecto a lo visto hasta ahora:

```
// Método de objeto definido con llaves
miCoche.arrancar();
// Método de objeto definido con function
var miCoche = new Coche();
miCoche.arrancar();
```

Si necesitamos que nuestra función reciba parámetros tan sólo habrá que indicarlos cuando definamos la función externa. La asignación al método no varía respecto a una función sin parámetros, y el método quedará preparado para recibir los mismos parámetros:

```
// Definición de función con parámetros
function acelerarCoche(velocidad){
   alert("acelerando hasta " + velocidad);
}
// Asignar función a método
var miCoche = {acelerar: acelerarCoche};
function Coche() {
   // Definición de métodos
   this.acelerar = acelerarCoche;
}
```

Veamos cómo serían entonces las llamadas con parámetros:

```
// Método de objeto definido con llaves
miCoche.acelerar(100);
// Método de objeto definido con function
var miCoche = new Coche();
miCoche.acelerar(100);
```

¡Exactamente igual que con el primer método! Esto es lo bueno de la flexibilidad que nos ofrece JavaScript: podemos definir las cosas de distintas maneras, pero la forma de utilizarlas es la misma.

Una característica que debemos tener en cuenta de los métodos es que su código podrá acceder a todas las propiedades del objeto simplemente con utilizar la sentencia `this`. Esto se puede hacer con cualquiera de los dos métodos que acabamos de ver:

```
<SCRIPT TYPE="text/javascript">
// Definición de función con acceso a propiedades
function mostrarColorCoche() {
   alert("El color del coche es: " + this.color);
}
// Definición del objeto
function Coche(marca, modelo, color) {
   // Definición de propiedades
   this.marca = marca;
   this.modelo = modelo;
   this.color = color;
   // Acceso a propiedades desde función definida dentro
   this.mostrarMiColor = function() { alert("Mi color
   es: " + this.color); };
   // Acceso a propiedades desde función existente
   this.mostrarColor = mostrarColorCoche;
}
var miCoche = new Coche("Fiat", "Punto", "Rojo");
miCoche.mostrarMiColor();
miCoche.mostrarColor();
</SCRIPT>
```

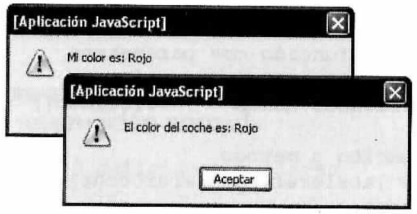


Figura 6.1. Acceso a propiedades desde un método.

6.3. Estructuras de manipulación de objetos

Como ya adelantamos en el capítulo referente a las estructuras de control, JavaScript nos ofrece algunas estructuras para acceder a las propiedades y métodos de un objeto.

6.3.1. Sentencia for - in

Gracias a esta sentencia podremos obtener todas las propiedades y métodos de una instancia de un objeto y también acceder a sus valores o ejecutarlos. Esto es útil cuando no conocemos sus nombres o queremos recorrer todas de principio a fin. La sintaxis que se debe seguir para utilizarla es la siguiente:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia for-in
   for (nombrePropiedad in instancia) {
      // Mi bloque de instrucciones
   }
</SCRIPT>
```

Si usamos esta estructura, el bloque de instrucciones que definamos será ejecutado para cada una de las propiedades. El nombre de la propiedad queda almacenado en una variable (`nombrePropiedad`) que puede ser declarada en el momento de escribir la sentencia, como ya ocurría en los bucles `for`. Esta variable la usaremos dentro del bloque para realizar los accesos que necesitemos. Si el objeto no tuviese ninguna propiedad el bucle no se ejecutaría, pasando directamente a la siguiente instrucción del *script*.

Imaginemos que usamos dentro de esta sentencia una instancia del objeto `Coche` con las propiedades `color`, `marca` y `modelo` y el método `arrancar`:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de la instancia
   var miCoche = new Coche("Fiat", "Punto", "Rojo");
   // Declaración de variable
   var mensaje = "Contenido de \"miCoche\":\n";
   // Sentencia for-in
   for (var miPropiedad in miCoche) {
      mensaje += "\t* " + miPropiedad + "\n";
   }
   // Mostramos las propiedades
   alert(mensaje);
</SCRIPT>
```

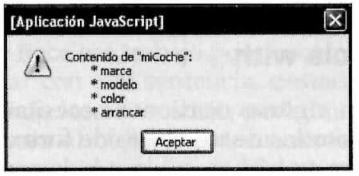


Figura 6.2. Sentencia for-in muestra las propiedades de la instancia.

Como resultado de ejecutar este *script* veríamos un mensaje con todos los nombres de las propiedades y métodos de nuestra instancia: `color`, `marca`, `modelo` y `arrancar`.

También sería posible acceder al valor de dichas propiedades mediante el uso de los corchetes ([ ]) como operador sobre objetos. Veamos cómo realizar esto con el ejemplo anterior:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de la instancia
   var miCoche = new Coche("Fiat", "Punto", "Rojo");
   // Declaración de variable
   var mensaje = "Valores en \"miCoche\":\n";
   // Sentencia for-in
   for (var miPropiedad in miCoche) {
      mensaje += "\t*  \'" + miPropiedad + "\' tiene
      valor: " + miCoche[miPropiedad] + "\n";
   }
   // Mostramos las propiedades y valores
   alert(mensaje);
</SCRIPT>
```

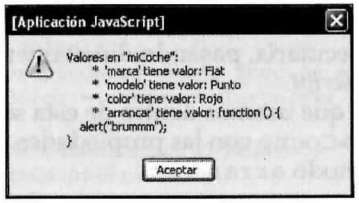


Figura 6.3. Sentencia for-in muestra las propiedades de la instancia y sus valores.

Gracias a este código obtendríamos el resultado de la figura 6.3. Como curiosidad, puede ver que el "valor" del método `arrancar` se muestra en realidad como el código que contiene.

Quizá ha pasado desapercibido, pero dese cuenta de que hemos accedido a todas las propiedades de una instancia sin saber sus nombres ni cuántas había. Esta es la gran ventaja de poder contar con esta estructura de control.

6.3.2. Sentencia with

Notará que en algunas ocasiones necesitará acceder a las propiedades y métodos de un objeto de forma muy consecutiva, haciendo que su código sea poco elegante y pesado de escribir. Por ejemplo:


```
<SCRIPT TYPE="text/javascript">
   // Accesos consecutivos a propiedades y métodos
   alert(miCoche.color);
   alert(miCoche.marca);
   alert(miCoche.modelo);
   miCoche.arrancar();
</SCRIPT>
```

Como ve, para mostrar los valores de estas tres propiedades y llamar al método debemos escribir repetidamente el nombre del objeto al que queremos acceder (`miCoche`). Esto resultará aún más molesto cuantas más operaciones necesitemos hacer sobre las propiedades o métodos. Sin embargo, gracias a la sentencia `with`, podremos fijar el objeto al que vamos a acceder y escribir únicamente el nombre de la propiedad o método para acceder a ellos. Veamos primero la sintaxis:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia with
   with (objeto) {
      // Bloque de instrucciones sobre las propiedades
      // y métodos
   }
</SCRIPT>
```

Ahora usaremos esta nueva estructura con el ejemplo que vimos al principio del apartado para que pueda obsevar la diferencia:

```
<SCRIPT TYPE="text/javascript">
   // Sentencia with
   with (miCoche) {
      alert(color);
      alert(marca);
      alert(modelo);
      arrancar();
   }
</SCRIPT>
```

¿Ha visto? Ya no es necesario escribir una y otra vez el nombre de la instancia, lo cual hace más cómodo y limpio el acceder tanto a las propiedades como al método. Note que también se ha eliminado el operador punto (.) puesto que la sentencia `with` hace ese trabajo por nosotros.

Para terminar con esta sentencia, destacar que dentro del bloque de instrucciones también podremos usar variables y funciones ajenas al objeto, es decir, otras variables o funciones que tengamos definidas en cualquier parte de nuestro código.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable
   var velocidad = 100;
   // Declaración de función
   function subirVelocidad() {
      velocidad += 20;
   }
   // Sentencia with
   with (miCoche) {
      alert(color);
      alert(marca);
      alert(modelo);
      acelerar(velocidad);
      subirVelocidad();
      acelerar(velocidad);
   }
</SCRIPT>
```

Tras ejecutar este *script*, se nos mostrarán mensajes con el color, marca y modelo de nuestra instancia `miCoche`. A continuación se llama al método `acelerar` de la instancia con la variable `velocidad` como parámetro, con valor 100. Acto seguido se ejecuta una función ajena al objeto que incrementa esa variable `velocidad` y, para terminar, se vuelve a llamar al método `acelerar` con su parámetro mostrando el valor 120.

7 Objetos de JavaScript

Dentro del núcleo de JavaScript existen varios objetos que ya están definidos y tienen una serie de funcionalidades tremendamente útiles en muchos casos.

De cada uno de estos objetos expondremos el constructor, propiedades y métodos que tenga asociados, así como ejemplos de algunos de ellos para mejorar su comprensión o utilidad.

Todos los objetos predefinidos de JavaScript disponen del método `toString()`, que devuelve el objeto expresado como una cadena, realizando para ello conversiones implícitas si es necesario. Para hacer uso de él, basta con hacer la llamada desde el objeto que nos interese: `miObjeto.toString()`.

7.1. Objetos envoltorio

Los objetos que vamos a ver en este apartado se tratan en realidad de un envoltorio (*wrapper*) de los tipos de datos predefinidos (booleano, número y cadena o *string*), como ya adelantamos en el capítulo 2. Con esto queremos decir que crear una instancia de uno de estos objetos mediante la sentencia `new` sería lo mismo que declarar una variable de esos tipos directamente con el valor, pudiendo por tanto éstas últimas acceder a la mayoría de las propiedades y métodos de su objeto envoltorio.

7.1.1. Objeto Boolean

Como puede imaginar, este objeto expresa un valor como un booleano, o lo que es lo mismo, como `true` o `false`.

Constructor

Para crear una instancia debe hacer lo siguiente:

```
<SCRIPT TYPE="text/javascript">
   // Objeto Boolean
   var nombre_variable = new Boolean(valor);
</SCRIPT>
```

Aquí debe tener cuidado puesto que la conversión al tipo booleano no se realiza de una forma intuitiva:

- Si el valor es omitido o es 0, -0, `null`, cadena vacía (`""`), valor booleano `false`, valor `undefined` o valor `NaN`, entonces este objeto devuelve una instancia con valor `false`.
- En cualquier otro caso, devuelve `true`.

Advertencia: *Tenga presente estas reglas porque casos como la cadena `"false"` no hacen que el objeto devuelva `false` como podríamos pensar, sino que obtenemos un `true`.*

Tabla 7.1. Ejemplos del constructor Boolean.

Constructor	*Resultado*
new Boolean()	false
new Boolean(1)	true
new Boolean(0)	false
new Boolean(65)	true
new Boolean(false)	false
new Boolean(true)	true
new Boolean("")	false
new Boolean("0")	true
new Boolean("1")	true
new Boolean("true")	true
new Boolean("false")	true

Propiedades y métodos

Este objeto no tiene ninguna propiedad o método. No hemos empezado por el más completo, pero no se preocupe que los siguientes traerán más sorpresas.

7.1.2. Objeto Number

Este objeto nos permite crear instancias que tengan un valor numérico (entero o real).

Constructor

Su constructor es muy simple:

```
<SCRIPT TYPE="text/javascript">
   // Objeto Number
   var nombre_variable = new Number(valor);
</SCRIPT>
```

Al igual que ocurre con el objeto `Boolean`, en función del valor que se reciba como parámetro en el constructor se obtendrán distintos resultados:

- Si no recibe un valor, devuelve un 0.
- Si recibe un número (expresado como tal o en forma de *string*), entonces inicializa el objeto con ese valor. La cadena vacía equivale a un 0.
- En caso de que reciba un valor no numérico, devuelve `NaN`.
- El valor `false` se interpreta como un 0 y `true` como un 1.

Tabla 7.2. Ejemplos del constructor Number.

Constructor	*Resultado*
new Number()	0
new Number(1)	1
new Number(0)	0
new Number(65)	65
new Number(false)	0
new Number(true)	1
new Number("")	0
new Boolean("0")	0
new Boolean("1")	1
new Number("true")	NaN
new Number("false")	NaN

Propiedades y métodos

Este objeto (ahora sí) tiene una serie de propiedades, que listaremos a continuación, con una peculiaridad: únicamente son accesibles desde el propio objeto y no desde sus instancias.

- `MAX_VALUE`: Indica el mayor número que se puede representar en JavaScript.
- `MIN_VALUE`: Lo mismo, pero con el menor número representable.
- `NaN` (*Not A Number*): Valor especial, que ya vimos junto con los tipos de datos, que indica que el valor no es un número.
- `POSITIVE_INFINITY`: Valor especial (`Infinity`) que indica que el número es mayor que el máximo representable (`MAX_VALUE`).
- `NEGATIVE_INFINITY`: Valor especial (`-Infinity`) que indica que el número es menor que el mínimo representable (`MIN_VALUE`).

```
<SCRIPT TYPE="text/javascript">
   // Instancia de Number
   var numero = new Number(5);
   // Propiedades del objeto Number
   alert("Propiedades de Number:" + "\n" +
   "\tMAX_VALUE: " + Number.MAX_VALUE + "\n" +
   "\tMIN_VALUE: " + Number.MIN_VALUE + "\n" +
   "\tNaN: " + Number.NaN + "\n" +
   "\tPOSITIVE_INFINITY: " + Number.POSITIVE_
   INFINITY + "\n" +
   "\tNEGATIVE_INFINITY: " + Number.NEGATIVE_
   INFINITY);
   // Propiedades de la instancia
   alert("Propiedades de numero:" + "\n" +
   "\tMAX_VALUE: " + numero.MAX_VALUE + "\n" +
   "\tMIN_VALUE: " + numero.MIN_VALUE + "\n" +
   "\tNaN: " + numero. NaN + "\n" +
   "\tPOSITIVE_INFINITY: " + numero.POSITIVE_
   INFINITY + "\n" +
   "\tNEGATIVE_INFINITY: " + numero.NEGATIVE_
   INFINITY);
</SCRIPT>
```

Como puede ver en la figura 7.1, las propiedades no son accesibles desde las instancias creadas a partir del objeto.

Pasamos ahora a ver los métodos, donde ocurre justo lo contrario: sólo pueden ser utilizados desde las instancias. Esto tiene sentido puesto que hace falta un valor sobre el que aplicar las operaciones que realizan. Los métodos no modifican

el valor de la instancia sino que devuelven el resultado de su operación, por lo que si queremos usar éste en otra parte de nuestro código haría falta almacenarlo previamente.

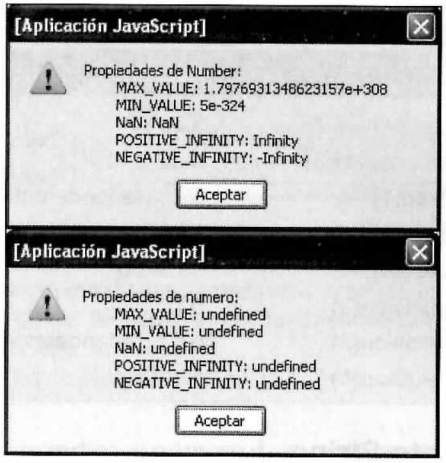


Figura 7.1. Propiedades del objeto Number y una instancia del mismo.

- `toExponential(decimales)`: Convierte un valor en una expresión exponencial (e o E), con el número de posiciones decimales especificada por el parámetro `decimales`.
- `toFixed(decimales)`: Ajusta el número de posiciones decimales de un valor a las definidas por `decimales`, redondeando si se reduce la cantidad de decimales actual.
- `toPrecision(digitos)`: Ajusta el número de dígitos totales de un valor (parte entera y decimal) a los especificados por `digitos`. Aquí también se redondea si se eliminan decimales.

El parámetro de estos métodos debe ser siempre un número positivo. Si se utiliza un número real, se omitirán sus decimales.

> **Nota**: *Recuerde que una variable con un valor numérico escrito directamente se comporta igual que una instancia, además es la forma más común de escribirlas. Por tanto:* `var x = new Number(5)` *equivale a* `var x = 5`.

En la tabla 7.3 podrá ver cómo funcionan los métodos de este objeto. Para los ejemplos se utilizará la variable `numero` con valor 53.27.

Tabla 7.3. Ejemplos de los métodos de Number.

Ejemplo	*Resultado*
numero.toExponential(1)	5.3e+1
numero.toExponential(3)	5.327e+1
numero.toFixed(1)	53.3 (redondeando)
numero.toFixed(2)	53.27
numero.toFixed(4)	53.2700
numero.toPrecision(2)	53
numero.toPrecision(3)	53.3 (redondeando)
numero.toPrecision(4)	53.27

7.1.3. Objeto String

El objeto `String` nos permite realizar operaciones, bastante útiles, sobre cadenas de texto.

Constructor

Para crear una instancia de este objeto se debe escribir lo siguiente:

```
<SCRIPT TYPE="text/javascript">
   // Objeto String
   var nombre_variable = new String(valor);
</SCRIPT>
```

El valor que le pasemos al constructor será convertido a una cadena de texto en todos los casos. Si se omite, se creará una cadena vacía.

Tabla 7.4. Ejemplos del constructor String.

Constructor	*Resultado*
new String()	"" (cadena vacía)
new String(1)	"1"
new String(0)	"0"
new String(65)	"65"

Constructor	*Resultado*
new String(false)	"false"
new String(true)	"true"
new String("")	"" (cadena vacía)
new String("0")	"0"
new String("1")	"1"
new String("true")	"true"
new String("false")	"false"

Propiedades y métodos

Todas las propiedades y métodos que se listan a continuación se utilizan siempre sobre las instancias que hayamos creado de este objeto.

La lista de propiedades es corta... sólo una y fácil de entender:

- `length`: indica el número de caracteres que contiene la cadena.

```
<SCRIPT TYPE="text/javascript">
   // Instancia de String
   var nombre = new String("Pepe");
   alert(nombre.length); // veremos un 4
   nombre = "Yolanda";
   alert(nombre.length); // veremos un 7
   nombre = "";
   alert(nombre.length); // veremos un 0
</SCRIPT>
```

Tras este "duro esfuerzo" vamos con los métodos, que sí son unos cuantos. No los voy a listar uno a uno ya que se pueden agrupar en cierta medida por su similitud funcional.

> **Nota**: *El primer carácter de una cadena se considera que está en la posición cero (0). Deberemos tener esto en cuenta en algunos de los métodos del objeto* `String`.

- `anchor(nombre)`, `link(direccion)`: Crean un ancla y un enlace HTML respectivamente (etiqueta `<A>`). El primero coloca el valor del parámetro `nombre` dentro del atributo `NAME` del ancla, y el segundo el valor de `direccion` en el atributo `HREF`. El valor de la cadena se coloca entre las etiquetas `<A>` y `</A>`.

- `big()`, `blink()`, `bold()`, `fixed()`, `italics()`, `small()`, `strike()`, `sub()`, `sup()`: Encierran el valor de la cadena entre las etiquetas HTML de formato (en el mismo orden): <BIG>, <BLINK>, <B>, <TT>, <I>, <SMALL>, <STRIKE>, <SUB> y <SUP>. El valor de la cadena se coloca entre la etiqueta y su cierre.
- `fontcolor(color)`, `fontsize(tamaño)`: Cambian el color y el tamaño de un texto HTML a través de los atributos `COLOR` y `SIZE` de la etiqueta <FONT>. El color puede expresarse como una cadena ("Red"), un valor RGB (255, 0, 0) o un código hexadecimal (#FF0000). Por otro lado, el tamaño del texto se define mediante un número del 1 al 7, siendo el 1 el tamaño más pequeño y el 7 el mayor. El valor de la cadena se coloca entre las etiquetas <FONT> y </FONT>.
- `charAt(posicion)`, `charCodeAt(posicion)`: Devuelven el carácter y el código UNICODE del carácter que se encuentre en la posición indicada por su parámetro. El primer carácter está en la posición cero (0).
- `indexOf(texto, inicio)`, `lastIndexOf(texto, inicio)`: Devuelven la posición de la primera y última coincidencia, respectivamente, de `texto` dentro la cadena. Si no se encuentra entonces se devuelve el valor -1. El parámetro `inicio` es opcional e indica la posición en la que se debe empezar a buscar. El método `indexOf` busca desde `inicio` hacia el final de la cadena y `lastIndexOf` desde `inicio` hacia el comienzo. Además, ambos métodos diferencian entre minúsculas y mayúculas.
- `concat(cadena1, cadena2, ..., cadenaN)`: Une el valor de la cadena con el de las cadenas pasadas como parámetros (se puede poner una o varias). Equivale a utilizar el operador sobre cadenas (+).
- `fromCharCode(codigo1, codigo2, ..., codigoN)`: Construye una cadena a partir de una lista de valores UNICODE. Este método es exclusivo del objeto `String` y no puede ser utilizado desde una instancia del mismo.
- `split(separador, limite)`: Divide la cadena en un subconjunto de cadenas tomando el parámetro `separador` como delimitador de cadenas. En este caso se devuelve un objeto `Array` (aprenderemos a usar este objeto justo a continuación de `String`) con todas las cadenas separadas. Si el parámetro opcional `limite` tiene un valor, entonces se devuelven tantas subcadenas como indique.
- `substring(inicio, fin)`, `substr(inicio, longitud)`, `slice(inicio, fin)`: Extrae y devuelve la porción de la cadena que se encuentra entre las posiciones `inicio` y `fin` o una cantidad de caracteres (`longitud`) a partir de la posición `inicio`. Si se omite el segundo parámetro en estos métodos, entonces se devuelve la parte entre `inicio` y el final de la cadena.
- `match(expresion)`, `replace(expresion, reemplazo)`, `search(expresion)`: Estos métodos buscan una coincidencia de `expresion` dentro de la cadena o, en el caso de `replace`, sustituye la primera ocurrencia por el valor de `reemplazo`. En todas ellas la búsqueda es sensible a mayúsculas y minúsculas. El parámetro `expresion` puede ser una cadena o una expresión regular. Los valores devueltos son distintos en cada uno:
  - `match`: devuelve el valor de `expresion` si lo encuentra y `null` en caso contrario.
  - `replace`: devuelve una cadena con los reemplazos realizados, si se encontraron coincidencias. Tenga en cuenta que sólo se reemplazará la primera ocurrencia de la expresión especificada.
  - `search`: este método devuelve la posición donde se encuentra la primera ocurrencia de la expresión, o -1 si no hay coincidencias. Es similar al método `indexOf`.
- `toLowerCase()`, `toUpperCase()`: Devuelve la cadena con todos los caracteres en minúsculas o mayúsculas, respectivamente.

Truco: *Los métodos que trabajan con posiciones como parámetro esperan un valor positivo pero en algunos es posible utilizar también negativos. En estos casos, el valor -1 representa la última posición de la cadena, que equivale a escribir* `miCadena.length - 1`.

En la tabla 7.5 veremos el funcionamiento de la mayoría de estos métodos. Para los ejemplos se utilizará la variable `texto` con valor "JavaScript mola".

Tabla 7.5. Ejemplos de los métodos de String.

Ejemplo	*Resultado*
texto.anchor("lista")	<A NAME="lista">JavaScript mola</A>
texto.bold()	<B>JavaScript mola</B>
texto.charAt(0)	J
texto.charAt(2)	v
texto.indexOf("a")	1
texto.indexOf("a", 4)	14 (busca desde "S" en adelante)
texto.indexOf("z")	-1 (no existe)
texto.lastIndexOf("a")	14
texto.lastIndexOf ("a", 0)	-1 (no existe. busca desde "J" hacia atrás)
texto.concat " (" y mucho")	JavaScript mola y mucho"
texto.concat (" y mucho")	"JavaScript mola y mucho"
String.fromCharCode (72, 79, 76, 65)	"HOLA" (sólo objeto String)
texto.split(" ")	Array con cadenas "JavaScript" y "mola"
texto.substring(0, 4)	Java
texto.substring(4)	Script mola
texto.substr(4, 6)	Script
texto.substr(11)	mola
texto.substr(-4)	mola
texto.match("mo")	mo
texto.match("z")	null
texto.replace ("JavaScript", "JS")	JS mola
texto.replace ("javascript", "JS")	JavaScript mola
texto.search("va")	2
texto.search("z")	-1
texto.toLowerCase()	javascript mola
texto.toUpperCase()	JAVASCRIPT MOLA

Recuerde que estos métodos no modifican la instancia, por lo que debe almacenar el valor en una variable si quiere usar el resultado posteriormente.

```
<SCRIPT TYPE="text/javascript">
   // Instancia de String
   var texto = new String("JavaScript mola");
   // Definición de variable
   var textoMayusculas;
   // Uso de métodos de String
   textoMayusculas = texto.toUpperCase();
   texto = texto.concat(" y mucho");
   alert(textoMayusculas + "\n" + texto);
</SCRIPT>
```

Con el código propuesto veremos un mensaje con el texto "JAVASCRIPT MOLA" seguido de "JavaScript mola y mucho" una línea por debajo. Como puede observar, se puede utilizar la misma variable con la que creamos la instancia para almacenar los resultados de los métodos.

7.2. Objeto Array

Las variables de JavaScript normalmente almacenan un único valor, el cual podemos recuperar cuando utilizamos su nombre dentro de una expresión. Esto se nos puede quedar corto en determinadas ocasiones, como por ejemplo si queremos almacenar por separado los platos del menú de un restaurante para después mostrarlos en nuestra página. Gracias a los *arrays* podremos almacenar todos estos platos en una única variable como si ésta tuviera huecos o casillas que se rellenan con los valores, permitiéndonos acceder a cada plato individual con tan sólo indicar cuál es su hueco, es decir, la posición dentro del *array*. A esta posición se le denomina índice.

7.2.1. Constructor

Para crear un *array* tenemos varios constructores:

1. `new Array()`: crea un *array* vacío, sin huecos donde almacenar nuestros datos. Esto es útil cuando no sabemos la longitud inicial que tendrá el *array* cuando lo utilicemos en nuestro código.
2. `new Array(longitud)`: en este caso el *array* se crea con tantos huecos como indique el parámetro `longitud`.

3. `new Array(valor1, valor2, ..., valorN)`: con este constructor crearemos un *array* con tantos huecos o posiciones como valores hayamos pasado como parámetros y, además, dichos huecos irán rellenos con el valor correspondiente.

El tipo de datos de los valores almacenados en un *array* puede ser cualquiera, incluso puede ser distinto en cada una de las posiciones. En otros lenguajes, los *arrays* están forzados a que los datos sean siempre del mismo tipo (entero, booleano, etc).

Tabla 7.6. Ejemplos del constructor Array.

Constructor	*Resultado*
new Array()	Array vacío y sin posiciones.
new Array(5)	Array con 5 posiciones vacías.
new Array("5")	Array con 1 posición y 1 valor de tipo cadena
new Array("a", "b", "c")	Array con 3 posiciones y 3 valores de tipo cadena.
new Array(1, 2, 3)	Array con 3 posiciones y 3 valores de tipo numérico.
new Array("a", 2, false)	Array con 3 posiciones y 3 valores de tipos distintos.

También tenemos la posibilidad de crear un *array* mediante el uso de corchetes (`[]`), lo que equivaldría a crear una instancia sin usar el constructor tal y como hemos visto en otros objetos.

```
<SCRIPT TYPE="text/javascript">
   // Array con corchetes
   var nombre_variable = []; // Array vacío
   var nombre_variable = ["a", "b", "c"]; // Array con 3
   // posiciones y 3 valores de tipo cadena
   var nombre_variable = [1, 2, 3]; // Array con 3
   // posiciones y 3 valores de tipo numérico
   var nombre_variable = ["a", 2, false]; // Array con 3
   // posiciones y 3 valores de tipos distintos
</SCRIPT>
```

7.2.2. Trabajar con un array

Para ver el contenido de un *array* es suficiente con utilizarlo junto con un `alert` o ejecutar su método `toString()` quedando los valores separados por comas. Esto último es lo

que hace `alert` de forma automática para poder representar el objeto como una cadena.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array("Arroz", "Filete con patatas",
   "Flan");
   // Visualizar contenido
   alert("El menú de hoy es: " + menu);
</SCRIPT>
```

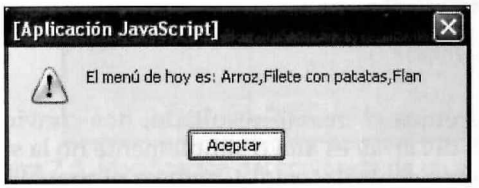


Figura 7.2. Visualización del contenido de un array.

Ahora bien, si queremos acceder a cada posición de un *array*, con el fin de obtener o modificar su contenido, tendremos que usar, a continuación del nombre de la variable, los corchetes e indicar la posición (índice) sobre la que queremos trabajar. La primera posición está representada por el índice 0, como ocurre con las cadenas.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array("Arroz", "Filete con patatas",
   "Flan");
   // Acceso a los valores
   alert("El menú de hoy es:\n" +
      "\t- " + menu[0] + "\n" +
      "\t- " + menu[1] + "\n" +
      "\t- " + menu[2]);
</SCRIPT>
```

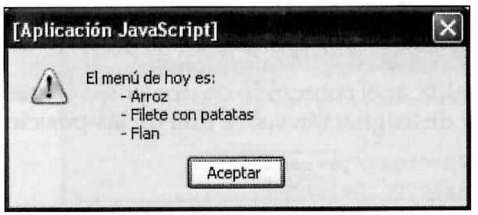


Figura 7.3. Acceso al contenido de un array.

Una forma más cómoda de acceder a los valores es crear un bucle `for`, utilizando el contador como índice del *array*. Veamos el ejemplo anterior escrito con un bucle:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variables
   var menu = new Array("Arroz", "Filete con patatas",
   "Flan");
   var mensaje = "El menú de hoy es:\n";
   // Acceso a los valores con un bucle
   for(var i=0; i<3; i++) {
      mensaje += "\t- " + menu[i] + "\n";
   }
   alert(mensaje);
</SCRIPT>
```

Obtendremos el mismo resultado, nos conviene cuando la longitud del *array* es alta o simplemente no la sabemos (ya veremos cómo controlar esto). Como ve, el contador comienza en 0, para acceder a la primera posición, y aumenta hasta el valor 2, que corresponde a la tercera y última.

> **Nota**: *Si intentamos acceder a una posición que no existe, entonces JavaScript nos devolverá un valor* `undefined`.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array("Arroz", "Filete con patatas",
   "Flan");
   // Acceso un valor que no existe
   alert("El último plato de hoy es: " + menu[3]);
</SCRIPT>
```

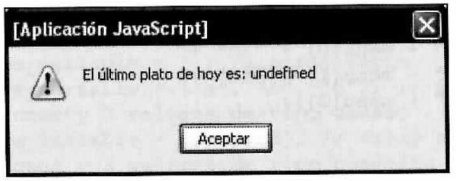


Figura 7.4. Acceso a una posición inexistente en el array.

Para modificar el contenido de una posición basta con usar el operador de asignación sobre una de las posiciones.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array("Arroz", "Filete con patatas",
   "Flan");
```



```
   // Modificación de un valor
   alert("El menú de hoy es: " + menu);
   menu[0] = "Macarrones";
   alert("El menú de hoy es: " + menu);
</SCRIPT>
```

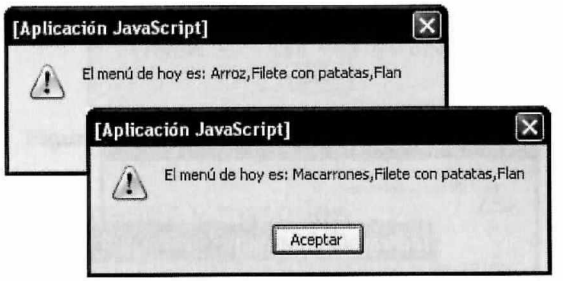


Figura 7.5. Modificación del contenido de un array.

Si el *array* tiene los huecos creados (se especificó una longitud o número de elementos) se rellenarán con los nuevos valores, pero si no es así entonces se crean sobre la marcha aunque superen la longitud inicialmente fijada. De este modo es posible tener un *array* vacío y asignarle tantos valores como queramos o tener un *array* de una longitud fijada y ampliarla.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array("Arroz", "Filete con patatas",
   "Flan");
   // Ampliación de un valor
   alert("El menú de hoy es: " + menu);
   menu[3] = "Café";
   alert("El menú de hoy es: " + menu);
</SCRIPT>
```

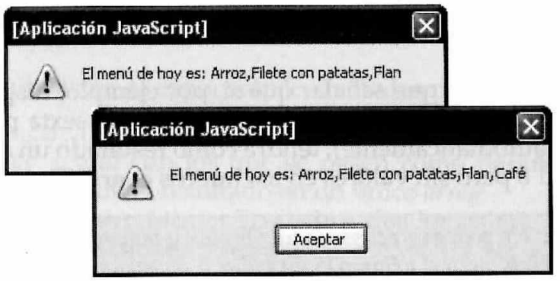


Figura 7.6. Ampliación del contenido de un array con valores.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array();
   // Ampliación de valores
   alert("El menú de hoy es: " + menu); // vacío
   menu[0] = "Arroz";
   menu[1] = "Filete con patatas";
   menu[2] = "Flan";
   alert("El menú de hoy es: " + menu);
</SCRIPT>
```

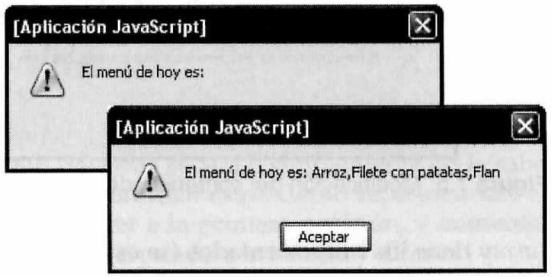


Figura 7.7. Ampliación del contenido de un array vacío.

Debe saber que si ha especificado una longitud, no es obligatorio rellenar el *array* completo, aunque las posiciones libres estarán disponibles.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array(5);
   // Ampliación de valores
   alert("El menú de hoy es: " + menu); // 5 posiciones
                                        // vacías
   menu[0] = "Arroz";
   menu[1] = "Filete con patatas";
   menu[2] = "Flan";
   alert("El menú de hoy es: " + menu); // dos posiciones
                                        // libres
</SCRIPT>
```

También hay que señalar que si, por ejemplo, dispone de un *array* de longitud 3 y añade un valor en la sexta posición (creada automáticamente), tendrá como resultado un *array* de longitud 6 pero sin valores en los huecos cuarto y quinto.

> **Nota**: *No podemos acceder a una posición mayor a la longitud del array, pero sí asignarle un valor y recuperarlo posteriormente, puesto que se creará automáticamente.*

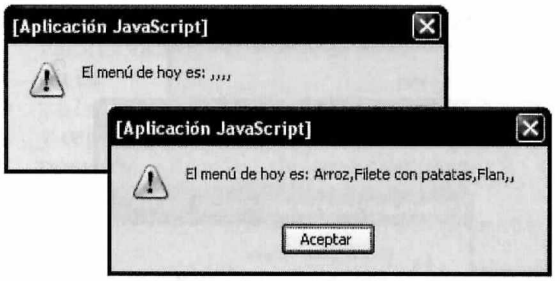


Figura 7.8. Array con dos posiciones libres.

7.2.3. Propiedades y métodos

El objeto `Array` sólo dispone de una propiedad:

- `length`: nos indicará la longitud del *array*, en concreto la posición del último elemento. Aquí no influye el que los huecos tengan valor o no, sino que la posición exista y sea accesible.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var menu = new Array();
   // Visualización de la longitud
   alert("El menú tiene " + menu.length + " platos");
   // vacío
   menu = new Array(3);
   alert("El menú tiene " + menu.length + " platos");
   // 3 posiciones vacías
   menu = new Array("Arroz", "Filete con patatas",
   "Flan");
   alert("El menú tiene " + menu.length + " platos");
   // array con valores
   menu[4] = "Copa de licor"; // quinta posición
   alert("El menú tiene " + menu.length + " platos");
   // array con algunas posiciones vacías
</SCRIPT>
```

A continuación se explicarán los métodos para este objeto:

- `concat(array1, array2, ..., arrayN)`: Une el *array* actual con los que se pasen como parámetros, devolviendo el resultado en un único *array*.
- `join(separador)`: Traslada todos los valores de un *array* a una cadena intercalando el carácter `separador` entre ellos. Si no se especifica un separador se utiliza la coma (`,`) por defecto.

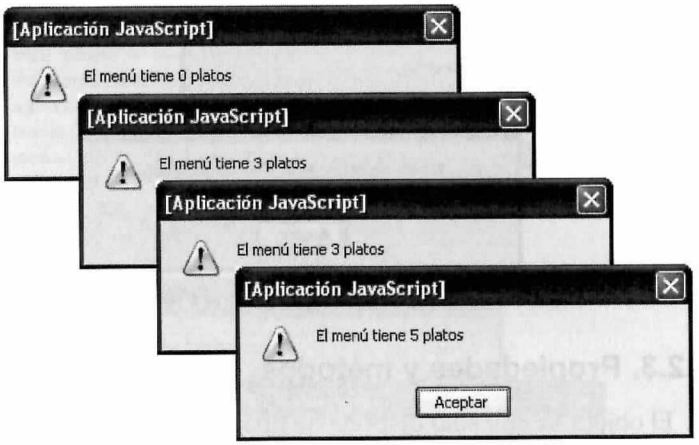


Figura 7.9. Longitud de distintos arrays.

- `pop()`: Devuelve y elimina el valor en la última posición de un *array*. Al mismo tiempo se borra la posición que ocupaba, reduciendo así la longitud del *array*.
- `push(valor1, valor2, ..., valorN)`: Añade uno o más valores al final del *array* y devuelve la longitud del mismo. Si el *array* no dispone de posiciones suficientes para almacenar los valores se crearán nuevas automáticamente.
- `shift()`: Realiza la misma operación que `pop`, pero con el primer valor.
- `unshift(valor1, valor2, ..., valorN)`: Se comporta igual que `push`, pero insertando uno o varios valores al principio del *array*.
- `slice(inicio, fin)`: Devuelve un nuevo *array* que tiene únicamente los valores que estén entre las posiciones `inicio` y `fin` sin incluir éste, es decir, el último valor en añadirse será el de la posición `fin-1`. Si se omite el segundo parámetro entonces se toman todos los valores desde `inicio` hasta la última posición.
- `splice(inicio, longitud_borrado, valor1, valor2, ..., valorN)`: Añade y/o elimina valores de un array. Este método, a diferencia del resto, no devuelve un nuevo *array* sino que modifica el actual. La función de cada parámetro es la siguiente:
  - `inicio`: indica dónde vamos a comenzar a añadir/borrar valores.

  - `longitud_borrado`: este método nos dice cuántos valores vamos a borrar (puede ser ninguno, poniendo un 0).
  - `valor1, valor2, ..., valorN`: son opcionales y representan los valores que se van a añadir en la posición `inicio`, después de borrar la cantidad `longitud_borrado` de elementos. Estos valores no sobrescriben los existentes, sino que se intercalan con ellos.
- `reverse()`: Invierte el orden de todos los valores del *array*, de forma que el primero pasa a ser el último y el último el primero. No se devuelve un nuevo *array* sino que modifica el actual.
- `sort(funcion)`: Ordena alfabéticamente los valores del *array*, modificándolo en lugar de devolver uno nuevo. Si se pasa una función como parámetro entonces es utilizada para determinar cómo deben ordenarse los valores. Esa función debe comparar valores de dos en dos y, para que el método `sort` pueda "comprenderla", tiene que devolver unos valores concretos:
  - Si el primer valor se considera menor que el segundo, entonces hay que devolver -1 (o cualquier número negativo).
  - Si el primer valor se considera mayor que el segundo, se devuelve un 1 (o cualquier número positivo).
  - Si ambos valores se consideran equivalentes, debe devolver un 0.

En la tabla 7.7 puede encontrar algunos ejemplos de estos métodos. En ellos tendremos definido el *array* `semana` con los valores "lunes", "martes" y "jueves".

Tabla 7.7. Ejemplos de los métodos de Array.

Ejemplo	*Resultado*
semana.concat(new Array("viernes"))	lunes, martes, jueves, viernes
semana.join("-")	lunes-martes-miércoles
semana.pop()	Array semana: lunes, martes. Valor devuelto: jueves
semana.push("viernes")	lunes, martes, jueves, viernes

Ejemplo	Resultado
semana.shift()	Array semana: martes, jueves. Valor devuelto: lunes
semana.unshift("domingo")	domingo, lunes, martes, jueves
semana.slice(0, 2)	lunes, martes
semana.slice(1, 3)	martes, jueves
semana.slice(0)	lunes, martes, jueves
semana.splice(1, 1)	Array semana: lunes, jueves
semana.splice(2, 0, "miércoles")	Array semana: lunes, martes, miércoles, jueves
semana.splice(1, 1, "miércoles")	Array semana: lunes, miércoles, jueves
semana.reverse()	Array semana: jueves, martes, lunes
semana.sort()	Array semana: jueves, lunes, martes

Vamos a ver ahora un ejemplo del método `sort` empleando una función que ordene las cadenas por su longitud en lugar de alfabéticamente, como hace por defecto el método.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de array
   var semana = new Array("lunes", "miércoles", "jueves");
   // Declaración de función
   function compararLongitud(cadena1, cadena2) {
      if (cadena1.length > cadena2.length)
         return 1;
      else if (cadena1.length < cadena2.length)
         return -1;
      else
         return 0;
   }
   // Ordenación por defecto. Saldrá "jueves, lunes,
   // miércoles"
   alert(semana.sort());
   // Ordenación con función. Saldrá "lunes, jueves,
   // miércoles"
   alert(semana.sort(compararLongitud));
</SCRIPT>
```


7.2.4. Arrays multidimensionales

Los *arrays* con los que hemos trabajado hasta ahora eran bastante simples ya que únicamente contenían una lista de valores. Esto se conoce como *array* unidimensional.

JavaScript nos permite almacenar en la posición de un *array* otro *array*, o lo que es lo mismo, almacenar una lista de valores dentro de una única posición en lugar de tener un solo valor. A esto se le denomina *array* bidimensional, aunque probablemente le suene más el término matriz o tabla.

Así mismo, dentro de la posición de un *array* bidimensional podremos volver a almacenar otro *array*, y así sucesivamente hasta llegar al número de dimensiones que queramos.

Como todo esto explicado con palabras resulta un tanto lioso, vamos a ir desarrollando un ejemplo hasta llegar a tener un *array* de tres dimensiones. Partiremos con unos *arrays* unidimensionales que almacenan una lista de asignaturas:

```
<SCRIPT TYPE="text/javascript">
   // Arrays unidimensionales
   var asignaturas_letras = new Array("Historia",
   "Latín", "Filosofía");
   var asignaturas_ciencias = new Array("Física",
   "Química", "Matemáticas");
   // Mostramos los arrays unidimensionales
   alert(asignaturas_letras + "\n" +
   asignaturas_ciencias);
</SCRIPT>
```

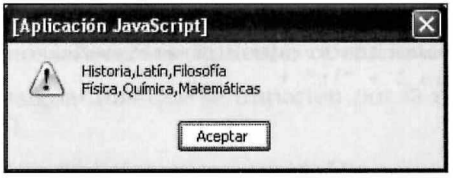


Figura 7.10. Arrays unidimensionales.

Para acceder a cada uno de los valores de un *array*, ya hemos visto que es tan sencillo como escribir un índice entre corchetes (`[]`):

```
<SCRIPT TYPE="text/javascript">
   // Arrays unidimensionales
   var asignaturas_letras = new Array("Historia",
   "Latín", "Filosofía");
   var asignaturas_ciencias = new Array("Física",
   "Química", "Matemáticas");
```

```
    // Mostramos los primeros valores
    alert(asignaturas_letras[0] + "\n" +
    asignaturas_ciencias[0]);
</SCRIPT>
```

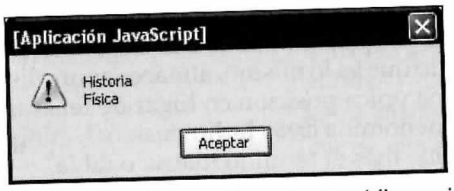


Figura 7.11. Primer valor de los arrays unidimensionales.

Hasta aquí no hay nada nuevo, así que vamos a subir de nivel. Vamos a crear unos *arrays* bidimensionales que representarán las asignaturas que se imparten en un aula. Cada uno de estos *arrays* (aulas) tendrá dos posiciones que van a simbolizar las asignaturas que se dan por la mañana y las que se dan por la tarde. ¡Vamos al lío!

```
<SCRIPT TYPE="text/javascript">
    // Arrays unidimensionales
    var asignaturas_letras = new Array("Historia",
    "Latín", "Filosofía");
    var asignaturas_ciencias = new Array("Física",
    "Química", "Matemáticas", "Biología");
    // Arrays bidimensionales
    var aula_1 = new Array(asignaturas_ciencias,
    asignaturas_letras);
    var aula_2 = new Array(asignaturas_letras,
    asignaturas_ciencias);
    // Mostramos los arrays bidimensionales
    alert(aula_1 + "\n" +
    aula_2);
</SCRIPT>
```

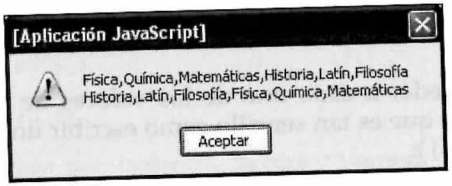


Figura 7.12. Arrays bidimensionales.

La figura 7.12 puede dar la impresión de que seguimos trabajando con *arrays* unidimensionales con 6 posiciones cada uno, pero eso es porque JavaScript muestra los valores separados

por comas, así que como en este caso los valores son a su vez otro *array* JavaScript representa el primer *array*, después una coma y por último el segundo *array*.

Para acceder a los valores debemos escribir dos parejas de corchetes (`[][]`), de manera que el primero sirve para acceder a las posiciones del *array* bidimensional (dos posiciones en nuestro ejemplo, que representan mañana y tarde) y el segundo para las del *array* unidimensional contenido en la posición anterior (tres y cuatro posiciones por cada posición del bidimensional). Vamos a ver esto en forma de matriz, que seguro le será más fácil de visualizar.

Tabla 7.8. Array bidimensional aula_1 como una matriz.

Turno	*Asignatura1*	*Asignatura2*	*Asignatura3*	*Asignatura4*
Mañana	Física	Química	Matemáticas	Biología
Tarde	Historia	Latín	Filosofía	-

Tabla 7.9. Array bidimensional aula_2 como una matriz.

Turno	*Asignatura1*	*Asignatura2*	*Asignatura3*	*Asignatura4*
Mañana	Historia	Latín	Filosofía	-
Tarde	Física	Química	Matemáticas	Biología

Con las tablas 7.8 y 7.9 como apoyo, y suponiendo que tenemos ya definidos todos los *arrays* necesarios en nuestro código, vamos a hacer las siguientes operaciones:

1. Las asignaturas que se imparten por la mañana en el aula 1.

```
<SCRIPT TYPE="text/javascript">
    // Mostramos los valores
    alert(aula_1[0]);
</SCRIPT>
```

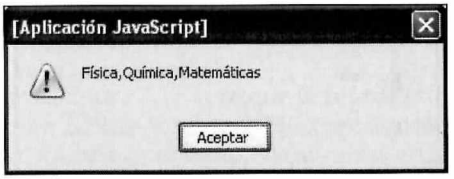


Figura 7.13. Asignaturas de aula_1 por las mañanas.

2. Las asignaturas que se imparten por la mañana y por la tarde en el aula 2.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert("Mañanas: " + aula_2[0] + "\n" +
   "Tardes: " + aula_2[1]);
</SCRIPT>
```

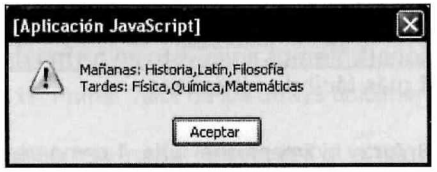


Figura 7.14. Asignaturas de aula_2 por las mañanas y tardes.

3. La primera asignatura que se imparte por la mañana en el aula 1.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(aula_1[0][0]);
</SCRIPT>
```

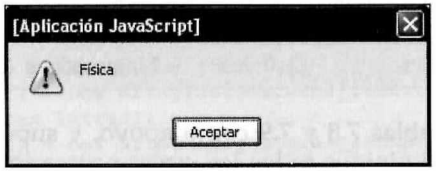


Figura 7.15. Primera asignatura de aula_1 por las mañanas.

4. La segunda asignatura que se imparte por la mañana en el aula 2.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(aula_2[0][1]);
</SCRIPT>
```

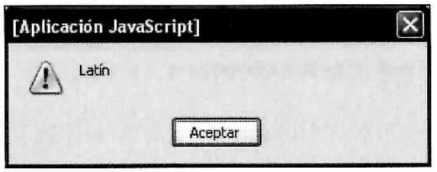


Figura 7.16. Segunda asignatura de aula_2 por las mañanas.

5. Número de asignaturas que se imparten por la tarde en el aula 2.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(aula_2[1].length); // Mostrará 4
asignatuas
</SCRIPT>
```

Bueno, ¿qué tal se ha dado? Sé que es un poco duro al principio pero poco a poco estas operaciones le parecerán más sencillas de entender.

Ahora vamos con un último paso: *arrays* tridimensionales (suena a título de película, ¿verdad?). Para hacer esto vamos a suponer que cada una de nuestras aulas están en un piso de un edificio. Para ello vamos a definir un nuevo *array* con dos posiciones representando los pisos, por lo que cada posición tendrá un aula como valor.

```
<SCRIPT TYPE="text/javascript">
   // Arrays unidimensionales
   var asignaturas_letras = new Array("Historia",
   "Latín", "Filosofía");
   var asignaturas_ciencias = new Array("Física",
   "Química", "Matemáticas", "Biología");
   // Arrays bidimensionales
   var aula_1 = new Array(asignaturas_ciencias,
   asignaturas_letras);
   var aula_2 = new Array(asignaturas_letras,
   asignaturas_ciencias);
   // Array tridimensional
   var plantas = new Array(aula_1, aula_2);
   // Mostramos los arrays tridimensionales
   alert(plantas);
</SCRIPT>
```

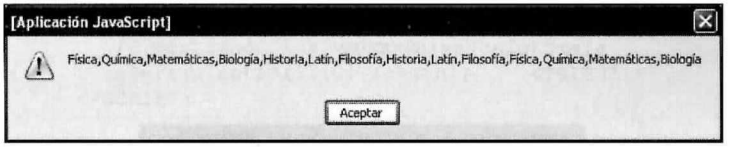


Figura 7.17. Arrays tridimensionales.

Si se fija en la figura 7.17 verá que la representación de este *array* es la unión de los dos *arrays* bidimensionales separados por una coma. Vamos a ver también este *array* en forma de matriz para que le sirva de ayuda en los ejemplos posteriores:

Tabla 7.10. Array tridimensional plantas como una matriz.

Planta	*Turno mañanas*	*Turno tardes*
Primera	Física, Química, Biología, Matemáticas	Historia, Latín, Filosofía
Segunda	Historia, Latín, Filosofía	Física, Química, Matemáticas, Biología

Entonces, para acceder ahora a un valor del *array* bidimensional (aulas) debemos escribir una pareja de corchetes (`[] []`) pero para alcanzar el valor de un *array* unidimensional (asignaturas) hay que escribir tres parejas (`[] [] []`). Veamos esto con ejemplos:

1. Las asignaturas que se imparten en la primera planta.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(plantas[0]);
</SCRIPT>
```

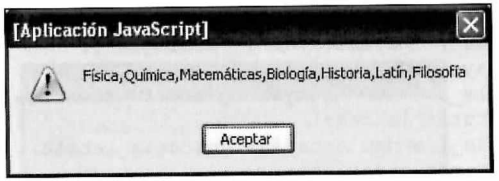


Figura 7.18. Asignaturas de la planta 1.

2. Las asignaturas que se imparten por la mañana en la primera planta.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(plantas[0][0]);
</SCRIPT>
```

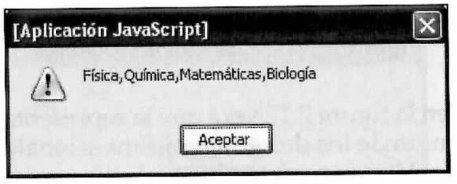


Figura 7.19. Asignaturas de la planta 1 por la mañana.

3. La primera asignatura que se imparte por la mañana en la primera planta.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(plantas[0][0][0]);
</SCRIPT>
```

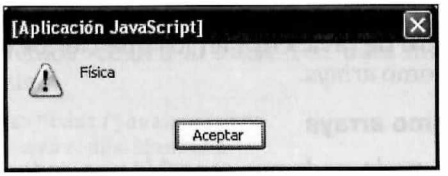


Figura 7.20. Primera asignatura de la planta 1 por la mañana.

4. La tercera asignatura que se imparte por la tarde en la primera planta.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(plantas[0][1][2]);
</SCRIPT>
```

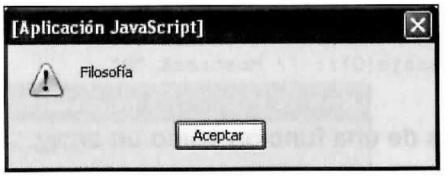


Figura 7.21. Tercera asignatura de la planta 1 por la tarde.

5. Número de asignaturas que se imparten por la mañana en la segunda planta.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos los valores
   alert(plantas[1][0].length);
</SCRIPT>
```

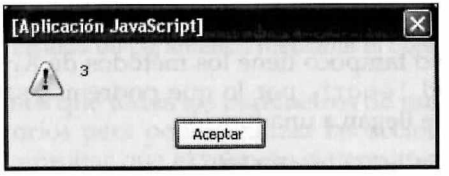


Figura 7.22. Cantidad de asignaturas de la planta 2 por la mañana.

Espero que le haya quedado claro la forma de acceder a las distintas dimensiones en cada uno de los casos. De todas formas le recomiendo que practique con sus propios ejemplos para coger soltura en el manejo de *arrays*.

7.2.5. Arrays en elementos de JavaScript

Me gustaría comentarle brevemente un par de casos especiales dentro de JavaScript en los que ciertos elementos se comportan como *arrays*.

Cadenas como arrays

En cierto modo podemos concebir una cadena como una lista de letras o caracteres, pudiendo obtener (pero no modificar) cada uno de ellos de forma independiente, aunque no tendremos disponibles los métodos asociados al objeto Array puesto que se trata de una instancia del objeto `String`.

```
<SCRIPT TYPE="text/javascript">
   // Declaramos una cadena
   var mensaje = "Bienvenido";
   // Acceso a caracteres como un array
   alert(mensaje[0]); // Mostrará "B"
   alert(mensaje[4]); // Mostrará "v"
   mensaje[0] = "b"; // No tendrá efecto
   alert(mensaje[0]); // Mostrará "B"
</SCRIPT>
```

Argumentos de una función como un array

¿Creía que se me había olvidado? Soy despistado, pero cumplo con mi palabra. En el capítulo referente a las funciones se dijo que se podría controlar que los parámetros fuesen obligatorios u opcionales, gracias al objeto `arguments` que nos proporciona JavaScript, el cual almacena en cada posición el valor del parámetro situado en la misma posición. Además, sólo estará disponible dentro de cada función.

No importa si la función no tiene definido ningún parámetro o tiene un número menor de los que se le hayan puesto. El objeto `arguments` albergará igualmente todos los parámetros que se indiquen en la llamada a la función.

Este objeto tampoco tiene los métodos de `Array`, pero sí su propiedad `length`, por lo que podremos saber cuántos parámetros le llegan a una función.

```
<SCRIPT TYPE="text/javascript">
   // Declaramos una función
   function mostrarNumeroParametros() {
      alert(arguments.length);
   }
   // Llamadas con distinto número de parámetros
   mostrarNumeroParametros(); // Mostrará 0
   mostrarNumeroParametros(1); // Mostrará 1
   mostrarNumeroParametros("a", "b", false); // Mostrará 3
</SCRIPT>
```

Para ver todos los argumentos que se han pasado a la función podemos recurrir al bucle `for` para mostrarlos más cómodamente.

```
<SCRIPT TYPE="text/javascript">
   // Declaramos una función
   function mostrarParametros() {
      var valores = "Parámetros:\n";
      for(var i=0; i<arguments.length; i++) {
         valores += "\t- Parámetro " + i + ": " +
         arguments[i] + "\n";
      }
      alert(valores);
   }
   // Llamadas con distinto número de parámetros
   mostrarParametros();
   mostrarParametros(1);
   mostrarParametros("a", "b", false);
</SCRIPT>
```

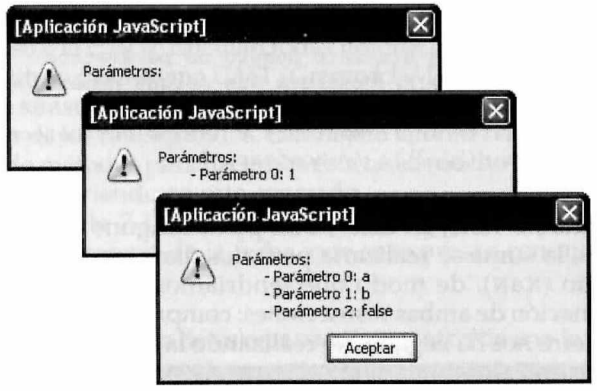


Figura 7.23. Listado de parámetros mediante el objeto arguments.

Si queremos que todos los parámetros de nuestra función sean obligatorios para poder realizar las acciones, entonces debemos comprobar que el número de argumentos que nos llega es el mismo que esperamos.

```
<SCRIPT TYPE="text/javascript">
   // Función con parámetros obligatorios
   function sumar(numero1, numero2) {
      if (arguments.length == 2) {
         return numero1 + numero2;
      } else {
         return "No hay dos números";
      }
   }
   // Llamadas con distinto número de parámetros
   alert(sumar()); // Mostrará "No hay dos números"
   alert(sumar(5)); // Mostrará "No hay dos números"
   alert(sumar(5, 2)); // Mostrará 7
</SCRIPT>
```

Por otro lado, para hacer que un parámetro sea opcional, y poder darle un valor por defecto si queremos, tendremos que comprobar el valor de la posición correspondiente en el objeto `arguments`. Si tiene un valor `undefined` significa que no lo hemos recibido. En el ejemplo, haremos opcional el segundo número de la suma.

```
<SCRIPT TYPE="text/javascript">
   // Función con parámetros opcionales
   function sumar(numero1, numero2) {
      if (arguments[1] == undefined) {
         numero2 = 3;
      }
      return numero1 + numero2;
   }
   // Llamadas con distinto número de parámetros
   alert(sumar()); // Mostrará NaN porque numero1 no es
                   // obligatorio
   alert(sumar(5)); // Mostrará 8 (valor por defecto)
   alert(sumar(5, 2)); // Mostrará 7
</SCRIPT>
```

Como ha visto, en caso de no pasar ninguno de los parámetros, la suma se realizaría pero nos daría un resultado no deseado (`NaN`), de modo que tendríamos que realizar una combinación de ambas restricciones: comprobar que el primer parámetro nos ha llegado, no realizando la suma si no es así, y asignar un valor por defecto al segundo si tampoco lo hemos recibido.

```
<SCRIPT TYPE="text/javascript">
   // Función con parámetros opcionales
   function sumar(numero1, numero2) {
      if (arguments[0] == undefined) {
         return "Falta el primer número";
      }
      if (arguments[1] == undefined) {
         numero2 = 3;
      }
      return numero1 + numero2;
   }
   // Llamadas con distinto número de parámetros
   alert(sumar()); // Mostrará "Falta el primer número"
   alert(sumar(5)); // Mostrará 8 (valor por defecto)
   alert(sumar(5, 2)); // Mostrará 7
</SCRIPT>
```

7.3. Objeto Date

Dado que JavaScript no posee un tipo de dato específico para manejar fechas, como ocurre en otros lenguajes, nos ofrece como alternativa este objeto que nos permite manipular información acerca del año, mes, día, hora, minutos, segundos y milisegundos.

La particularidad del objeto `Date` es la forma que tiene de almacenar la fecha, ya que la representa como los milisegundos que han pasado desde una fecha base que es el 1 de enero de 1970 a las 00:00:00. Aún así, el rango de fechas que puede manejar es de -100000000 (-100 millones) hasta +100000000 (100 millones) de días relativos al 1 de enero de 1970 UTC (*Universal Coordinated Time*). El UTC, o tiempo civil, es la zona horaria de referencia respecto a la cual se calculan todas las otras zonas del mundo y es sustituto del sistema GMT (*Greenwich Mean Time*).

No se asuste puesto que no tendrá que descifrar cada fecha a partir de los milisegundos. Este objeto también viene acompañado de métodos para hacernos esta tarea mucho más sencilla. Los iremos viendo en este apartado.

En la tabla 7.11 se listan los posibles valores que puede tomar cada campo de la fecha, que se expresan con números enteros.

Tabla 7.11. Rango de valores para los campos de una fecha.

Campo	*Rango de valores*
Milisegundos	0-999
Segundos y minutos	0 - 59
Horas	0 - 23
Día de la semana	0 (domingo) - 6 (sábado)

Campo	Rango de valores
Día del mes	1 - 31
Mes	0 (enero) - 11 (diciembre)
Año	Desde 1900

7.3.1. Constructor

La sintaxis para crear una instancia de este objeto es bastante simple:

```
<SCRIPT TYPE="text/javascript">
   // Objeto Date
   var nombre_variable = new Date(fecha);
</SCRIPT>
```

El parámetro `fecha` puede ser cualquiera de estas expresiones:

- Vacío: Si no se pasa este parámetro, entonces se crea una instancia con la fecha y hora actuales. Si le suena de otros lenguajes, sería el equivalente a la función `now()`.
- Una cantidad de milisegundos: Se crea una fecha sumándolos a la fecha base.
- "Mes_en_letras día, año horas:minutos:segundos": El mes debe estar escrito en inglés. Si se omite la parte correspondiente a las horas, minutos o segundos, tomarán valor 00 por defecto.
- "Año, mes, día": Estos valores deben ser numéricos enteros.
- "Año, mes, día, hora, minuto, segundos": Exactamente igual que el anterior pero ampliado para incluir la hora.

Tabla 7.12. Ejemplos del constructor Date.

Constructor	Resultado
new Date() La fecha y hora actuales.	
new Date(86400000)	2-enero-1970 a las 00h 00min 00seg (milisegundos de un día)
new Date("August 15, 2008")	15 de agosto de 2008 a las 00h 00min 00seg.
new Date("August 15, 2008 13:15:34")	15 de agosto de 2008 a las 13h 15min 34seg.

Constructor	Resultado
new Date("Agosto 15, 2008")	Fecha no válida (mes en español).
new Date(2008, 7, 15)	15 de agosto de 2008.
new Date(2008, 7, 15, 13, 15, 34)	15 de agosto de 2008 a las 13h 15min 34seg.

7.3.2. Propiedades y métodos

Este objeto no tiene ninguna propiedad por lo que pasaremos directamente a ver sus métodos, que los agruparemos en tres tipos de funcionalidad: obtención, asignación y transformación.

Métodos de obtención

- `getTime()`: Con esto obtenemos la fecha tal y como la guarda el objeto `Date`, es decir, expresada como el número de milisegundos que han pasado desde la fecha base.
- `getMilliseconds()`, `getSeconds()`, `getMinutes()`, `getHours()`: Devuelven el campo correspondiente a los milisegundos, segundos, minutos u hora, respectivamente, de la instancia actual.
- `getDate()`, `getDay()`, `getMonth()`: Nos permiten obtener el día del mes, el día de la semana y el mes de la instancia actual, representados siempre con un número entero de acuerdo a lo visto en la tabla 7.11.
- `getYear()`, `getFullYear()`: Devuelven el año de la instancia actual. `getFullYear` siempre lo hará como un número de cuatro cifras mientras que `getYear`, que caerá en desuso, se comporta de diferente manera según el navegador (se utilizan como ejemplo dos de los más utilizados):
  - Tanto en Internet Explorer como FireFox, si el año está comprendido entre 1900 y 1999, lo devolverá en forma de dos dígitos (00-99).
  - En Internet Explorer, obtendremos un número de cuatro cifras en el resto de casos (1890 ó 2012).
  - En FireFox, si el año es anterior a 1900 nos dará los años de diferencia como un número con signo negativo (-10 equivale a 1890).

- En FireFox, los años a partir de 2000 nos dará un número a partir de 100 (112 equivale a 2012).

- `getUTCMilliseconds()`, `getUTCSeconds()`, `getUTCMinutes()`, `getUTCHours()`, `getUTCDate()`, `getUTCDay()`, `getUTCMonth()`, `getUTCFullYear()`: Funcionan exactamente igual que sus análogos pero devolviendo el campo de acuerdo al UTC.
- `getTimezoneOffset()`: Devuelve la cantidad de minutos de diferencia que existen entre el GMT y la fecha de la instancia actual. Esto nos será útil para averiguar la zona horaria en la que nos encontramos (pasando esos minutos a horas). Si sale un valor negativo, -60 por ejemplo, estaremos en la zona horaria GMT+1. Sin embargo, el mismo valor pero positivo (60), indicaría que estamos en GMT-1.

Métodos de asignación

Los métodos que veremos a continuación modifican la instancia con el valor resultante de la operación, así que no devuelven ningún valor.

- `setTime(milisegundos)`: Este método crea una fecha añadiendo o restando los milisegundos indicados a la fecha base.
- `setMilliseconds(mseg)`, `setSeconds(seg, mseg)`, `setMinutes(min, seg, mseg)`, `setHours(hora, min, seg, mseg)`: Fija los campos correspondientes a los milisegundos (mseg), segundos (seg), minutos (min) y hora. En todos estos métodos el primer parámetro es obligatorio y el resto son opcionales.
- `setDate(día)`, `setMonth(mes, día)`, `setYear año)`, `setFullYear(año, mes, día)`: Fija los campos que corresponden al día del mes, el mes y el año. Al igual que los métodos anteriores, sólo es obligatorio el primer parámetro. `setYear` sigue las mismas reglas sobre el año que explicamos con `getYear` y además está cayendo en desuso, por lo que es recomendable utilizar `setFullYear` en su lugar.
- `setUTCMilliseconds(mseg)`, `setUTCSeconds(seg, mseg)`, `setUTCMinutes(min, seg, mseg)`, `setUTCHours(hora, min, seg, mseg)`, `setUTCDate día)`, `setUTCMonth(mes, día)`, `setUTCFullYear año, mes, día)`: Realizan lo mismo que sus equivalentes pero ajustadas al UTC.

Métodos de transformación

- `parse(fecha_cadena)`: Recibe una fecha expresada como una cadena en inglés y devuelve el número de milisegundos que han pasado desde la fecha base. Si no es posible interpretar la cadena como una fecha se devuelve `NaN`. Este método sólo puede ser llamado desde el objeto `Date` y no desde sus instancias.
- `UTC(año, mes, día, hora, minuto, segundo, milisegundo)`: Toma la fecha indicada por los parámetros y halla el número de milisegundos que han pasado desde la fecha base de acuerdo al UTC. Los tres primeros parámetros son obligatorios para usar este método (`año`, `mes`, `día`) y el resto opcionales. También es un método exclusivo del objeto `Date`.
- `toDateString()`, `toLocaleDateString()`: Devuelven la fecha sin información sobre la hora, y en forma de texto. Además, `toLocaleDateString` ajusta los nombres de los campos a la configuración de idioma de la máquina local sobre la que se ejecuta el código.
- `toTimeString()`, `toLocaleTimeString()`: Similar a los anteriores pero devuelven únicamente la parte correspondiente a la hora.
- `toGMTString()`, `toUTCString()`, `toLocaleString()`: Expresan la fecha y la hora en base al GMT, UTC y configuración local respectivamente.

Vamos a ver algunos ejemplos de estos métodos, por lo que utilizaremos una instancia `hoy` que contiene la fecha 13 de enero de 2009 17:23:55 y 356 milisegundos.

Tabla 7.13. Ejemplos de los métodos de Date.

Ejemplo	*Resultado*
hoy.getTime()	1231863835256
hoy.getMinutes()	23
hoy.getDate()	13
hoy.getDay()	2 (martes)
hoy.getYear()	2009 (Internet Explorer) ó 109 (FireFox)
hoy.getFullYear()	2009

Ejemplo	Resultado
hoy.getTimezoneOffset()	-120 (GMT +2)
hoy.setTime(1500089)	variable hoy: 1 de enero de 1970 01:25:00:089
hoy.setMonth(11)	variable hoy: 1260721435256 (13 diciembre 2009 17:23:55:356)
hoy = Date.parse ("April 30, 2007")	1177884000000
hoy.toLocaleDateString()	martes, 13 de enero de 2009
hoy.toUTCString()	Tue, 13 Jan 2009 15:23:55 GMT (2h de diferencia)
Date.parse("Jan 2, 1970")	86400000 (GMT+0)

7.3.3. Trabajar con fechas

A la hora de realizar operaciones con fechas, JavaScript siempre va a operar con los valores expresados en milisegundos aunque nosotros podremos recuperar después ese valor en el formato que más nos convenga.

En nuestros cálculos usaremos normalmente alguno de los operadores aritméticos (suma y resta) y los comparadores.

Vamos a ver un ejemplo suponiendo que queremos calcular la edad de una persona a partir de su fecha de nacimiento. Para ello tendremos que calcular la diferencia que existe entre el año de su nacimiento y el actual usando el operador aritmético de la resta (-). La pega que existe al trabajar con fechas es que vamos a tener los valores expresados como milisegundos, por lo que esta resta nos dará un valor en esas unidades y deberemos expresar en años.

> **Nota**: *Para pasar una fecha en milisegundos a otra unidad de tiempo debemos tener en cuenta lo siguiente: 1 año son 365 días, 1 día son 24 horas, 1 hora son 60 minutos, 1 minuto son 60 segundos y 1 segundo 1000 milisegundos.*

```
<SCRIPT TYPE="text/javascript">
   // Definimos las fechas
   var nacimiento = new Date(1975, 8, 30);
   // 30 septiembre 1975
   var hoy = new Date(); // 18 enero 2011
   var edadMilisegundos = hoy - nacimiento;
```



```
   var edad = parseInt(edadMilisegundos / 1000 / 60 /
   60 / 24 / 365); // usamos parseInt para descartar
                   // los decimales
   // Mostramos la edad
   alert(edadMilisegundos); // Mostrará 1114045200000
                            // milisegundos
   alert(edad); // Mostrará 35
</SCRIPT>
```

Si ahora quisiéramos realizar este cálculo utilizando funciones del objeto fecha no basta con restar los años (método `getFullYear`) puesto que deberíamos tener en cuenta si ya ha cumplido años en la fecha actual o no. Para ello debemos seguir estos pasos:

1. Calculamos temporalmente la edad mediante la resta de los años de ambas fechas.
2. Si el mes actual (método `getMonth`) es posterior al de nacimiento significa que ya ha cumplido años, por lo que podríamos devolver la edad calculada en el paso anterior. En caso de que el mes actual sea menor que el de nacimiento significaría que no ha llegado su cumpleaños todavía y tendríamos que restar un año a la edad obtenida en el paso anterior y devolverla. Por último, si los meses coinciden nos quedaría por analizar la diferencia entre los días (en el paso siguiente).
3. Aquí analizaremos la diferencia entre los días correspondientes al mismo mes. Si el día actual (método `getDate`) es menor al de nacimiento, restamos un año a la edad del paso 1 y la devolvemos. Por otro lado, si el día es igual o mayor, la edad se quedaría como está y podría devolverse.

Ahora vamos a trasladar todos estos pasos a una función que usaremos después en unos ejemplos:

```
<SCRIPT TYPE="text/javascript">
   function calcularEdad(nacimiento) {
      var hoy = new Date();
      var edad = hoy.getFullYear() - nacimiento.
      getFullYear();
      // Comparamos los meses
      if (hoy.getMonth() > nacimiento.getMonth()) {
         // No hacemos nada porque ya es la correcta
      } else if (hoy.getMonth() < nacimiento.getMonth()){
         edad--; // Restamos un año porque no ha llegado
                 // el cumpleaños
```

```
    } else {
       // Comparamos los días
       if (hoy.getDate() < nacimiento.getDate()) {
          edad--; // Restamos un año porque no ha
                  // llegado el cumpleaños
       }
    }
    return edad;
 }
</SCRIPT>
```

Vamos ahora a comprobar el funcionamiento de nuestra función con un primer ejemplo en el que el cumpleaños todavía no ha llegado (fecha actual 18 de enero de 2011):

```
<SCRIPT TYPE="text/javascript">
   // Definimos las fechas
   var nacimiento = new Date(1975, 8, 30);
   // 30 septiembre 1975
   var miEdad = calcularEdad(nacimiento);
   // Mostramos la edad
   alert(miEdad); // Mostrará un 35
</SCRIPT>
```

¡Bien! Todo ha funcionado como esperábamos. Veamos qué pasa ahora si nuestro cumpleaños ya ha pasado (fecha actual 18 de enero de 2011).

```
<SCRIPT TYPE="text/javascript">
   // Definimos las fechas
   var nacimiento = new Date(1975, 0, 5); // 5 enero 1975
   var miEdad = calcularEdad(nacimiento);
   // Mostramos la edad
   alert(miEdad); // Mostrará un 36
</SCRIPT>
```

¡Estamos en racha! Ahora queda la prueba de fuego. ¿Y si el cumpleaños es hoy mismo? (fecha actual 18 de enero de 2011).

```
<SCRIPT TYPE="text/javascript">
   // Definimos las fechas
   var nacimiento = new Date(1975, 0, 18);
   // 18 enero 1975
   var miEdad = calcularEdad(nacimiento);
   // Mostramos la edad
   alert(miEdad); // Mostrará un 36
</SCRIPT>
```

Todo perfecto, así da gusto. Ahora ya sabemos manejar fechas un poquito más.

7.4. Objeto Math

Hasta ahora podíamos realizar operaciones matemáticas sencillas en JavaScript gracias a los operadores aritméticos, pero en ocasiones necesitaremos hacer cálculos más complejos. El objeto `Math` nos proporciona un buen surtido de propiedades y métodos que nos permitirán satisfacer la mayoría de nuestras necesidades dentro del código.

7.4.1. Constructor

Este objeto no tiene constructor, por lo que todas sus propiedades y métodos deben utilizarse directamente a través del objeto `Math`.

> **Truco:** *Recuerde que puede utilizar la sentencia `with (Math)` para acceder más cómodamente a los elementos de este objeto.*

7.4.2. Propiedades y métodos

Vamos a presentar ahora las propiedades de `Math`, que en realidad son un conjunto de constantes matemáticas.

- `E`: Representa la constante de Euler (2,718).
- `PI`: Nos da el número Pi (3,14159).
- `SQRT2`: Con esta obtendríamos la raíz cuadrada de 2 (1,414).
- `SQRT1_2`: La raíz cuadrada de ½ (0,707).
- `LN2`: Devuelve el logaritmo neperiano de 2 (0,693).
- `LN10`: El logaritmo neperiano de 10 (2,302).
- `LOG2E`: Devuelve el resultado del logaritmo de E en base 2 (1,442).
- `LOG10E`: Igual que el anterior, pero en base 10 (0,434).

```
<SCRIPT TYPE="text/javascript">
   // Propiedades del objeto Math
   with(Math) {
      alert("Propiedades de Math:" + "\n" +
      "\tE: " + E + "\n" +
      "\tPi: " + PI + "\n" +
      "\tRaíz cuadrada de 2: " + SQRT2 + "\n" +
      "\tRaíz cuadrada de 1/2: " + SQRT1_2 + "\n" +
      "\tLogaritmo neperiano de 2: " + LN2 + "\n" +
      "\tLogaritmo neperiano de 10: " + LN10 + "\n" +
      "\tLogaritmo de E en base 2: " + LOG2E + "\n" +
```

```
        "\tLogaritmo de E en base 10: " + LOG10E);
    }
</SCRIPT>
```

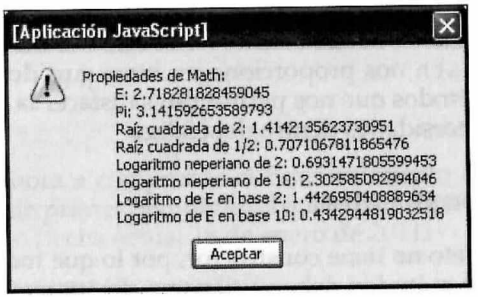


Figura 7.24. Propiedades del objeto Math.

Los métodos de `Math` nos servirán para realizar cálculos matemáticos más complejos que los que podíamos hacer hasta ahora con los operadores:

- `abs(numero)`: Nos devuelve el valor absoluto de `numero`.
- `sin(angulo)`, `cos(angulo)`, `tan(angulo)`: Funciones trigonométricas para calcular el seno, coseno y tangente. Su parámetro debe estar expresado en radianes en lugar de en grados. Tenga en cuenta que el valor devuelto por estos métodos será un valor entre -1 y 1.
- `asin(numero)`, `acos(numero)`, `atan(numero)`: Realizan los cálculos trigonométricos arcoseno, arcocoseno y arcotangente de su parámetro, el cual deberá ser un valor entre -1 y 1 (salvo para `atan`). Si el parámetro está fuera de esos valores, se devuelve `NaN`. El valor del ángulo devuelto depende del método que usemos:
  - `acos` devuelve un ángulo entre 0 y Pi radianes.
  - `asin` y `atan` nos darán un ángulo entre -Pi/2 y Pi/2.
- `atan2(posicionY, posicionX)`: Nos devuelve el ángulo que forma, respecto al eje X, el punto situado en las coordenadas indicadas como parámetros.
- `exp(numero)`, `log(numero)`: Devuelven, respectivamente, el resultado de E elevado al parámetro `numero` (E^{numero}) y el logaritmo neperiano (base E) del parámetro.
- `ceil(numero)`: Devuelve el entero inmediatamente superior de `numero` si éste tiene decimales, o directamente `numero` si no es así.

- `floor(numero)`: Igual que el anterior, pero devolviendo el entero inmediatamente inferior.
- `min(numero1, numero2)`, `max(numero1, numero2)`: Nos proporcionan el menor o mayor valor de sus dos parámetros.
- `pow(base, exponente)`: Nos calcula una potencia, es decir, devuelve el primer parámetro elevado al segundo ($base^{exponente}$).
- `random()`: Nos dará un número real aleatorio entre 0 y 1.
- `round(numero)`: Redondea el parámetro al entero más cercano. Si `numero` no tiene decimales, o el primero es menor o igual que 4, entonces el redondeo se hace hacia abajo. En otro caso, se redondea hacia el siguiente entero.
- `sqrt(numero)`: Nos calcula la raíz cuadrada del parámetro `numero`.

Tabla 7.14. Ejemplos de los métodos de Math.

Ejemplo	*Resultado*
Math.abs(1)	1
Math.abs(-3.5)	3
Math.sin(5)	-0.9589242746631385
Math.sin(Math.PI/2)	1
Math.acos(0.5)	1.0471975511965979
Math.acos(2)	NaN (parámetro fuera de rango)
Math.atan2(1, 3.2)	0.3028848683749714
Math.exp(0)	1
Math.exp(1)	2.718281828459045 (E)
Math.log(2)	0.6931471805599453
Math.log(0)	-Infinity (no se puede calcular)
Math.ceil(3)	3
Math.ceil(3.2)	4
Math.ceil(3.9)	4
Math.floor(3)	3
Math.floor(3.2)	3
Math.floor(3.9)	3
Math.min(2, 7)	2

Ejemplo	Resultado
Math.max(2, 7)	7
Math.pow(2, 3)	8
Math.random()	0.048257613864494786
Math.round(3)	3
Math.round(3.2)	3
Math.round(3.9)	4
Math.sqrt(9)	3
Math.sqrt(5)	2.23606797749979

7.5. Objeto RegExp

Este objeto se utiliza para trabajar con las expresiones regulares que definamos en nuestros *scripts*. En el apartado 2.3 del capítulo 2 vimos cómo definir nuestras propias expresiones regulares o patrones.

7.5.1. Constructor

El constructor de este objeto nos permitirá definir la expresión regular que se aplicará sobre un texto y recoger los resultados. La sintaxis sería la que sigue:

```
<SCRIPT TYPE="text/javascript">
   // Objeto RegExp
   var miPatron = new RegExp(patrón, modificadores);
</SCRIPT>
```

El patrón corresponde con un *string* que representa una expresión regular válida, sin incluir las barras (/), mientras que los modificadores son los mismos que vimos en el capítulo 2 y pueden especificarse de forma opcional. Por tanto, es posible definir un patrón y sus modificadores del mismo modo que si lo escribiésemos como una cadena delimitada por barras (/), siendo ambas sentencias correctas y equivalentes.

> **Advertencia**: *Si utiliza la barra invertida (\ ) como parte del patrón, recuerde que al tratarse de una cadena debe utilizar el carácter de escape. Por tanto el carácter especial* `\d` *se deberá escribir como* `\\d` *y la propia barra invertida como* `\\\` *(quedando* `\\` *después de procesar la cadena).*

Veamos algunos ejemplos de patrones:

1. Patrón que nos encuentre las coincidencias con "hol" seguido de una o más aes.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("hola+");
</SCRIPT>
```

2. Patrón que nos encuentre las coincidencias con "hol" seguido de una o más aes y sin distinción de mayúsculas y minúsculas.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("hola+", "i");
</SCRIPT>
```

3. Patrón que nos encuentre las coincidencias con las líneas que empiecen por "A".

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("^A");
</SCRIPT>
```

4. Patrón que nos encuentre las coincidencias con un número de dos dígitos.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("\\d\\d");
</SCRIPT>
```

5. Patrón que nos encuentre las coincidencias con una fecha.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("\\d{2}-\\d{2}-\\d{4}");
</SCRIPT>
```

7.5.2. Propiedades y métodos

Las propiedades de este objeto se centran principalmente en la activación (con valor `true`) o desactivación (con valor `false`) de algunos modificadores del patrón que se especificó en el constructor, aunque también hay otros con un fin diferente.

- `global`: Sirve para fijar el modificador "g", para forzar que se sigan buscando coincidencias después de encontrar la primera.

- `ignoreCase`: Fija el modificador "i", que elimina la distinción entre mayúsculas y minúsculas.
- `multiline`: Activa el modificador "m", para permitirnos usar varios caracteres de posición (^ y $) en el patrón que definamos.
- `lastIndex`: Representa la posición (índice) dentro del texto donde se empezará a buscar la próxima coincidencia.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("script", "g");
   // Mostramos los modificadores
   alert("global: " + miPatron.global + "\n" +
   "ignoreCase: " + miPatron.ignoreCase + "\n" +
   "multiline: " + miPatron.multiline);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("script", "gm");
   // Mostramos los modificadores
   alert("global: " + miPatron.global + "\n" +
   "ignoreCase: " + miPatron.ignoreCase + "\n" +
   "multiline: " + miPatron.multiline);
</SCRIPT>
```

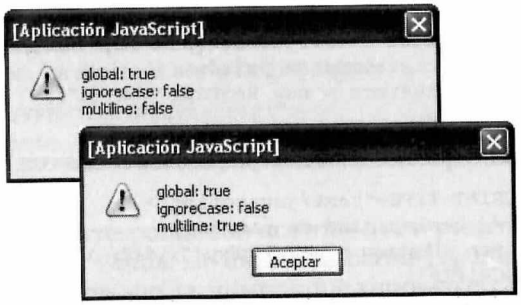


Figura 7.25. Propiedades del objeto RegExp.

Las siguientes propiedades sólo están accesibles desde el objeto `RegExp`, aunque hace referencia a valores de la instancia.

- `input`: Nos permite fijar u obtener el texto sobre el que se aplica el patrón.
- `lastMatch`: Nos devuelve la última cadena encontrada mediante el patrón.
- `lastParen`: Nos devuelve la última coincidencia que está encerrada entre paréntesis.

- `leftContext`, `rightContext`: Nos permiten conocer los caracteres que están a la izquierda o derecha de la última coincidencia.

Por otro lado, los métodos que contiene `RegExp` nos permitirán aplicar el patrón con distintas finalidades.

- `compile(patron, modificadores)`: Nos permite fijar un nuevo patrón en una instancia de `RegExp` y opcionalmente también sus modificadores. Si sólo se especifica el patrón, se pierden los modificadores que tuviera.
- `exec(texto)`: Aplica el patrón sobre el texto pasado como parámetro y devuelve la cadena encontrada o un valor `null` si no hubo coincidencias. Si se omite el parámetro se tomará como texto el especificado a través de la propiedad `input` del objeto `RegExp`.
- `test(texto)`: Aplica el patrón sobre el parámetro `texto` pero únicamente nos dirá si se encontraron coincidencias (valor `true`) o no (valor `false`). Este método es muy rápido y se recomienda usarlo cuando no necesitemos los valores encontrados. Si se omite el parámetro, ocurre lo mismo que con el método `exec`.

Veamos algunos ejemplos utilizando `/el/` como patrón y "el pájaro saltó del nido" como texto donde aplicarlo.

Tabla 7.15. Ejemplos de los métodos de RegExp.

Ejemplo	*Resultado*
miPatron.compile("del")	Se cambia el patrón a aplicar de /el/ a /del/ sin modificadores.
miPatron.compile("del", "i")	Se cambia el patrón a aplicar de /el/ a /del/ y se activa el modificador "i".
miPatron.exec(texto)	Devuelve "el" ya que se ha encontrado coincidencia.
miPatron.test(texto)	Devuelve true ya que se ha encontrado coincidencia.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de patrón
   var miPatron = new RegExp("el", "g");
   // Definimos el texto donde aplicar el patrón
```

```
    RegExp.input = "el pájaro saltó del nido";
    // Aplicamos el patrón
    miPatron.exec();
    // Mostramos la última cadena encontrada ("el")
    alert(RegExp.lastMatch);
<SCRIPT>
```

Como ya se dijo en su momento, algunos métodos del objeto `String` pueden funcionar también con expresiones regulares escritas como una cadena. La forma de usarlos sería la siguiente:

```
miTexto.nombre_metodo(/patrón/)
```

En concreto, los métodos `search(patron)`, `match(patron)`, `replace(patron, reemplazo)` y `split(patron, limite)` funcionan exactamente igual recibiendo una cadena o patrón como parámetro. En la tabla 7.16 puede ver algunos ejemplos con "el pájaro saltó del nido" como valor de `miTexto`.

Tabla 7.16. Ejemplos de los métodos de String con patrones.

Ejemplo	*Resultado*
miTexto.search(/el/)	Nos devuelve la posición 0, que es donde está la primera ocurrencia.
miTexto.seach(/la/)	Nos devuelve -1, ya que no hay coincidencias.
miTexto.match(/el/)	Nos devuelve un array con un valor (el primer "el").
miTexto.match(/el/g)	Nos devuelve un array con dos valores (los correspondientes a "el" y "del").
miTexto.replace(/el/, "El")	Nos devuelve el texto modificando "el" por "El".
miTexto.replace (/el/g, "El")	Nos devuelve el texto modificando "el" por "El" y "del" por "dEl".
miTexto.split(/del/)	Obtenemos un array con los valores "el pájaro saltó" y " nido"
miTexto.split(/del/, 1)	Obtenemos un array con el valor "el pájaro saltó"
miTexto.split(/a/)	Obtenemos un array con los valores "el páj", "ro s" y "ltó del nido"
miTexto.split(/a/, 2)	Obtenemos un array con los valores "el páj" y "ro s"

8

Objetos del navegador (DOM)

Aunque ya comentamos algo en la introducción del libro, no está mal refrescar un poco qué es esto del DOM.

Cuando creamos una página Web nos limitamos a poner etiquetas que delimitan cada uno de los elementos que tendrá la página: cabecera, cuerpo, párrafo, tabla, etc. Sin embargo, el navegador debe "pintar" esa página para que el usuario pueda conocer su contenido, interpretando todas las etiquetas y creando a la vez una jerarquía de objetos que representan los elementos que la componen. Esta jerarquía es el Modelo de Objetos del Documento, más conocido como DOM (*Document Object Model*).

Esto nos será muy útil como programadores en JavaScript ya que gracias al DOM podremos acceder a cada elemento de la página y obtener o modificar sus propiedades a nuestro antojo.

Una vez que conozca la jerarquía de objetos, puede consultar un gráfico resumen en el apéndice D, al final de esta guía.

8.1. Compatibilidad entre navegadores

A pesar de que el DOM sigue unos estándares definidos por el W3C (*World Wide Web Consortium* `http://www.w3.org`), sus objetos no siempre están definidos exactamente igual de un navegador a otro o incluso entre versiones del mismo navegador. Se podría decir que aprovechan para implementar su propia interpretación del DOM llegando hasta el extremo de añadir nuevos objetos que, obviamente, sólo son reconocidos por ese navegador concreto. Por tanto, será tarea del progra-

mador tener que salvar esas posibles incompatibilidades o implementaciones distintas para asegurar que el *script* se ejecutará correctamente en el mayor número posible de navegadores. Lamentablemente, todos estos esfuerzos no harán más que complicar y "ensuciar" nuestro código.

Pero veamos el lado bueno. No siempre se utilizan esas propiedades u objetos extra y, actualmente, la mayor parte del DOM es común a todos los navegadores. Sólo necesitará tener algo de cuidado cuando esté programando en DHTML (*Dynamic HTML*) ya que hace un uso intensivo del DOM.

En los apartados siguientes veremos los objetos principales que forman el DOM. Si quiere conocer el detalle de los objetos de más bajo nivel (capas, textos, tablas, etc.) podrá encontrarlos en cualquier libro relacionado con DHTML.

8.2. Objeto window

Estamos ante el objeto estrella del DOM, puesto que es el más alto dentro de toda la jerarquía y el encargado de representar la ventana del navegador donde se está visualizando la página. Si utilizamos un navegador que maneje pestañas (como FireFox o Internet Explorer 8), cada una de ellas tendrá su propio objeto `window` ya que en definitiva se trata de páginas distintas.

Para obtener sus colecciones de objetos, propiedades o métodos no es necesario hacer referencia siempre a `window` ya que JavaScript nos permite esta pequeña comodidad al tratarse del más alto de la jerarquía.

8.2.1. Colecciones de objetos

Como veremos en otros objetos de JavaScript, `window` contiene a su vez otros objetos.

- `frames`: Se trata de un *array* con la colección de *frames* o marcos (etiquetas `<FRAME>` o `<IFRAME>`) que contenga la página. Podemos acceder a ellos a través de su posición en el *array* o su nombre (atributo `NAME`).

```
var miFrame = window.frames[0];
var miFrame = window.frames["menu"];
```

En realidad un *frame* es un objeto `window`, por lo que tendrá las mismas propiedades y métodos que él.

8.2.2. Propiedades

A continuación veremos la mayoría de las propiedades que tenemos a mano con el objeto `window`, que no son pocas:

- `length`: fija o devuelve el número de marcos de la ventana actual.
- `name`: fija o devuelve el nombre de la ventana que nos servirá para referirnos a ella en el código. No lo confunda con el título de la página que aparece en la parte superior izquierda.
- `menubar`: Nos devuelve un objeto que representa la barra de menú del navegador (ver figura 8.1).
- `toolbar`: Nos devuelve un objeto que representa la barra de herramientas del navegador (ver figura 8.1).
- `statusbar`: Igual que el anterior pero con la barra de estado (ver figura 8.1).
- `defaultStatus`: Nos permite fijar u obtener el mensaje por defecto que se muestra en la barra de estado. Esta propiedad no es compatible en FireFox por defecto.
- `status`: Nos permite modificar el texto de la barra de estado. Esta propiedad no es compatible en FireFox por defecto.
- `scrollbars`: Nos devuelve el objeto que simboliza las barras de desplazamiento (ver figura 8.1).

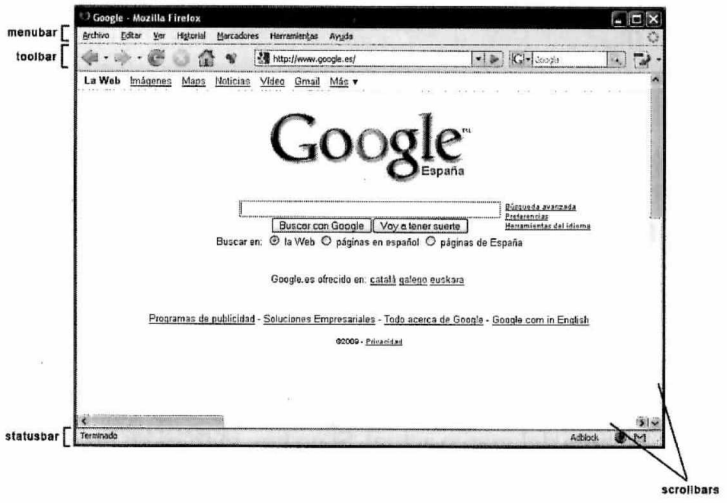


Figura 8.1. Barras de un navegador.

- `location`, `history`, `document`: Nos devuelven los objetos `location`, `history` y `document`, los cuales veremos más adelante en este capítulo de forma individual.
- `outerHeight`, `outerWidth`: Establece o devuelve el tamaño, en pixels, del espacio de toda la ventana en vertical u horizontal, respectivamente. Este valor incluye las barras de desplazamiento, de herramientas, etc (ver figura 8.2).
- `innerHeight`, `innerWidth`: Obtiene la altura o anchura, en pixels, del espacio donde se visualiza propiamente la página (ver figura 8.2). Este valor tiene en cuenta las barras de desplazamiento, si hubiera, pero excluye todas las demás (menú, herramientas, estado, etc.).

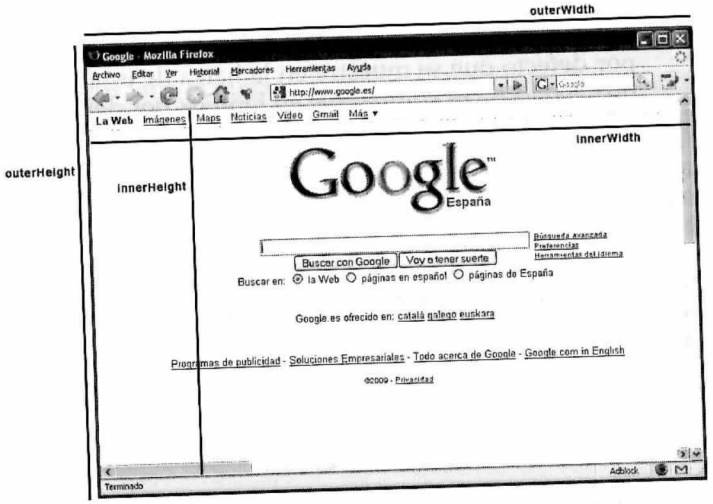


Figura 8.2. Propiedades de espacio del objeto window.

- `parent`: Se refiere a la ventana donde está situado el marco (`<FRAMESET>`) en el que estamos trabajando. En caso de no haber marcos en la página, equivale a la propia ventana.
- `top`: Devuelve la ventana que se encuentra un nivel por encima del marco actual. Si no hay marcos en la página, se devuelve la propia ventana.
- `self`: Se refiere a la ventana actual (es similar a la sentencia `this` que vimos con los objetos).

- `opener`: Hace referencia a la ventana que abrió la actual (ventana emergente o *pop-up*). Si la ventana no fue abierta por otra, obtendremos un valor `undefined`.
- `closed`: Nos indica si la ventana está cerrada o no a través de un valor booleano.

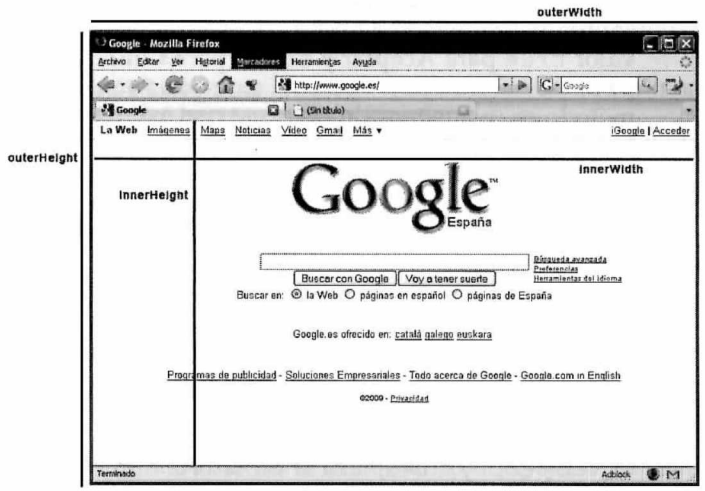


Figura 8.3. Propiedades de espacio del objeto window con pestañas.

> **Truco:** *Para hacer que las propiedades `defaultStatus` y `status` funcionen con FireFox, se debe modificar su configuración accediendo al URL about:config y cambiando el valor de la clave `dom.disable_window_status_change` a falso.*

El siguiente ejemplo nos permite cambiar el tamaño de la ventana actual ajustando el número de pixels por el valor que queramos:

```
<SCRIPT TYPE="text/javascript">
   // Redimensionamos la ventana
   window.outerWidth = 640;
   window.outerHeight = 480;
</SCRIPT>
```

8.2.3. Métodos

Entre los métodos del objeto `window` encontraremos muchas funciones de utilidad para aportar más dinamismo a nuestras páginas.

- `alert(mensaje)`: ¿Le suena;)? ¡Ahora sabe de dónde viene! Por si no lo recuerda, muestra un mensaje en un cuadro de diálogo con el botón **Aceptar**.
- `confirm(mensaje)`: Muestra un mensaje en un cuadro de diálogo pero, a diferencia de `alert`, muestra los botones **Aceptar** y **Cancelar**. Además, nos permite saber sobre qué botón se ha hecho clic, ya que devuelve un valor `true` para **Aceptar** y `false` para **Cancelar**.
- `prompt(mensaje, valor_por_defecto)`: Este método es una extensión de `confirm`, ya que incluye una caja de texto para recoger un valor escrito por el usuario, el cual además es devuelto. Opcionalmente podemos indicar un valor que será mostrado por defecto en la caja de texto cuando se muestre el mensaje. Si el usuario hace clic sobre **Cancelar** obtendremos un valor `null`.
- `focus(),blur()`: Establece o retira el foco de un objeto. Se dice que un objeto tiene el foco cuando lo estamos utilizando (por ejemplo, escribiendo en un cuadro de texto) y lo pierde cuando dejamos de usarlo. Con estos métodos podemos forzar estas situaciones.
- `moveBy(x, y)`: Desplaza la ventana el número de píxeles indicados. La `x` indica el desplazamiento horizontal y la `y` el vertical.
- `moveTo(x, y)`: Mueve la ventana a una posición concreta.
- `resizeBy(x, y)`: Redimensiona la ventana tantos píxeles como indiquen los parámetros. La `x` se refiere a la anchura y la `y` a la altura.
- `resizeTo(x, y)`: Establece una anchura y altura concretas a la ventana.
- `scrollBy(x, y)`: Realiza un desplazamiento horizontal y/o vertical de tantos píxeles como marquen los parámetros.
- `scrollTo(x, y)`: Realiza un desplazamiento horizontal y/o vertical hasta una posición concreta.
- `open(URL, nombre, parámetros)`: Crea una nueva ventana (normalmente un *pop-up*) en la que se carga el URL especificado. Esta ventana recibe también un nombre para identificarla y, opcionalmente, una serie de parámetros que se pasan como una cadena de pares "propiedad=valor" separados por comas, siendo `yes` o `1` valores positivos y `no` o `0` los negativos. Pasemos a ver cuáles podemos incluir:

  - `toolbar`: Nos permite indicar si la ventana tendrá barra de herramientas o no.
  - `menubar`: Muestra u oculta la barra de menús de la nueva ventana.
  - `location`: Nos permite mostrar u ocultar la barra de direcciones en la nueva ventana.
  - `directories`: Nos deja decidir si la ventana tendrá botones de dirección o no.
  - `scrollbars`: Nos deja indicar si la nueva ventana tendrá barras de desplazamiento o no.
  - `status`: Nos permite controlar la visibilidad de la barra de estado.
  - `resizable`: Permite establecer si la nueva ventana podrá ser cambiada de tamaño (con el ratón) o no.
  - `fullscreen`: Nos deja indicar si la ventana se verá a pantalla completa o no.
  - `width, height`: Con esto establecemos el ancho o alto que tendrá la ventana, en pixels.
  - `top, left`: Podemos indicar la distancia, en pixels, a la que estará la ventana respecto al lado superior o izquierdo de la pantalla.

  Este método devuelve un identificador para la ventana que hemos abierto (para poder hacer referencia a ella después) o un valor `null` si la ventana no pudo abrirse (por ejemplo a causa de un bloqueo de ventanas emergentes). Se puede decir que el método `open` crea una instancia del objeto `window`.
- `close()`: Cierra la ventana actual.
- `print()`: Manda imprimir la página que estemos viendo.
- `setInterval(expresion, milisegundos)`: Evalúa una expresión continuamente después de que hayan pasado el número de milisegundos especificados. Este método nos devuelve un identificador, que podrá ser utilizado para detener su ejecución.
- `setTimeout(expresion, milisegundos)`: Evalúa una expresión una única vez después de esperar a que pasen el número de milisegundos indicados. Este método también nos devuelve un identificador para poder cancelar esta ejecución antes de que ocurra.
- `clearInterval(identificador),clearTimeout identificador)`: Cancelan respectivamente la ejecución de `setInterval` y `setTimeout` a través del identificador pasado como parámetro.

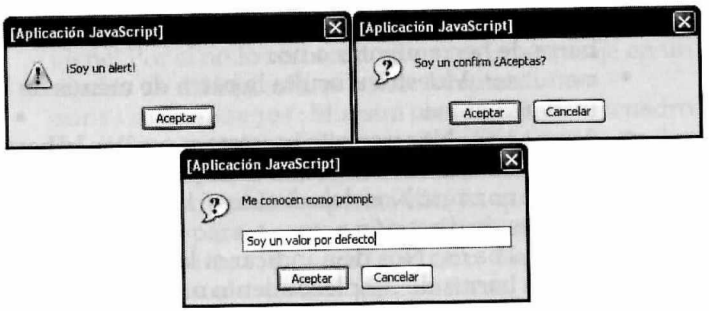


Figura 8.4. Ejemplos de los métodos alert, confirm y prompt.

A continuación veremos algunos *scripts* que utilizan los métodos del objeto `window` referentes a los cuadros de diálogo.

```
<SCRIPT TYPE="text/javascript">
   // Capturamos el botón
   var boton = confirm("¿Quieres continuar?");
   if (boton) {
      alert("¡Adelante!");
   } else {
      alert("Hasta pronto");
   }
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Capturamos el texto
   var texto = prompt("¿Qué navegador utilizas?");
   if (texto == null) {
      alert("Vale, sólo era una pregunta…");
   } else {
      alert("Utilizas: " + texto);
   }
</SCRIPT>
```

Con los siguientes ejemplos podremos crear nuevas ventanas con o sin parámetros adicionales.

```
<SCRIPT TYPE="text/javascript">
   // Ventana nueva con opciones por defecto
   var ventana = window.open("mi_pagina.html",
   "ventana1");
   if (ventana == null) {
      alert("No se pudo abrir la ventana");
   }
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Ventana nueva con opciones personalizadas
   window.open("mi_pagina.html", "ventana1",
```



```
   "width=500,height=300");
   window.open("mi_pagina.html", "ventana2", "width=500,
   height=300,top=5,left=5");
   window.open("mi_pagina.html", "ventana3", "toolbar=1,
   menubar=0,location=0");
</SCRIPT>
```

Ahora usaremos los métodos que evalúan una expresión pasado un plazo de tiempo.

```
<SCRIPT TYPE="text/javascript">
   // Muestra un mensaje cada 5 segundos
   setInterval("alert(\"Riiiiiinggg\")", 5000);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Muestra un mensaje a los 5 segundos
   setTimeout("alert(\"¡Se acabó el tiempo!\")", 5000);
</SCRIPT>
```

También es posible usar funciones que ya estuvieran definidas en nuestro *script*.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function saltarAlarma() {
      alert("Riiiiiinggg");
   }
   // Ejecuta una función cada 5 segundos
   setInterval("saltarAlarma()", 5000);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function mostrarMensaje() {
      alert("¡Se acabó el tiempo!");
   }
   // Ejecuta una función a los 5 segundos
   setTimeout("mostrarMensaje()", 5000);
</SCRIPT>
```

El código que sigue ejecutará la función `mostrarCuentaAtras` cada segundo hasta que la variable `segundos` valga cero, momento en el cual se cancelará la repetición de `setInterval` automáticamente.

```
<SCRIPT TYPE="text/javascript">
   var segundos = 5;
   var temporizador;
   // Declaración de función
   function mostrarCuentaAtras() {
      if (segundos > 0) {
         alert("Quedan " + segundos + " segundos para el
         lanzamiento");
```

```
        segundos--;
      } else {
        clearInterval(temporizador);
        alert("¡Ignición!");
      }
   }
   // Ejecuta una función cada segundo
   temporizador = setInterval("mostrarCuentaAtras()",
   1000);
</SCRIPT>
```

Este otro código ejecutará la función `mostrarMensaje` después de 10 segundos, a menos que pulsemos sobre el botón **Aceptar** del cuadro de diálogo que se muestra.

```
<SCRIPT TYPE="text/javascript">
   var temporizador;
   var boton;
   // Declaración de función
   function mostrarMensaje() {
      alert("¡Ya estoy aquí!");
   }
   // Ejecuta una función a los 10 segundos
   temporizador = setTimeout("mostrarMensaje()", 10000);
   // Controlamos la cancelación del timeout
   boton = confirm("¿Cancelamos la espera?");
   if (boton) {
      clearTimeout(temporizador);
   }
</SCRIPT>
```

Así mismo, también podremos usar funciones con parámetros. Para ello podemos escribir como una cadena la llamada con sus correspondientes valores.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function mostrarMensaje(mensaje) {
      alert(mensaje);
   }
   // Ejecuta una función cada 5 segundos
   setInterval("mostrarMensaje(\"Riiiiiinggg\")", 5000);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function mostrarMensaje(mensaje) {
      alert(mensaje);
   }
   // Ejecuta una función a los 5 segundos
   setTimeout("mostrarMensaje(\"Riiiiiinggg\")", 5000);
</SCRIPT>
```


Otra manera de usar funciones con parámetros, es incluir una llamada normal dentro de un sentencia `function`.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function mostrarMensaje(mensaje) {
      alert(mensaje);
   }
   // Ejecuta una función cada 5 segundos
   setInterval(function(){ mostrarMensaje("Riiiiiinggg")
}, 5000);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function mostrarMensaje(mensaje) {
      alert(mensaje);
   }
   // Ejecuta una función a los 5 segundos
   setTimeout(function(){ mostrarMensaje("Riiiiiinggg")
}, 5000);
</SCRIPT>
```

De esta forma, además podremos incluir más líneas de código que necesitemos ejecutar dentro de esas llamadas programadas. Como ejemplo, haremos dos llamadas a la función `mostrarMensaje` con dos textos distintos. El efecto es que se mostrarán ambos mensajes, uno a continuación del otro, pasados 5 segundos.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de función
   function mostrarMensaje(mensaje) {
      alert(mensaje);
   }
   // Ejecuta una función a los 5 segundos
   setTimeout(function(){ mostrarMensaje("Riiiiiinggg");
   mostrarMensaje("Y más riiiing!"); }, 5000);
</SCRIPT>
```

8.3. Objeto navigator

Con este objeto podremos acceder a información acerca del navegador que está utilizando el usuario para ver la página. Esto nos será de utilidad para tomar decisiones dentro de nuestro código ya que sabremos de antemano qué cosas podrían no funcionar en el navegador. Por ejemplo, si detectamos que el usuario está utilizando FireFox 1.0, haremos que nuestro código no ejecute el método `resizeTo` del objeto `window` ya

que no lo reconocería, o si el navegador es Opera Mini es recomendable que nuestra ventana no tenga unas dimensiones muy grandes ya que el usuario estará usando un Smartphone, con una resolución más baja que un ordenador.

8.3.1. Propiedades

Gracias a esta lista de propiedades podemos obtener bastante información sobre el navegador del usuario, aunque puede que algunos valores le sorprendan por no ajustarse a lo que esperaba:

- `appCodeName`: Devuelve el nombre del código del navegador.
- `appName`: Nos dice el nombre del navegador.
- `appMinorVersion`: Nos dice la versión de la última revisión del navegador.
- `appVersion`: Nos proporciona información acerca de la versión del navegador.
- `userAgent`: Devuelve la cabecera completa del navegador que se envía en cada petición HTTP. Esta propiedad contiene los valores de las propiedades `appCodeName` y `appVersion`.
- `browserLanguage`: Devuelve el idioma del navegador.
- `systemLanguage`: Nos dice el idioma por defecto del sistema operativo del usuario.
- `userLanguage`: Indica el idioma preferido del usuario para el sistema operativo que está utilizando.
- `language`: Nos devuelve el idioma del navegador.
- `cpuClass`: Nos ofrece información acerca del tipo de procesador que tiene el usuario.
- `platform`: Indica la plataforma del sistema operativo.
- `cookieEnabled`: Permite saber si el navegador tiene activadas las *cookies* o no. Aunque tendremos un capítulo dedicado a ello, una *cookie* es una información que se guarda en el navegador en las visitas a páginas Web.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos las propiedades
   alert("appCodeName: " + navigator.appCodeName +
   "\n" +
   "appName: " + navigator.appName  + "\n" +
   "appMinorVersion: " + navigator.appMinorVersion
   + "\n" +
   "appVersion: " + navigator.appVersion  + "\n" +
   "userAgent: " + navigator.userAgent  + "\n" +
```



```
   "browserLanguage: " + navigator.browserLanguage
   + "\n" +
   "systemLanguage: " + navigator.systemLanguage  +
   "\n" +
   "userLanguage: " + navigator.userLanguage +
   "\n" +
   "language: " + navigator.language  + "\n" +
   "cpuClass: " + navigator.cpuClass  + "\n" +
   "platform: " + navigator.platform  + "\n" +
   "cookieEnabled: " + navigator.cookieEnabled);
</SCRIPT>
```

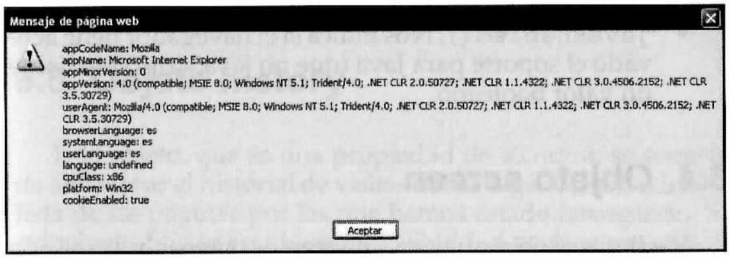


Figura 8.5. Propiedades del objeto navigator en Internet Explorer 8.

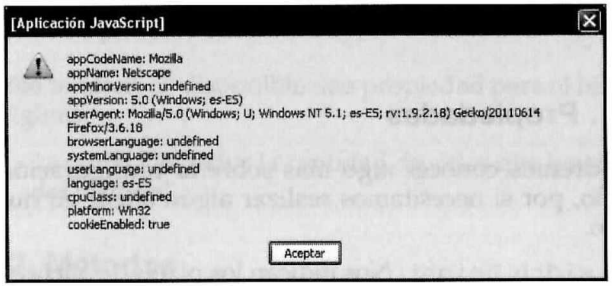


Figura 8.6. Propiedades del objeto navigator en Firefox 3.6.

```
appCodeName: Mozilla
appName: Opera
appMinorVersion:
appVersion: 9.80 (J2ME/MIDP;Opera Mini/6.24093/25.729; U; es)
userAgent: Opera/9.80 (J2ME/MIDP; Opera Mini/6.24093/25.729; U; es) Presto/2.5.25 Version/10.5454
browserLanguage: es-ES
systemLanguage: undefined
userLanguage: es-ES
language: es-ES
cpuClass: undefined
platform: Pike v7.8 release 517
cookieEnabled: true
```

Figura 8.7. Propiedades del objeto navigator en Opera Mini 6.

Nota: *No todas estas propiedades están disponibles en todos los navegadores, por lo que debemos tener esto en cuenta si vamos a utilizarlas dentro de nuestro código.*

8.3.2. Métodos

La verdad que el número de métodos de este objeto no es muy amplio ya que no podemos modificar ninguna de las características del navegador, de modo que nos tendremos que conformar con uno.

- `javaEnabled()`: Nos indica si el navegador tiene activado el soporte para Java (que no JavaScript) mediante un valor booleano.

8.4. Objeto screen

Mediante el objeto `screen` podremos obtener información acerca de la configuración de pantalla del usuario que está viendo nuestra página. Únicamente tendremos disponibles unas pocas propiedades y, desafortunadamente, ningún método.

8.4.1. Propiedades

Podremos conocer algo más sobre la configuración del usuario, por si necesitamos realizar algún ajuste en nuestro código.

- `width, height`: Nos indican los pixels de ancho y alto que tiene la resolución actual.
- `availWidth, availHeight`: Nos devuelve la anchura y altura que queda libre en el área de trabajo del escritorio, es decir, resta al ancho o alto total el espacio que ocupan las barras de herramientas que tengamos en nuestro sistema operativo.
- `colorDepth`: Nos permite conocer la profundidad de color, en bits, de la pantalla del usuario.

```
<SCRIPT TYPE="text/javascript">
   // Mostramos las propiedades
   alert("width: " + screen.width + "\n" +
   "height: " + screen.height  + "\n" +
   "availWidth: " + screen.availWidth + "\n" +
```



```
   "availHeight: " + screen.availHeight  + "\n" +
   "colorDepth: " + screen.colorDepth);
</SCRIPT>
```

Figura 8.8. Propiedades del objeto screen.

8.5. Objeto history

Este objeto, que es una propiedad de `window`, se encarga de almacenar el historial de visitas del navegador, es decir, una lista de las páginas por las que hemos estado navegando. Su principal utilidad reside en la posibilidad de movernos hacia delante o atrás dentro de esa lista.

8.5.1. Propiedades

Sólo tendremos disponible una propiedad para el historial de páginas.

- `length`: Devuelve la cantidad de sitios que hay dentro del historial.

8.5.2. Métodos

Los métodos de `history` son los que realmente nos ayudarán a desplazarnos por las páginas del historial.

- `back(), forward()`: Carga la página inmediatamente anterior o posterior a la actual. Esto equivale a hacer clic sobre el botón **Atrás** o **Adelante** de nuestro navegador.
- `go(posicion)`: Carga una página específica que está en la posición indicada dentro del historial. El parámetro `posicion` puede ser un entero (negativo para páginas anteriores, positivo para posteriores) o una cadena que represente un URL.

Truco: *El método* `back` *equivale a la llamada* `go(-1)`.

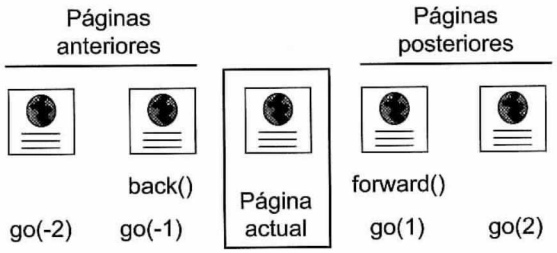


Figura 8.9. Representación de los métodos de history.

8.6. Objeto location

Este objeto está representado por una propiedad del objeto `window` y contiene el URL actual, del cual podremos obtener una serie de valores que nos pueden ser de utilidad en algunos de nuestros *scripts*.

8.6.1. Propiedades

Gracias a estas propiedades podremos "destripar" el URL en varias partes para usar la que más se ajuste a nuestras necesidades.

- `href`: Devuelve el URL completo. También podemos reemplazarlo para dirigirnos a otra página.
- `hostname`: Nos devuelve únicamente la parte del URL que hace referencia al nombre o dirección IP del servidor donde está alojada la página.
- `pathname`: Contiene la ruta que se sigue, dentro del servidor Web, hasta alcanzar la página actual.
- `hash`: Nos devuelve la parte del URL que hay después de la almohadilla (`#`). Esto representa un enlace o ancla (*anchor*).
- `search`: Nos devuelve la parte del URL que hay después del signo de interrogación (`?`). A esto se le conoce como consulta o *query string*, ya que es donde se pasan valores de una página a otra.
- `port`: Nos indica el puerto por el que hemos accedido al servidor. Esto viene indicado a continuación del

`hostname` mediante dos puntos (`:`). El puerto por defecto para servidores Web es el 80 y no es común verlo reflejado en el URL.
- `host`: Devuelve el nombre del servidor (`hostname`) y el número del puerto (`port`).
- `protocol`: Nos indica el protocolo que se ha usado para acceder al servidor (normalmente HTTP).

```
<SCRIPT TYPE="text/javascript">
   with(location) {
      alert("href: " + href + "\n" +
      "hostname: " + hostname + "\n" +
      "pathname: " + pathname + "\n" +
      "hash: " + hash + "\n" +
      "search: " + search + "\n" +
      "port: " + port + "\n" +
      "host: " + host + "\n" +
      "protocol: " + protocol);
   }
</SCRIPT>
```

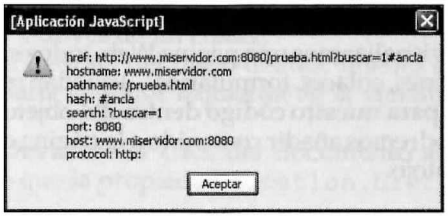


Figura 8.10. Propiedades del objeto location.

```
<SCRIPT TYPE="text/javascript">
   // Redireccionamos a otra página del mismo
   // servidor
   location.href = "mi_pagina.html";
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Redireccionamos a otra Web
   location.href = "http://www.miweb.com";
</SCRIPT>
```

8.6.2. Métodos

Su función principal es la de interactuar con un URL.

- `assign(URL)`: Carga un nuevo URL. Se creará una nueva entrada en el objeto `history`.

- `replace(URL)`: Reemplaza el URL actual por otro. No crea una nueva entrada en `history` sino que reemplaza la actual.
- `reload()`: Recarga la página actual. Es lo mismo que pulsar F5 en el navegador.

```
<SCRIPT TYPE="text/javascript">
   // Redireccionamos a otra página
   location.assign("mi_pagina.html");
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Obtenemos la hora actual
   var ahora = new Date();
   // Mostramos la hora
   alert(ahora);
   // Recargamos la página dentro de 5 segundos
   setTimeout("location.reload()", 5000);
</SCRIPT>
```

8.7. Objeto document

Cuando visualizamos una página Web, todos sus elementos (texto, imágenes, enlaces, formularios...) quedan representados y accesibles para nuestro código dentro del objeto `document`. Por tanto, podremos añadir contenido a la página o modificarlo a nuestro antojo.

8.7.1. Colecciones de objetos

Al igual que `window`, este objeto contiene a su vez una colección de otros objetos.

- `anchors`: Se trata de un *array* con todas las anclas (*anchors*) del documento. Estos elementos se identifican por la etiqueta <A> y tienen definido el atributo `NAME`. Esta colección de objetos se mantiene por temas de compatibilidad con versiones anteriores.
- `forms`: Contiene todas las referencias a los formularios (<FORM>) de la página, dentro de un *array*.
- `images`: Aquí están recopiladas todas las imágenes representadas con la etiqueta <IMG>.
- `links`: Contiene todos los enlaces de la página, como un *array*. Se identifican por la etiqueta <AREA> o <A> pero con el atributo `HREF` definido. No los confunda con los *anchor*, que deben tener el atributo `NAME` definido.

El acceso a cada objeto individual se realiza mediante su posición en el *array* o nombre (atributo `NAME`).

```
var miAncla = document.anchors[0];
var miFormulario = document.forms["formNuevoUsuario"];
var miImagen = document.images["logo"];
var miEnlace = document.link[3];
```

Los objetos `anchor`, `image` y `link` los veremos en este capítulo, mientras que al objeto `form` le dedicaremos un capítulo aparte ya que es más complejo.

8.7.2. Propiedades

- `cookie`: Nos devuelve en forma de cadena los valores de todas las *cookies* que tenga el documento.
- `domain`: Nos indica el nombre del servidor donde está alojado el documento.
- `lastModified`: Nos dice la fecha de la última modificación del documento.
- `referrer`: Indica el URL del documento que llamó al actual, a través de un enlace.
- `title`: Se trata del título del documento que aparece en la parte superior izquierda de la ventana del navegador.
- `URL`: Devuelve el URL del documento actual. Es lo mismo que la propiedad `location.href`.

```
<SCRIPT TYPE="text/javascript">
   with(document) {
      alert("cookie: " + cookie + "\n" +
      "domain: " + domain + "\n" +
      "lastModified: " + lastModified + "\n" +
      "referrer: " + referrer + "\n" +
      "title: " + title + "\n" +
      "URL: " + URL);
   }
</SCRIPT>
```

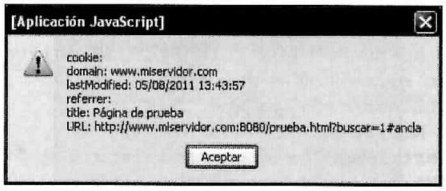


Figura 8.11. Propiedades del objeto document.

8.7.3. Métodos

- `write(texto)`, `writeln(texto)`: Escribe texto HTML en el documento. El método `writeln` añade además un salto de línea (
) al final del texto que hayamos pasado como parámetro.
- `open()`: Permite escribir sobre el documento mediante los métodos `write` o `writeln` una vez que el documento ha sido cargado.
- `close()`: Finaliza la escritura sobre el documento, abierta mediante el método `open`.
- `getElementById(identificador)`: Devuelve el primer elemento del documento cuyo atributo `ID` coincida con el parámetro `identificador`.
- `getElementsByName(nombre)`: Devuelve un *array* de elementos cuyo atributo `NAME` coincida con el indicado.
- `getElementsByTagName(etiqueta)`: Devuelve un *array* de elementos cuya etiqueta sea la misma que la especificada por el parámetro.

Nota: *Los métodos `write`, `writeln`, `open` y `close` en ningún momento modificarán el fichero físico sino que sólo cambiarán el texto visible en pantalla.*

```
<SCRIPT TYPE="text/javascript">
   // Escribimos algo en la página
   document.write("Esto es un texto <B>en negrita
   </B>.");
</SCRIPT>
<DIV ID="capa1" TITLE="CapaPrueba">Esto es una
capa HTML</DIV>
<DIV ID="capa2" TITLE="CapaFondo">Esto es otra
capa más</DIV>
<SCRIPT TYPE="text/javascript">
   // Recogemos un elemento de la página
   var elemento = document.getElementById("capa1");
   // Mostramos su atributo title ("CapaPrueba")
   alert(elemento.title);
</SCRIPT>
<SCRIPT TYPE="text/javascript">
   // Recogemos una colección de objetos
   var elementos = document.
   getElementsByTagName("DIV");
   // Mostramos cuántos hay (2)
   alert("Hay " + elementos.length + " elementos
   DIV");
</SCRIPT>
```


8.8. Objeto anchor

Este objeto engloba todas las propiedades y métodos que tienen los enlaces internos al documento, también llamados anclas o *anchor*. Dentro de una página Web están identificadas con la etiqueta <A> y tienen un valor en el atributo `NAME`, sin importar el resto de atributos. Esto es así para poder mantener la compatibilidad con versiones antiguas de JavaScript.

8.8.1. Propiedades

Las propiedades de este objeto representan básicamente todos los atributos HTML que puede tener. Salvo que se indique lo contrario, podremos, por tanto, obtener cómo modificar el valor de los siguientes atributos:

- `id`: Identificador del ancla (atributo `ID`).
- `name`: Nombre del ancla (atributo `NAME`).
- `target`: Ventana o marco donde se cargará el ancla (atributo `TARGET`).
- `text`: Esta propiedad es únicamente de lectura, y nos permite saber el texto que contiene el ancla (lo que ve el usuario en la página).
- `innerHTML`: Es igual que el anterior, pero nos permite además modificar ese texto.

Veamos un ejemplo con un sencillo *anchor*:

```
<A ID="lstPropiedades" NAME="listado" TARGET="_self">Ver
listado de propiedades</A>
<SCRIPT TYPE="text/javascript">
   with(document.anchors["listado"]) {
      alert("id: " + id + "\n" +
      "nombre: " + name + "\n" +
      "destino: " + target + "\n" +
      "texto (text): " + text + "\n" +
      "texto (innerHTML): " + innerHTML);
   }
</SCRIPT>
```

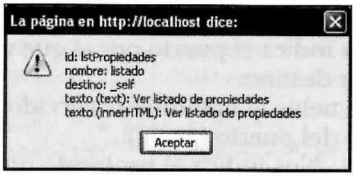


Figura 8.12. Propiedades del objeto anchor.

8.8.2. Métodos

Este objeto sólo cuenta con dos métodos referentes a la posesión del foco.

- `focus()`, `blur()`: Hacen que reciba o pierda el foco, respectivamente.

8.9. Objeto link

Mediante este objeto podremos tener acceso a todos los enlaces externos al documento o página actual. Se consideran objetos `link` todos los elementos HTML con etiquetas <AREA> y los <A> con el atributo `HREF` definido (aunque tengan también valor en `NAME`).

8.9.1. Propiedades

La lista de propiedades que tiene este tipo de objeto es muy similar a la que vimos para `location` pero aplicadas sobre el enlace, y también comparte algunas con `anchor`.

- `id`: Identificador del enlace (atributo `ID`).
- `href`: Devuelve el URL completo del enlace (atributo `HREF`).
- `target`: Ventana o marco donde se cargará el enlace (atributo `TARGET`).
- `hostname`: Nos devuelve únicamente la parte del URL que hace referencia al nombre o dirección IP del servidor donde está alojada la página destino.
- `pathname`: Contiene la ruta que se sigue hasta alcanzar la página destino.
- `hash`: Nos devuelve la parte del URL que hay después de la almohadilla (#). Esto representa un enlace o ancla (*anchor*).
- `search`: Nos devuelve la parte del URL que hay después del signo de interrogación (?), también llamado consulta o *query string*.
- `port`: Nos indica el puerto por el que vamos a acceder al servidor destino.
- `host`: Devuelve el nombre del servidor (`hostname`) y el número del puerto (`port`).
- `protocol`: Nos indica el protocolo que vamos a usar para acceder al servidor (normalmente HTTP).

- `text`: Nos permite saber el texto del enlace (sólo lectura).
- `innerHTML`: Nos permite obtener y modificar el texto del enlace.

```
<A ID="lstPropiedades" HREF="#propiedades">Ver
listado de propiedades</A>
<SCRIPT TYPE="text/javascript">
   with(document.links[0]) {
      alert("id: " + id + "\n" +
      "URL: " + href  + "\n" +
      "destino: " + target + "\n" +
      "servidor: " + hostname + "\n" +
      "ruta: " + pathname + "\n" +
      "hash: " + hash + "\n" +
      "consulta: " + search + "\n" +
      "puerto: " + port + "\n" +
      "servidor y puerto: " + host + "\n" +
      "protocolo: " + protocol + "\n" +
      "texto (text): " + text + "\n" +
      "texto (innerHTML): " + innerHTML);
   }
</SCRIPT>
```

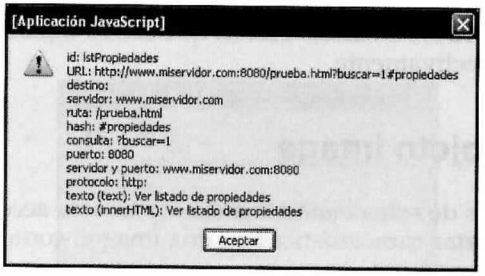


Figura 8.13. Propiedades del objeto link.

```
<A ID="lnkHome" HREF="inicio.html">Ir a página
principal</A>
<SCRIPT TYPE="text/javascript">
   with(document.links[0]) {
      alert("id: " + id + "\n" +
      "URL: " + href  + "\n" +
      "destino: " + target + "\n" +
      "servidor: " + hostname + "\n" +
      "ruta: " + pathname + "\n" +
      "hash: " + hash + "\n" +
      "consulta: " + search + "\n" +
      "puerto: " + port + "\n" +
      "servidor y puerto: " + host + "\n" +
```

```
        "protocolo: " + protocol + "\n" +
        "texto (text): " + text + "\n" +
        "texto (innerHTML): " + innerHTML);
    }
</SCRIPT>
```

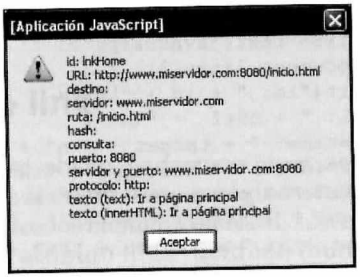


Figura 8.14. Propiedades del objeto link.

8.9.2. Métodos

Dada la similitud de `link` con el objeto `anchor`, cuenta con los mismos métodos que éste.

- `focus()`, `blur()`: Hacen que reciba o pierda el foco, respectivamente.

8.10. Objeto image

A través de este objeto seremos capaces de acceder y manipular ciertas características de una imagen contenida en la página. No se puede hacer gran cosa, pero siempre está bien contar con ello.

Este objeto no cuenta con ningún método, así que veremos únicamente sus propiedades.

8.10.1. Propiedades

- `id`: Se corresponde con el identificador de la imagen (atributo `ID`).
- `name`: Nos da acceso al nombre del objeto (atributo `NAME`).
- `src`: Con esta propiedad podremos manipular el URL de la imagen que se muestra (atributo `SRC`). Cuando se recupera su valor se muestra el URL completo de la

imagen, es decir, incluyendo el nombre del servidor y la ruta hasta la imagen.

- `width`, `height`: Como podrá adivinar, nos permite conocer y cambiar la anchura y altura de la imagen (atributos `WIDTH` y `HEIGHT`).
- `alt`: Corresponde con el texto alternativo que se muestra al usuario cuando la imagen no puede ser cargada (atributo `ALT`).
- `title`: Afecta al texto que se muestra sobre la imagen cuando ponemos el cursor sobre ella (atributo `TITLE`).

```
<IMG ID="imgLogo" NAME="logo" SRC="imagenes/logo.
jpg" WIDTH="120" HEIGHT="45" ALT="Logo de la
empresa" TITLE="Este es nuestro logo"></IMG>
<SCRIPT TYPE="text/javascript">
   with(document.images[0]) {
      alert("id: " + id + "\n" +
      "nombre: " + name+ "\n" +
      "URL: " + src+ "\n" +
      "ancho: " + width+ "\n" +
      "alto: " + height + "\n" +
      "texto alternativo: " + alt + "\n" +
      "texto sobre imagen: " + title);
   }
</SCRIPT>
```

Figura 8.15. Propiedades del objeto image.

9 Formularios

En este capítulo verá una de las principales utilidades de JavaScript, al menos teniendo en cuenta el alcance que tiene esta guía: la posibilidad de controlar los formularios de una página.

La aparición de JavaScript en la programación Web fue motivada, en parte, por la necesidad de incluir un mecanismo de control sobre los formularios directamente en el lado cliente para así evitar cargar en exceso el servidor, liberándolo además de esas tareas.

Un formulario HTML permite recolectar información proporcionada directamente por el usuario que visita nuestra página para después utilizarla en nuestro sitio Web. El tipo de información que se puede recoger es muy variada, siendo desde datos personales hasta una lista de la compra.

9.1. Formularios en HTML

En este apartado se explicará cómo crear un formulario en nuestra página, así como los elementos que puede contener. No pretendo que sea un manual detallado, pero sí veo aconsejable desarrollarlo para que conozca todas las partes y características de un formulario de cara a controlarlas después con JavaScript.

9.1.1. Formulario

Dentro de una página Web pueden existir multitud de elementos, y uno de ellos son los formularios. Para incluir un formulario en nuestra página debemos utilizar la etiqueta

<FORM> al incio y </FORM> al final. Esta etiqueta por si sola no es representada por el navegador, esto es, el usuario no verá nada en su pantalla, sino que simplemente le indica que a continuación habrá elementos de formulario (campos), los cuales sí son mostrados al usuario para que pueda introducir la información pertinente.

Como todo elemento HTML, esta etiqueta dispone de una serie de atributos que nos permitirán amoldarla a nuestras necesidades:

- ID: Con él podremos dar al formulario un identificador único como elemento de la página.
- NAME: Nos permite asignarle un nombre al formulario para poder identificarlo dentro de la página.
- ACTION: Aquí debemos indicar a qué URL se van a enviar los datos que recoja el formulario. Por lo general, suele ser una página con un lenguaje de servidor, como PHP o ASP, que se encargará de recopilar los datos y tratarlos en el servidor Web.
- METHOD: Con este atributo podremos definir de qué manera se van a enviar los datos. Existen dos métodos:
  - get: Todos los valores del formulario se enviarán dentro del URL detrás del carácter de interrogación (?), formando la cadena de consulta o *query string*. Este método no es aconsejable para tratar información confidencial, ya que todos los datos quedan a la vista del usuario en la barra de direcciones.
  - post: En este caso los datos van encapsulados en la petición HTTP al servidor Web haciendo que todos los datos estén ocultos de cara al usuario, lo cual le convierte en el candidato perfecto para manejar información sensible como contraseñas o números de identificación.
- TARGET: Con esto indicamos en qué ventana se va a abrir el URL que recogerá el formulario (atributo ACTION). Los posibles valores son:
  - _blank: El URL se abrirá en una ventana nueva.
  - _self: En este caso se abrirá la ventana o marco actual. Es el valor por defecto.
  - _parent: La ventana que contenga los marcos será la encargada de cargar el URL. Si no hubiera marcos entonces se abrirá en la ventana actual.
  - _top: El URL será abierto en la ventana de nivel inmediatamente superior o, en caso de no haberla, en la actual.

- ENCTYPE: Nos permite especificar la codificación que llevarán los datos que contenga el formulario:
  - text/plain: Los caracteres que incluya cada campo del formulario serán enviados tal cual. Debido a ello, si el formulario incluye el envío de ficheros, su contenido no será tratado adecuadamente y tampoco podrá ser recuperado en la página destino (atributo ACTION). Este valor se toma por defecto si no se especifica este atributo.
  - multipart/form-data: Los caracteres también son enviados sin modificar, pero sí permite incluir ficheros junto con el formulario.
  - application/x-www-form-urlencoded: En este caso, los caracteres de cada campo del formulario serán codificados antes de ser enviados. Dicha codificación consiste en sustituir los espacios por signos de suma (+) y convertir los caracteres especiales a un valor hexadecimal.

No es obligatorio definir todos los atributos que acabamos de presentar, por lo que aquellos que no estén presentes tomarán su valor por defecto (si tuvieran). Por lo general, los campos ID, NAME y ACTION se deben especificar para tener unos datos mínimos.

Veamos algunos ejemplos, con funcionalidades inventadas, de cómo se incluiría un formulario en nuestra página Web:

```
<HTML>
<BODY>
<H3>Formulario para dar de alta un usuario</H3>
<FORM ID="formulario1" NAME="formRegistro"
ACTION="registrar.php" METHOD="get">
</FORM>
<H3>Formulario para subir ficheros al servidor</H3>
<FORM ID="formulario2" NAME="formFicheros"
ACTION="guardar_ficheros.php" METHOD="post"
ENCTYPE="multipart/form-data">
</FORM>
<H3>Formulario para confirmar una compra</H3>
<FORM ID="formulario3" NAME="formCompra"
ACTION="confirmar_compra.php" TARGET="_blank">
</FORM>
</BODY>
</HTML>
```

Como ya hemos dicho, si cargamos una página con este código no se verá nada en la pantalla ni serán de utilidad puesto que deberemos incluir además alguno de los campos que

vamos a ver a continuación. En los próximos ejemplos vamos a usar estos mismos formularios para ir completándolos y, finalmente, aplicarles controles mediante JavaScript.

9.1.2. Campos de entrada de datos

Estos campos son unos de los más importantes dentro de un formulario, ya que nos permiten tomar datos que el usuario escribe directamente "de su puño y letra". El navegador los representa como un cuadro de texto (de distintos tamaños) y seguro que los ha visto más de una vez en muchas de las páginas que visita frecuentemente, como el correo Web, la cuenta del banco o un foro. Los campos de entrada de datos van identificados por las etiquetas `<INPUT>` y `<TEXTAREA>`. Cada una de ellas tiene una serie de atributos, aunque tienen en común los siguientes:

- `NAME`: Como ocurre en todos los elementos HTML, este atributo nos permite asignar un nombre al campo para que pueda ser identificado.
- `DISABLED`: Nos permite desactivar un campo de forma que no pueda ser utilizado por el usuario. La información que contenga será visible, pero el campo aparecerá sombreado. No necesita ir acompañado de un valor como, por ejemplo, el atributo `NAME` sino que el campo se desactivará simplemente con tener este atributo presente en su etiqueta HTML.
- `READONLY`: Es similar a `DISABLED`, salvo que no sombrea el campo totalmente.

Nota: *Los campos con el atributo `DISABLED` presente no serán enviados junto con el resto del formulario. Si desea enviar un campo que no deba ser manipulado por el usuario, utilice `READONLY`.*

Veamos ahora los atributos propios de la etiqueta `<INPUT>`, que en este caso representa las cajas de texto simples:

- `TYPE`: Posibilita definir el tipo de la caja de texto mediante uno de estos valores:
  - `text`: Hace que se muestre una caja para introducir texto genérico.
  - `password`: Muestra una caja de texto con la peculiaridad de que los caracteres escritos se representan como asteriscos (*) para que no queden visibles en la pantalla. A la hora de trabajar con este campo, el texto se recupera tal y como se escribió.
  - `hidden`: Crea una caja de texto que estará oculta a la vista del usuario, de modo que no estará representada en la pantalla. Su utilidad reside en transportar valores temporales o de estado.
  - `file`: Nos mostrará un cuadro preparado para poder buscar un fichero en el ordenador del usuario y adjuntarlo al formulario.
- `MAXLENGTH`: Nos permite limitar el número máximo de caracteres que se pueden escribir en un cuadro de texto.
- `SIZE`: Con este atributo establecemos el número de caracteres que se verán al mismo tiempo en el cuadro de texto. No impone un límite en la longitud del texto sino únicamente en la cantidad de caracteres que podremos ver en nuestra pantalla dentro de ese cuadro. Si le resulta más fácil, se podría definir como el ancho que tendrá el cuadro. Obviamente este valor no tiene efecto sobre los campos ocultos (`hidden`).
- `VALUE`: Este atributo representa el texto que contiene el cuadro correspondiente. En caso de tener un campo de tipo `file` este atributo no tiene utilidad.
  Finalmente veremos los atributos que podremos aplicar únicamente en los campos `<TEXTAREA>`, que no son más que unos cuadros de texto en los que es posible escribir varias líneas:
- `COLS`, `ROWS`: Serían el equivalente al atributo `SIZE` de las cajas simples. Con ellos podremos establecer, respectivamente, el ancho y el alto que tendrá esta caja de texto, siendo `ROWS` el número de filas y `COLS` el número de caracteres visibles por fila.

Nota: *Si no se especifica un tamaño para los cuadros de texto (atributos `SIZE`, `COLS` o `ROWS`), cada navegador asignará unos valores por defecto.*

Veamos ahora cómo insertar este tipo de campos en nuestros formularios de ejemplo.

```
<HTML>
<BODY>
<H3>Formulario para dar de alta un usuario</H3>
<FORM ID="formulario1" NAME="formRegistro"
ACTION="registrar.php" METHOD="get">
Tu nombre: <INPUT TYPE="text" ID="entrada1"
```

```
NAME="nombreUsuario" MAXLENGTH="8" />
(Máximo 8 caracteres)<BR />
Tu contraseña: <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
Tu email: <INPUT TYPE="text" ID="entrada3"
NAME="email" SIZE="40" />
</FORM>
```

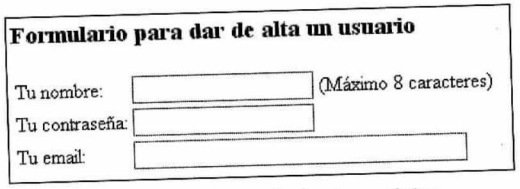


Figura 9.1. Formulario de registro.

```
<H3>Formulario para subir ficheros al servidor</H3>
<FORM ID="formulario2" NAME="formFicheros"
ACTION="guardar_ficheros.php" METHOD="post"
ENCTYPE="multipart/form-data">
Fichero: <INPUT TYPE="file" ID="entrada4"
NAME="fichero" /><BR />
Descripción: <TEXTAREA ID="entrada5" NAME="descripcion"
COLS="40" ROWS="3"></TEXTAREA>
</FORM>
```

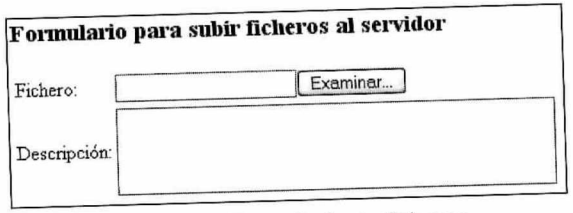


Figura 9.2. Formulario de ficheros.

```
<H3>Formulario para confirmar una compra</H3>
<FORM ID="formulario3" NAME="formCompra"
ACTION="confirmar_compra.php" TARGET="_blank">
<!-- Campo oculto con la fase del proceso de compra -->
<INPUT TYPE="hidden" ID="entrada6" NAME="fase"
VALUE="confirmacion">
Tu usuario: <INPUT TYPE="text" ID="entrada7"
NAME="usuario" MAXLENGTH="8"><BR />
Importe total: <INPUT TYPE="text" ID="entrada8"
NAME="importe" SIZE="10" VALUE="54,75" READONLY> euros
</FORM>
</BODY>
</HTML>
```

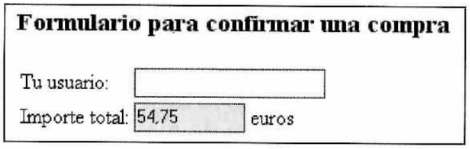



Figura 9.3. Formulario de compra.

9.1.3. Campos de selección de datos

Aparte de la entrada de datos mediante teclado, el usuario también podrá aportar información escogiendo un valor dentro de una serie de opciones que se le ofrece. Para incluir este tipo de campos en un formulario deben utilizarse las etiquetas `<INPUT>` o `<SELECT>`. La primera ya la conoce, aunque en este caso hay que definirla de forma que no corresponda con una caja de texto, y la segunda muestra una lista desplegable o *drop-down* para poder seleccionar un valor de esa lista.

Ambas etiquetas tienen una serie de atributos comunes:

- `NAME`: Representa el nombre con el que se identifica el campo.
- `DISABLED`: Permite deshabilitar el campo, de forma que el usuario no podrá interactuar con él.

La etiqueta `<INPUT>` representa en este caso unos campos o datos que son opcionales en nuestro formulario (por ejemplo, si se desea recibir publicidad en el email), aunque también se utiliza para poder elegir una opción entre varias disponibles (puntuación entre 1 y 10 en un test, por ejemplo). No almacenan un valor escrito por el usuario sino que únicamente pueden estar seleccionados o no. Cuando la etiqueta `<INPUT>` adopta esta funcionalidad dispone de tres atributos:

- `TYPE`: Nos permite definir el tipo del elemento con uno de estos valores:
  - `radio`: Mostrará un botón de radio, es decir, un círculo vacío cuando no está seleccionado y relleno cuando lo está. Se utilizan cuando queremos que el usuario elija un único valor entre una serie de opciones (puntuación entre 1 y 10).
  - `checkbox`: Este tipo representa una caja de validación que estará vacía cuando no esté seleccionada, y rellena con un aspa cuando sí lo esté. Sirven para que el usuario pueda activar o desactivar las opciones

que representen, como por ejemplo si desea recibir un boletín semanal en su correo electrónico.

- CHECKED: Hace que el elemento esté seleccionado desde un principio. Si el usuario lo deselecciona, este atributo no tendrá efecto a la hora de mandar el formulario. Al igual que los atributos DISABLED o READONLY, no es necesario un valor a continuación.
- VALUE: Representa el valor que tiene asignado el campo. Este valor no será mostrado junto al elemento de selección sino que será el valor que se envíe con el formulario, siempre y cuando el campo esté seleccionado. Sin embargo, si el elemento no está seleccionado el valor que enviaremos será una cadena vacía.
  Por otro lado, la etiqueta <SELECT> mostrará una lista desplegable con varias opciones entre las cuales el usuario podrá seleccionar una o varias. Un ejemplo de su uso sería que mostrase al usuario una lista de los meses del año haciendo que elija aquel que corresponda con su fecha de nacimiento. Esta etiqueta también dispone de atributos adicionales:
- SIZE: Nos permite definir cuántos elementos de la lista se mostrarán al mismo tiempo. El valor por defecto es 1, y el elemento se representa como una lista desplegable. Si este atributo tiene un valor superior, entonces se mostrará como un cuadro de texto con una barra de desplazamiento vertical tomando una apariencia similar a un TEXTAREA.
- MULTIPLE: Indica que se podrá seleccionar más de un valor de la lista. Para ello, hay que mantener presionado el botón **Control** mientras se hace clic sobre las opciones que se quiera. Este atributo no necesita un valor asociado, de modo que con estar presente ya se activa esta funcionalidad.

Para definir cada una de las opciones, que contienen las listas desplegables, hay que usar otra etiqueta, <OPTION>, que contiene los atributos comunes de <SELECT> (NAME y DISABLED) junto con otros dos:

- VALUE: Representa el valor que tiene asignada la opción. Este valor no se muestra junto al elemento, sino que es enviando dentro del formulario. Para asociarle un texto visible al usuario sólo hay que escribirlo al lado de la etiqueta <OPTION>, y no tiene por qué coincidir con el valor que se va a mandar. Podemos tener una opción con el valor 1 y el texto "Enero".

- SELECTED: Hace que la opción aparezca como seleccionada en la lista desde un principio. Si el usuario cambia la selección se anula el efecto de este atributo.

Veamos ahora cómo incluir estos nuevos campos a nuestros formularios.

```
<HTML>
<BODY>
<H3>Formulario para dar de alta un usuario</H3>
<FORM ID="formulario1" NAME="formRegistro"
ACTION="registrar.php" METHOD="get">
Tu nombre: <INPUT TYPE="text" ID="entrada1"
NAME="nombreUsuario" MAXLENGTH="8" /> (Máximo 8
caracteres)<BR />
Tu contraseña: <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
Tu email: <INPUT TYPE="text" ID="entrada3" NAME="email"
SIZE="40" />
Tu país: <SELECT ID="seleccion1" NAME="pais">
      <OPTION ID="opcion1" VALUE="AND">Andorra</OPTION>
      <OPTION ID="opcion2" VALUE="ESP"
      SELECTED>España</OPTION>
      <OPTION ID="opcion3" VALUE="POR" >Portugal
      </OPTION>
</SELECT>
</FORM>
```

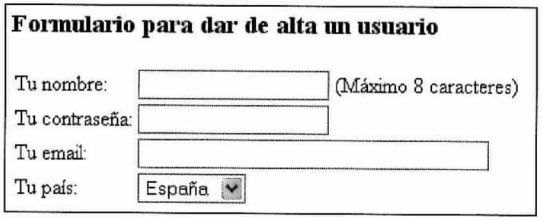


Figura 9.4. Formulario de registro.

```
<H3>Formulario para subir ficheros al servidor</H3>
<FORM ID="formulario2" NAME="formFicheros"
ACTION="guardar_ficheros.php" METHOD="post"
ENCTYPE="multipart/form-data">
Fichero: <INPUT TYPE="file" ID="entrada4"
NAME="fichero" /><BR />
Descripción: <TEXTAREA ID="entrada5" NAME="descripcion"
COLS="40" ROWS="3"></TEXTAREA><BR />
<INPUT TYPE="checkbox" ID="seleccion2"
NAME="comprimir" /> Comprimir en ZIP
</FORM>
```

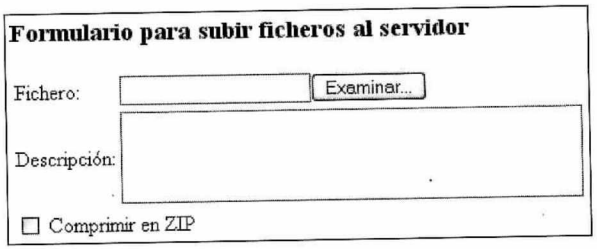


Figura 9.5. Formulario de ficheros.

```
<H3>Formulario para confirmar una compra</H3>
<FORM ID="formulario3" NAME="formCompra"
ACTION="confirmar_compra.php" TARGET="_blank">
<!-- Campo oculto con la fase del proceso de compra -->
<INPUT TYPE="hidden" ID="entrada6" NAME="fase"
VALUE="confirmacion">
Tu usuario: <INPUT TYPE="text" ID="entrada7"
NAME="usuario" MAXLENGTH="8"><BR />
Importe total: <INPUT TYPE="text" ID="entrada8"
NAME="importe" SIZE="10" VALUE="54,75" READONLY>
euros<BR />
Pago: <INPUT TYPE="radio" ID="seleccion3" NAME="pago"
VALUE="Tarjeta"> Tarjeta VISA/MASTERCARD <INPUT
TYPE="radio" ID="seleccion4" NAME="pago" VALUE="Paypal">
Paypal <INPUT TYPE="radio" ID="seleccion5" NAME="pago"
VALUE="Efectivo" CHECKED> En efectivo
</FORM>
</BODY>
</HTML>
```

Figura 9.6. Formulario de compra.

9.1.4. Botones

Por último veremos otros elementos de gran importancia dentro de un formulario: los botones. Con ellos podremos ejecutar una serie de operaciones con los datos introducidos en el formulario. Todos los botones se definen, de nuevo, con la etiqueta `<INPUT>` y los siguientes atributos:

- `NAME`: Indica el nombre del campo, mediante el cual es posible identificarlo.
- `TYPE`: Nos permite especificar el tipo de botón que vamos a incluir en el formulario:
  - `button`: Describe un botón de uso genérico, al que podremos asignar la acción que debe realizar al ser pulsado, que normalmente se trata de una función JavaScript.
  - `submit`: Es un tipo especial de botón que envía automáticamente toda la información del formulario cuando es pulsado.
  - `reset`: Se trata de otro botón especial que, cuando es pulsado, vuelve a dejar todo el formulario tal y como estaba al cargar la página, borrando toda la información que hubiese introducido el usuario y eliminando sus selecciones.
- `VALUE`: Al contrario que otros campos, el valor de este atributo corresponde con el texto que se muestra dentro del botón para así informar al usuario de la función que desempeña. Por ejemplo, se puede crear un botón con valor "Registrarse" o "Calcular". Si no se añade este atributo, el texto mostrado por defecto es una cadena vacía, excepto para los de tipo `submit` que será "Enviar consulta", y para el tipo `reset` que mostrará "Reestablecer". Estos textos por defecto dependen también del idioma del sistema y pueden variar de un navegador a otro, pero serán similares.

Finalmente, podremos completar nuestros tres formularios de ejemplo, añadiendo los botones correspondientes que determinarán las acciones que se llevarán a cabo en cada uno de ellos.

```
<HTML>
<BODY>
<H3>Formulario para dar de alta un usuario</H3>
<FORM ID="formulario1" NAME="formRegistro"
ACTION="registrar.php" METHOD="get">
Tu nombre: <INPUT TYPE="text" ID="entrada1"
NAME="nombreUsuario" MAXLENGTH="8" /> (Máximo 8
caracteres)<BR />
Tu contraseña: <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
Tu email: <INPUT TYPE="text" ID="entrada3" NAME="email"
SIZE="40" />
Tu país: <SELECT ID="seleccion1" NAME="pais">
```

```
      <OPTION ID="opcion1" VALUE="AND">Andorra</OPTION>
      <OPTION ID="opcion2" VALUE="ESP" SELECTED>España</
OPTION>
      <OPTION ID="opcion3" VALUE="POR" >Portugal</
OPTION>
</SELECT><BR /><BR />
<INPUT TYPE="reset" ID="boton1" NAME="limpiar" />
<INPUT TYPE="submit" ID="boton2" NAME="enviar" />
</FORM>
```

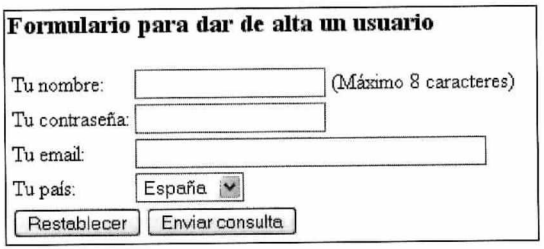


Figura 9.7. Formulario de registro.

```
<H3>Formulario para subir ficheros al servidor</H3>
<FORM ID="formulario2" NAME="formFicheros"
ACTION="guardar_ficheros.php" METHOD="post"
ENCTYPE="multipart/form-data">
Fichero: <INPUT TYPE="file" ID="entrada4" NAME="fichero"
/><BR />
Descripción: <TEXTAREA ID="entrada5" NAME="descripcion"
COLS="40" ROWS="3"></TEXTAREA><BR />
 <INPUT TYPE="checkbox" ID="seleccion2" NAME="comprimir"
/> Comprimir en ZIP<BR />
 <INPUT TYPE="reset" ID="boton3" NAME="limpiar"
VALUE="Cambiar fichero"/> <INPUT TYPE="submit"
ID="boton4" NAME="enviar" VALUE="Subir fichero"/>
 </FORM>
```

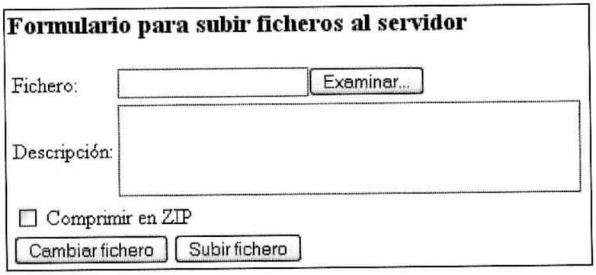


Figura 9.8. Formulario de ficheros.


```
<H3>Formulario para confirmar una compra</H3>
<FORM ID="formulario3" NAME="formCompra"
ACTION="confirmar_compra.php" TARGET="_blank">
<!-- Campo oculto con la fase del proceso de compra -->
<INPUT TYPE="hidden" ID="entrada6" NAME="fase"
VALUE="confirmacion">
Tu usuario: <INPUT TYPE="text" ID="entrada7"
NAME="usuario" MAXLENGTH="8"><BR />
Importe total: <INPUT TYPE="text" ID="entrada8"
NAME="importe" SIZE="10" VALUE="54,75" READONLY>
euros<BR />
Pago: <INPUT TYPE="radio" ID="seleccion3" NAME="pago"
VALUE="Tarjeta"> Tarjeta VISA/MASTERCARD <INPUT
TYPE="radio" ID="seleccion4" NAME="pago" VALUE="Paypal">
Paypal <INPUT TYPE="radio" ID="seleccion5" NAME="pago"
VALUE="Efectivo" CHECKED> En efectivo<BR /><BR />
<INPUT TYPE="button" ID="boton5" NAME="confirmar"
VALUE="Confirmar y seguir >" />
</FORM>
</BODY>
</HTML>
```

Figura 9.9. Formulario de compra.

9.1.5. Resumen de tipos de elementos <INPUT>

Para que no tenga que repasar de arriba abajo este capítulo en busca de los posibles valores del atributo `TYPE` para los campos `<INPUT>`, le incluyo a continuación una pequeña tabla resumen que espero le sea de ayuda.

Tabla 9.1. Valores del atributo TYPE en campos <INPUT>.

Elemento	*Valores TYPE*
Entrada de datos	text, password, file, hidden
Selección de datos	radio, checkbox
Botones	button, submit, reset

9.2. Formularios en JavaScript

Una vez que ya conocemos los formularios HTML y todo lo que pueden incluir vamos con lo que nos interesa: cómo acceder y trabajar con esos formularios desde nuestro código JavaScript.

9.2.1. Formulario

Como dijimos en el capítulo dedicado al DOM, el objeto `document` nos permite un acceso directo a todos los formularios de la página a través de la colección `forms`. De este modo, acceder a un formulario resulta tan sencillo como usar esta colección junto con un índice o el nombre de uno de los formularios existentes (atributo `NAME`).

```
<SCRIPT TYPE="text/javascript">
   // Acceso a un formulario
   var miForm = document.forms[0];
   var miForm = document.forms["formRegistro"];
</SCRIPT>
```

Aunque esta es la forma más correcta, también es posible acceder directamente con el nombre de la siguiente manera:

```
<SCRIPT TYPE="text/javascript">
   // Acceso a un formulario con el nombre
   var miForm = document.formRegistro;
</SCRIPT>
```

Tomando como base los tres formularios que hemos definido en el apartado anterior, veamos cómo obtener el número de formularios que contiene la página:

```
<SCRIPT TYPE="text/javascript">
   // Cantidad de formularios en la página
   alert("Hay " + document.forms.length + " formularios
   en esta página"); // Hay 3 formularios
</SCRIPT>
```

A continuación veremos todos los elementos a los que podemos acceder dentro de un formulario.

Colecciones de objetos

- `elements`: Se trata de una *array* con todos y cada uno de los campos que componen el formulario. Se puede acceder a cada campo mediante un índice o su nombre.


```
<SCRIPT TYPE="text/javascript">
   // Acceso a un campo
   var primerCampo = document.forms[0].
   elements[0];
   var unCampo = document.forms[0].
   elements["nombre"];
</SCRIPT>
```

También se puede usar directamente los nombres del formulario y/o el campo:

```
<SCRIPT TYPE="text/javascript">
   // Acceso a un campo
   var primerCampo = document.formRegistro.
   elements[0];
   var unCampo = document.forms[0].nombreUsuario;
   var otroCampo = document.formRegistro.clave;
</SCRIPT>
```

A través de esta colección de elementos podremos conocer el número de campos que contiene cada formulario:

```
<SCRIPT TYPE="text/javascript">
   // Cantidad de campos en cada formulario
   for(var i=0; i<document.forms.length; i++) {
      alert("El formulario " + i + " tiene " + document.
      forms[i].elements.length + " campos");
   }
</SCRIPT>
```

> **Nota**: *El tercer formulario contiene 7 campos puesto que también se contabilizan los campos ocultos (`hidden`).*

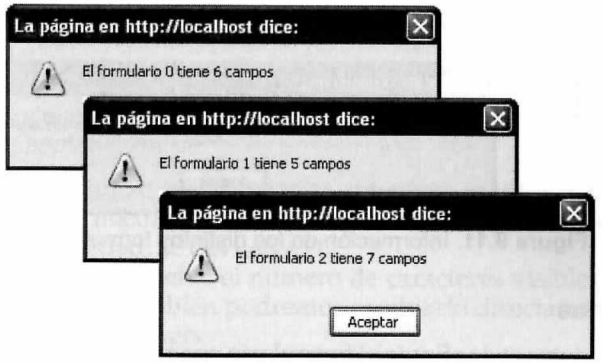


Figura 9.10. Información del número de campos de los distintos formularios.

Propiedades

Básicamente podremos obtener y modificar todos los atributos HTML del formulario

- `id`: Representa el identificador del formulario (atributo `ID`).
- `name`, `action`, `method`, `target`, `enctype`: Representan cada uno de los atributos que hemos descrito en el apartado anterior.
- `length`: Nos dirá el número de campos que tiene el formulario. Es lo mismo que `elements.length`.

Gracias a estas propiedades podremos obtener algo más de información de los formularios para usarlo en nuestros *scripts*:

```
<SCRIPT TYPE="text/javascript">
   var formulario;
   // Cantidad de campos en cada formulario
   for(var i=0; i<document.forms.length; i++) {
      formulario = document.forms[i];
      alert("El formulario " + i + " (" + formulario
      name + ") tiene " + formulario.length + " campos
      que va a enviar al URL " + formulario.action +
      " utilizando el método " + formulario.method);
   }
</SCRIPT>
```

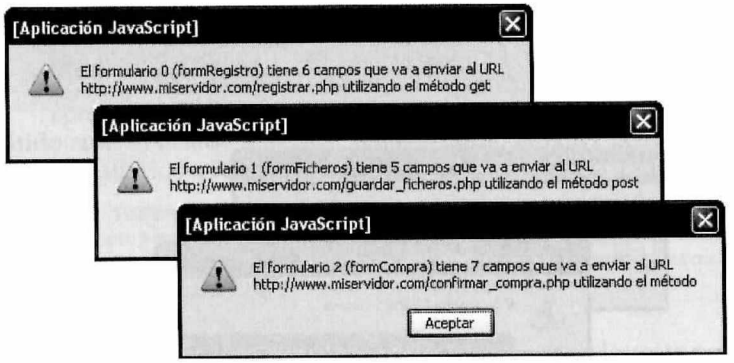


Figura 9.11. Información de los distintos formularios.

Métodos

- `submit()`: Envía el formulario con toda la información que haya introducido el usuario en él. Equivale a un botón de tipo `submit`.

- `reset()`: Deja el formulario en blanco, tal y como estaba al cargar la página. Realiza la misma acción que un botón de tipo reset.

A modo de ejemplo de estos métodos, crearemos un pequeño *script* que nos limpie el primer formulario pasados 10 segundos. Para ver su efecto escriba algo en alguno de sus campos antes de que pase ese tiempo.

```
<SCRIPT TYPE="text/javascript">
   // Función que limpia un formulario
   function limpiarForm(formulario) {
      formulario.reset();
   }
   // Inicialización de temporizador
   setTimeout("limpiarForm(document.forms[0])", 10000);
</SCRIPT>
```

9.2.2. Campos de entrada de datos

JavaScript también nos permite acceder a estos campos para manejarlos desde nuestros *scripts*.

Propiedades

Al igual que los elementos HTML que representan estos campos, hay una serie de propiedades comunes para todos ellos:

- `id`: Representa el identificador del formulario (atributo `ID`).
- `disabled`, `name`, `readOnly`, `type`, `value`: Devuelven el valor del atributo HTML correspondiente.
- `form`: Nos indica el objeto de formulario donde está contenido el campo actual.

Las cajas de texto, definidas con `<INPUT>`, tienen además estas propiedades:

- `maxLength`: Nos permite obtener y modificar la longitud máxima del campo. Si no se ha especificado esta propiedad, se devuelve -1.
- `size`: Devuelve el número de caracteres visibles en el campo. También podremos cambiarlo directamente en nuestro código.

Como no podía ser de otra manera, los `<TEXTAREA>` también tienen algunas propiedades suyas:

- `cols, rows`: Nos dejan obtener o modificar el ancho o el alto del cuadro de texto. Si alguna de estas propiedades no ha sido definida, se obtiene un valor -1.

Veamos cómo usar todas estas propiedades con nuestro ejemplo, tomando algunos campos cuya propiedad `type` sea de uno de estos tipos.

```
<SCRIPT TYPE="text/javascript">
   var formulario1 = document.forms[0];
   var formulario2 = document.forms[1];
   var formulario3 = document.forms[2];
   // Muestra las propiedades de un campo de entrada de datos
   function mostrarPropiedades(campo) {
      var datos = "El campo " + campo.name + " está
      en el formulario " + campo.form.name + " y
      sus propiedades son:\n";
      datos += "\t- ID: " + campo.id + "\n";
      datos += "\t- Nombre: " + campo.name + "\n";
      datos += "\t- Tipo: " + campo.type + "\n";
      datos += "\t- Valor: " + campo.value + "\n";
      datos += "\t- Deshabilitado: " + campo.disabled +
      "\n";
      datos += "\t- Sólo lectura: " + campo.readOnly + "\n";
      // Añadir los campos específicos del tipo de campo
      if (campo.type == "textarea") {
         datos += "\t- Ancho: " + campo.cols + "\n";
         datos += "\t- Alto: " + campo.rows + "\n";
      } else {
         datos += "\t- Longitud máxima: " + campo.
         maxLength + "\n";
         datos += "\t- Ancho: " + campo.size + "\n";
      }
      // Mostrar las propiedades
      alert(datos);
   }
   // Mostramos algunos campos
   mostrarPropiedades(formulario1.
   elements["nombreUsuario"]);
   mostrarPropiedades(formulario1.elements["clave"]);
   mostrarPropiedades(formulario2.elements["fichero"]);
   mostrarPropiedades(formulario2.elements["descripcion"]);
   mostrarPropiedades(formulario3.elements["fase"]);
   mostrarPropiedades(formulario3.elements["importe"]);
</SCRIPT>
```

Truco: *A través de JavaScript, la propiedad `type` no es exclusiva de los campos <INPUT>, sino que nos dice el tipo de cualquier campo.*

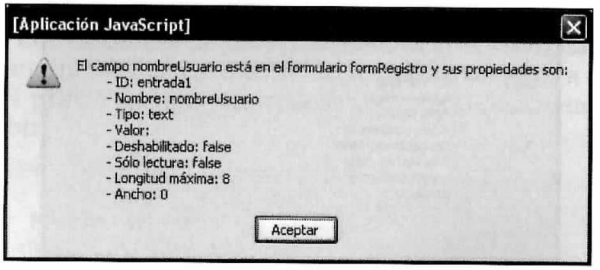


Figura 9.12. Información del campo nombreUsuario.

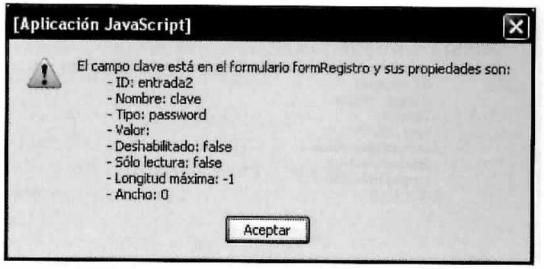


Figura 9.13. Información del campo clave.

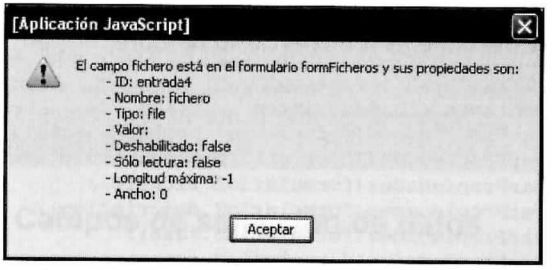


Figura 9.14. Información del campo fichero.

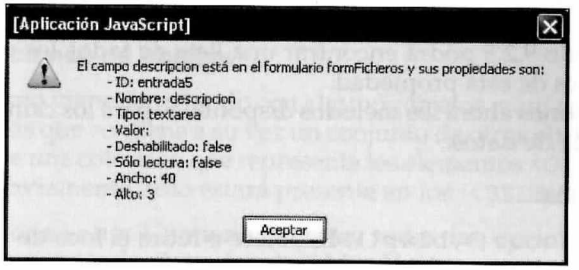


Figura 9.15. Información del campo descripcion.

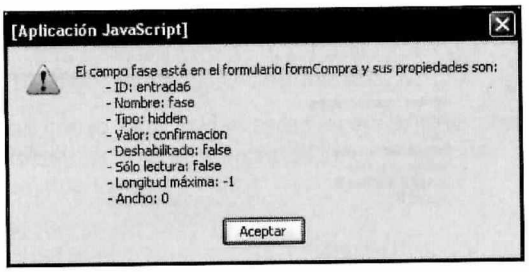


Figura 9.16. Información del campo fase.

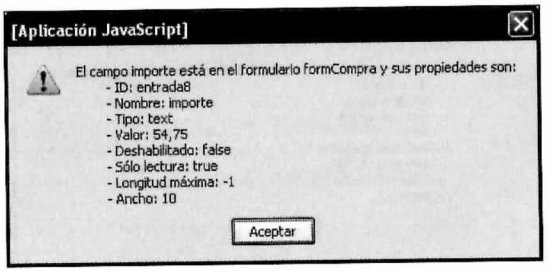


Figura 9.17. Información del campo importe.

Recuerde que también es válido acceder a un campo o formulario directamente a través de su nombre:

```
<SCRIPT TYPE="text/javascript">
   // Mostramos algunos campos
   mostrarPropiedades(formulario1.nombreUsuario);
   mostrarPropiedades(document.formRegistro.clave);
   mostrarPropiedades(formulario2.fichero);
   mostrarPropiedades(formulario2.descripcion);
   mostrarPropiedades(formulario3.fase);
   mostrarPropiedades(formulario3.importe);
</SCRIPT>
```

Se habrá dado cuenta de que hemos distinguido los diferentes tipos de campos a través de la propiedad `type`. En el apartado 9.2.5 podrá encontrar una lista de todos los valores posibles de esta propiedad.

Veamos ahora los métodos disponibles para los campos de entrada de datos.

Métodos

- `focus()`, `blur()`: Establece o retira el foco de la caja.
- `select()`: Selecciona todo el texto que esté escrito dentro de la caja de texto.

Con un sencillo código podremos hacer que el cursor se sitúe en cualquiera de las cajas (establecer el foco) nada más cargarse la página. De este modo forzamos un poco a que ese sea el primer campo que rellene usuario cuando empiece a escribir.

```
<SCRIPT TYPE="text/javascript">
   var formulario1 = document.forms[0];
   // Ponemos el foco en el email
   formulario1.elements["email"].focus();
</SCRIPT>
```

Mediante este otro script, seleccionaremos todo el texto que haya en el `<TEXTAREA>` una vez hayan pasado 10 segundos desde que se cargó la página. Si escribimos algo en él antes de que ocurra, veremos el efecto.

```
<SCRIPT TYPE="text/javascript">
   // Función que selecciona un campo
   function seleccionarCampo(campo) {
      campo.select();
   }
   // Inicialización de temporizador
   setTimeout("seleccionarCampo(document.forms[1].
   elements['descripcion'])", 10000);
</SCRIPT>
```

> **Nota**: *Aquí se han utilizado las comillas simples para indicar el nombre del campo simplemente por comodidad y evitar poner la comilla doble con el carácter de escape. Sería igualmente válido escribirlo como* `['\"descripcion\"']`.

9.2.3. Campos de selección de datos

Estos campos pueden ser controlados directamente desde el código, lo cual nos aportará una gran flexibilidad a la hora de validar nuestros formularios.

Colecciones de objetos

Como viene ocurriendo con algunos objetos, aquí tenemos uno más que contiene a su vez un conjunto de otros objetos. Se trata de una colección que representa los elementos `<OPTION>` que, obviamente, sólo estará presente en los `<SELECT>`.

- `options`: Contiene un *array* todas las opciones que muestra la lista desplegable. Únicamente se puede acceder a cada opción a través de un índice.

```
<SCRIPT TYPE="text/javascript">
   // Acceso a la colección de opciones de una
   // lista desplegable
   var campoSelect = document.forms[0].elements[3];
   var unaOpcion = campoSelect.options[0];
</SCRIPT>
```

Además de obtener cada opción por separado, podremos saber el número de todas ellas, ya que se trata de un `array`:

```
<SCRIPT TYPE="text/javascript">
   // Cantidad de opciones en el campo select
   var campoSelect = document.forms[0].elements[3];
   alert("El campo select " + campoSelect.name +
   " tiene " + campoSelect.options.length + " opciones");
   // Mostrará 3 opciones
</SCRIPT>
```

Propiedades

Todos los elementos HTML de selección que tenemos a nuestra disposición (`<INPUT>`, `<SELECT>` y `<OPTION>`) disponen de unas cuantas propiedades comunes:

- `id`: Equivale al identificador del campo (atributo `ID`).
- `name`, `type`: Representan el nombre y el tipo del elemento (atributos `NAME` y `TYPE`), excepto para los `<OPTION>`, que no disponen de estos atributos.
- `value`: Nos devuelve el valor que tenga asociado el campo en su atributo `VALUE`. En el caso de los `<SELECT>` nos devolverá el valor de la opción que esté seleccionada.
- `disabled`: Nos permite habilitar o deshabilitar el campo, modificando el atributo `DISABLED`.
- `form`: Nos devuelve el objeto de formulario en el que se encuentra el campo.

Adicionalmente, los dos tipos de elementos `<INPUT>` que pueden existir en un formulario (`radio` y `checkbox`) tienen estas propiedades:

- `checked`: Permite saber y establecer cuándo está seleccionado el campo.

Por otro lado, las listas desplegables (`<SELECT>`) también nos ofrecen un puñado más de propiedades, que además son bastante útiles:

- `multiple`: Nos permite conocer y establecer cuándo la lista admite selección múltiple.

- `size`: Nos devuelve el valor del atributo `SIZE`, que viene a ser el número de opciones visibles al mismo tiempo en la lista desplegable.
- `length`: Nos indica el número de opciones de las que dispone. Este valor equivale a la longitud de su colección de datos, es decir, `options.length`.
- `selectedIndex`: Nos permite obtener o establecer la opción que está seleccionada, a través de un índice que corresponde a su posición dentro de la lista. El índice 0 representa la primera opción en el orden que están definidas en la página HTML. La última posición se puede calcular restando 1 a su longitud (número de opciones). Esta propiedad toma el valor especial -1 cuando no hay ninguna opción seleccionada.

Por último, no nos olvidamos de los elementos `<OPTION>`, que son muy envidiosos y también tienen sus propiedades aparte:

- `selected`: Nos permite saber o fijar cuándo la opción estará seleccionada.
- `text`: Obtiene el texto que se encuentra junto a la etiqueta `<OPTION>` y también nos deja modificarlo.
- `index`: Devuelve la posición que ocupa la opción dentro de la lista desplegable.

A continuación dispone de un *script* que mostrará todas estas propiedades:

```
<SCRIPT TYPE="text/javascript">
   var formulario1 = document.forms[0];
   var formulario2 = document.forms[1];
   var formulario3 = document.forms[2];
   // Muestra las propiedades de un campo de selección
   // de datos
   function mostrarPropiedades(campo) {
      var datos = "El campo " + campo.name +
      " está en el formulario " + campo.form.name + " y
      sus propiedades son:\n";
      datos += "\t- ID: " + campo.id + "\n";
      // Los option no tienen estas propiedades
      if (typeof(campo.type) != "undefined") {
         datos += "\t- Nombre: " + campo.name + "\n";
         datos += "\t- Tipo: " + campo.type + "\n";
      }
      datos += "\t- Deshabilitado: " + campo.disabled +
      "\n";
      // Añadir los campos específicos del tipo de campo
```

```
    if (campo.type == "select-one" || campo.type ==
    "select-multiple") {
       datos += "\t- Opciones visibles: " + campo.size
       + "\n";
       datos += "\t- Opciones totales: " + campo.length
       + "\n";
       datos += "\t- Multiple: " + campo.multiple +
       "\n";
       datos += "\t- Opción seleccionada: " + campo.
       selectedIndex + "\n";
    } else if (campo.type == "radio" || campo.type ==
    "checkbox") {
       datos += "\t- Valor: " + campo.value + "\n";
       datos += "\t- Seleccionado: " + campo.checked +
       "\n";
    } else {
       datos += "\t- Texto: " + campo.text + "\n";
       datos += "\t- Posición: " + campo.index + "\n";
       datos += "\t- Seleccionado: " + campo.selected +
       "\n";
    }
    // Mostrar las propiedades
    alert(datos);
  }
  // Mostramos algunos campos
  mostrarPropiedades(formulario1.elements["pais"]);
  mostrarPropiedades(formulario1.elements["pais"].
  options[0]);
  mostrarPropiedades(formulario1.elements["pais"].
  options[1]);
  mostrarPropiedades(formulario2.elements["comprimir"]);
  mostrarPropiedades(formulario3.elements["pago"][0]);
  mostrarPropiedades(formulario3.elements["pago"][2]);
</SCRIPT
```

Nota: *Fíjese que la propiedad* `name` *para una opción devuelve el valor* `undefined` *al no tenerla disponible.*

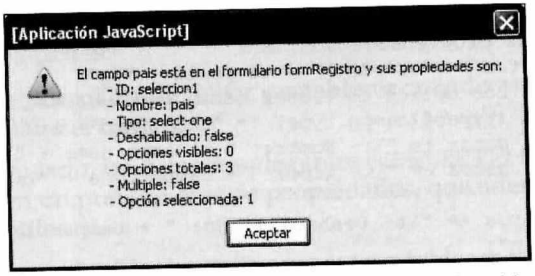


Figura 9.18. Información de una lista desplegable.

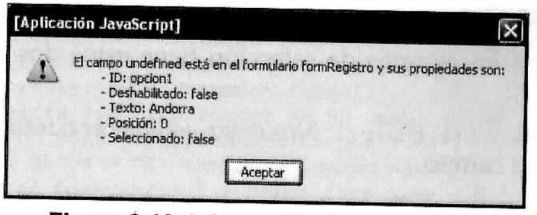


Figura 9.19. Información de una opción.

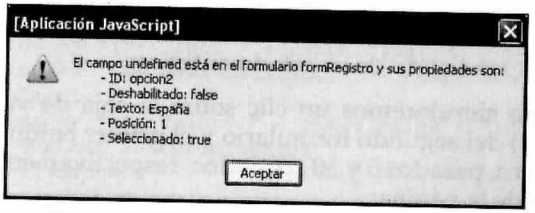


Figura 9.20. Información de una opción.

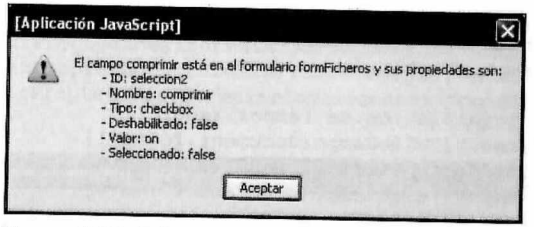


Figura 9.21. Información de un campo checkbox.

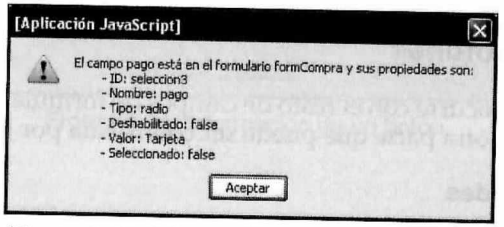


Figura 9.22. Información de un campo radio.

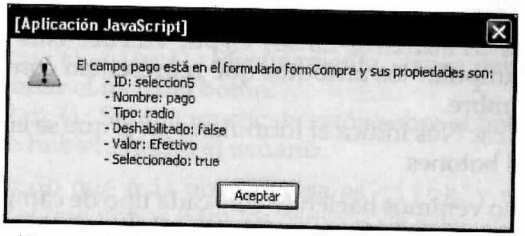


Figura 9.23. Información de un campo radio.

Métodos

Todos los campos de selección tiene estos dos métodos definidos:

- `focus()`, `blur()`: Nos dejan establecer o retirar el foco del campo.

Los `<INPUT>` disponen además de otro adicional:

- `click()`: Simula un clic de ratón sobre el campo, de forma que lo dejará seleccionado o deseleccionado dependiendo de su estado anterior.

Ahora simularemos un clic sobre la caja de validación (*checkbox*) del segundo formulario y el primer botón de radio del tercero, pasados 5 y 10 segundos, respectivamente, desde la carga de la página:

```
<SCRIPT TYPE="text/javascript">
   // Función que simula un clic sobre un campo
   function clicCampo(campo) {
      campo.click();
   }
   // Inicialización de temporizador
   setTimeout("clicCampo(document.forms[1].
   elements['comprimir'])", 5000);
   setTimeout("clicCampo(document.forms[2].
   elements['pago'][0])", 10000);
</SCRIPT>
```

9.2.4. Botones

Como ocurre con el resto de campos de formulario, los botones son otra parte que puede ser controlada por JavaScript.

Propiedades

Por una vez, ¡todos estos elementos tienen las mismas propiedades!

- `id`, `disabled`, `name`, `type`, `value`: Nos permiten manipular los atributos que representan con el mismo nombre.
- `form`: Nos indica el formulario en el que se encuentran los botones.

Y como venimos haciendo con cada tipo de campo, vamos a utilizarlas dentro de un *script*:


```
<SCRIPT TYPE="text/javascript">
   var formulario1 = document.forms[0];
   var formulario2 = document.forms[1];
   var formulario3 = document.forms[2];
   // Muestra las propiedades de un campo botón
   function mostrarPropiedades(campo) {
      var datos = "El campo " + campo.name + " está
      en el formulario " + campo.form.name + " y
      sus propiedades son:\n";
      datos += "\t- ID: " + campo.id + "\n";
      datos += "\t- Nombre: " + campo.name + "\n";
      datos += "\t- Tipo: " + campo.type + "\n";
      datos += "\t- Deshabilitado: " + campo.disabled +
      "\n";
      datos += "\t- Valor: " + campo.value + "\n";
      // Mostrar las propiedades
      alert(datos);
   }
   // Mostramos algunos campos
   mostrarPropiedades(formulario1.elements["limpiar"]);
   mostrarPropiedades(formulario1.elements["enviar"]);
   mostrarPropiedades(formulario2.elements["limpiar"]);
   mostrarPropiedades(formulario2.elements["enviar"]);
   mostrarPropiedades(formulario3.elements["confirmar"]);
</SCRIPT>
```

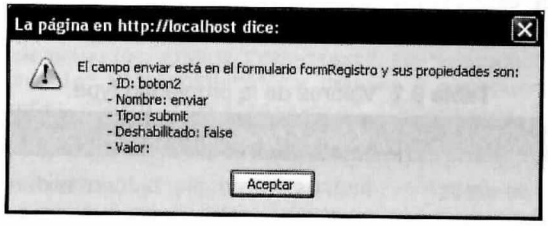


Figura 9.24. Información de un botón.

Métodos

En cuanto a los métodos que podremos utilizar con los botones, no hay ninguno nuevo, así que ya le empezarán a resultar incluso familiares.

- `focus()`, `blur()`: Efectivamente, sirven para colocar o quitar el foco del botón.
- `click()`: Simula un clic de ratón sobre el botón como si lo hubiera hecho el usuario.

El método que más nos interesa es `click`, y su uso es igual que el que vimos para los campos de selección. Con el siguiente código conseguiremos que se simule un clic sobre el

botón **Reestablecer** del primer formulario a los 10 segundos de haberse cargado la página. Esto hará que se borre toda la información que hubiéramos escrito en él, por lo que nuevamente tendremos que rellenar algunos campos previamente para poder ver su efecto.

```
<SCRIPT TYPE="text/javascript">
   // Función que simula un clic sobre un campo botón
   function clicCampo(campo) {
      campo.click();
   }
   // Inicialización de temporizador
   setTimeout("clicCampo(document.forms[0].
   elements['limpiar'])", 10000);
</SCRIPT>
```

9.2.5. Resumen de tipos de campos

Al igual que se hizo para los valores del atributo `TYPE`, la tabla 9.2 le permitirá tener a mano los valores que utiliza JavaScript para identificar todos los posibles valores de la propiedad `type` de los campos de un formulario. Tenga en cuenta que los elementos `<INPUT>` no son los únicos que disponen de esta propiedad, por lo que tendremos más valores que en la tabla 9.1.

Tabla 9.2. Valores de la propiedad type.

Campo	*Valores type*
Entrada de datos	text, password, file, hidden, textarea
Selección de datos	radio, checkbox, select-one (sin atributo MULTIPLE), select-multiple (con atributo MULTIPLE)
Botones	button, submit, reset

9.3. Validar datos de un formulario

Ahora que ya hemos visto cómo se crea un formulario HTML y cómo acceder a todos sus elementos y propiedades a través de JavaScript, vamos a aplicar todo esto en un caso real y de los más comunes: la validación de un formulario para detectar posibles errores en los datos antes de mandarlos al servidor.

9.3.1. Definir formulario

Para ir poco a poco con este ejemplo, vamos a definir primero un formulario que requiera registrarse como usuario, ya que es uno de los más típicos con los que nos podemos encontrar en una Web. Los datos que se piden normalmente son:

- Nombre de usuario, con el cual nos identificaremos en la página.
- Contraseña, que junto con el nombre de usuario nos permitirá acceder a nuestra cuenta.
- Email, para recibir noticias o mensajes.
- Idioma, por si la página está disponible en varios idiomas.
- Una caja de validación para autorizar al sitio Web a mandarnos publicidad mensual a nuestro email.

El código HTML con todos esos campos quedaría similar a esto:

```
<HTML>
<BODY>
<H3>Formulario para darse de alta como usuario en
nuestra Web</H3>
<FORM ID="formulario" NAME="formRegistro"
ACTION="registrar.php" METHOD="post">
Nombre de usuario: <INPUT TYPE="text" ID="entrada1"
NAME="usuario" MAXLENGTH="8" /> (Máximo 8 caracteres)<BR />
Contraseña: <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
Repetir contraseña: <INPUT TYPE="password" ID="entrada3"
NAME="claveRepe" /> (Deben coincidir)<BR />
Email: <INPUT TYPE="text" ID="entrada4" NAME="email"
SIZE="40" /><BR />
Idioma: <SELECT ID="seleccion1" NAME="idioma">
      <OPTION ID="opcion1" VALUE="EN">Inglés</OPTION>
      <OPTION ID="opcion2" VALUE="ES" SELECTED>Español</
OPTION>
      <OPTION ID="opcion3" VALUE="DE" >Alemán</OPTION>
</SELECT><BR />
<INPUT TYPE="checkbox" ID="seleccion2" NAME="publicidad"
/> Quiero recibir publicidad mensual en mi email
</FORM>
</BODY>
</HTML>
```

Opcionalmente, este formulario se puede complementar con un botón que reestablezca todo los campos, es decir, un botón de tipo `reset`.

Para comprobar que los campos del formulario no contienen errores, vamos a crear una función JavaScript por cada uno de ellos.

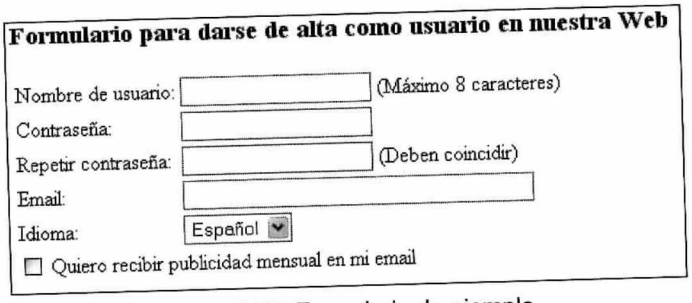


Figura 9.25. Formulario de ejemplo.

9.3.2. Validación del nombre de usuario

Para confirmar que este campo es válido, simplemente vamos a mirar que su valor no esté vacío. No sería necesario comprobar que la longitud es correcta (máximo 8 caracteres) puesto que esa limitación la hace el propio formulario al incluir el atributo `MAXLENGTH` en este campo. Por tanto, nuestra función devolverá un valor `true` si el campo no es vacío y `false` en caso contrario.

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba el nombre de usuario
   function comprobarUsuario(nombreUsuario) {
      if (nombreUsuario != "") {
         return true;
      } else {
         return false;
      }
   }
</SCRIPT>
```

Una forma más abreviada de definir esta función sería utilizar el valor devuelto por la propia comparación con la cadena vacía:

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba el nombre de usuario
   function comprobarUsuario(nombreUsuario) {
      return (nombreUsuario != "");
   }
</SCRIPT>
```


Esto no es importante ahora. Simplemente quería mostrarle las distintas formas que tenemos de realizar la misma operación.

9.3.3. Validación de la contraseña

Este campo habrá dos comprobaciones:

1. El valor de los dos campos contraseña no están vacíos.
2. Los valores de ambos campos coinciden.

Vamos a plasmar estas validaciones en forma de código JavaScript:

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba la contraseña
   function comprobarClave(clave, claveRepetida) {
      if (clave != "" && claveRepetida != "") {
         return (clave == claveRepetida);
      } else {
         return false;
      }
   }
</SCRIPT>
```

Como puede ver, en primer lugar se comprueba que ninguno de los dos campos esté vacío, devolviendo `false` si esto no fuese así, y a continuación se comparan sus valores devolviéndose como resultado.

Tanto para el nombre de usuario como para la contraseña se pueden aplicar muchas más validaciones, que se dejan a elección del desarrollador de la página. Por ejemplo, se podría restringir que el nombre de usuario sólo pudiera contener números y letras mediante una expresión regular, o que la contraseña deba contener mayúsculas y minúsculas obligatoriamente. Las posibilidades con JavaScript son muy amplias.

9.3.4. Validación del email

Con este campo vamos a ir un poco más allá y utilizaremos cosas que ya hemos visto en esta guía. Aparte de comprobar si el campo está vacío, validaremos también su formato a través de una expresión regular, por lo que usaremos uno de los patrones que expusimos en su momento. Con esto evitaremos que un usuario introduzca un texto que no corresponda a una dirección de correo electrónico.

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba el email
   function comprobarEmail(email) {
      var patron = new RegExp("^\\w+@\\w+\\.\\w{2,3}$");
      if (email != "") {
         return (patron.test(email));
      } else {
         return false;
      }
   }
</SCRIPT>
```

9.3.5. Validación del idioma

Dado que en este campo los valores siempre van a ser correctos, ya que se seleccionan de una lista, tan sólo debemos comprobar que se ha escogido una de esas opciones disponibles. La forma más sencilla de comprobar esto es consultar directamente la propiedad `selectedIndex` del campo `<SELECT>`. Si es distinto de -1 entonces lo daremos por válido.

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba el idioma
   function comprobarIdioma(indiceIdioma) {
      return (indiceIdioma != -1);
   }
</SCRIPT>
```

Como ha podido observar, esta función resulta bastante sencilla.

Otra forma de ver esto es mirar todas las opciones y buscar alguna que tenga su propiedad `selected` con valor `true`. Si quisiéramos hacerlo así, tendríamos que pasar directamente el campo `<SELECT>` como parámetro y recorrer su colección de opciones:

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba el idioma
   function comprobarIdioma(campoIdioma) {
      for(var i=0; i<campoIdioma.length; i++) {
         if (campoIdioma.options[i].selected) {
            return true;
         }
      }
      return false;
   }
</SCRIPT>
```


9.3.6. Validación del envío de publicidad

En este caso no hace falta comprobar si la caja está seleccionada o no porque se trata de un campo opcional (el usuario acepta o no recibir publicidad). Por tanto, para hacer algo más interesante este ejemplo y aplicar los conocimientos adquiridos, vamos a hacer que aparezca un cuadro de diálogo dándole una segunda oportunidad para que acepte la publicidad si el usuario no lo ha hecho antes.

Como siempre, esta función se puede programar de mil formas distintas. Nosotros vamos a crearla de forma que devolverá un valor `true` o `false` que nos servirá más adelante para fijar la propiedad `checked`, haciendo que se acepte o no el envío de publicidad.

```
<SCRIPT TYPE="text/javascript">
   // Función que comprueba la publicidad
   function comprobarPublicidad(aceptaPublicidad) {
      if (!aceptaPublicidad) {
         return confirm("Nuestra publicidad le mantendrá
         informado sobre las ofertas y novedades.
         \n¿Desea recibirla en su email?");
      } else {
         return true;
      }
   }
</SCRIPT>
```

9.3.7. Función principal de validación

Bueno, llega la hora de juntar todas estas funciones en una sola para validar nuestro formulario.

Como hemos definido las funciones para que devuelvan un valor booleano, utilizarlas será bastante fácil. Basta con comprobar su resultado y mostrar un mensaje de advertencia solamente cuando el campo no sea válido. Para darle un toque más profesional a nuestra función, además del mensaje, situaremos el foco sobre el campo que lo ha provocado, con lo que ayudaremos al usuario a subsanar el error más rápidamente.

Dicho esto, vamos con la función:

```
<SCRIPT TYPE="text/javascript">
   // Función que valida el formulario de registro
   function validarRegistro() {
      with(document.formRegistro) {
         // Comprobar campos obligatorios
```

```
        if (!comprobarUsuario(usuario.value)) {
           alert("El nombre de usuario está vacío.");
           usuario.focus();
        } else if (!comprobarClave(clave.value,
        claveRepe.value)) {
           alert("La contraseña está vacía o no está
           bien repetida.");
           clave.value = "";
           claveRepe.value = "";
           clave.focus();
        } else if (!comprobarEmail(email.value)) {
           alert("El email está vacío o no es válido.");
           email.select();
           email.focus();
        } else if (!comprobarIdioma(idioma.
        selectedIndex)) {
           alert("No hay un idioma seleccionado.");
           idioma.focus();
        }
        // Comprobar campo opcional
        publicidad.checked =
        comprobarPublicidad(publicidad.checked);
     }
  }
</SCRIPT>
```

Notará que el campo `publicidad` se trata de manera distinta al resto. Esto es porque se trata de un dato opcional y no requiere validación. Simplemente intentamos convencer al usuario para que seleccione esa casilla por lo que, tanto si nuestro mensaje surte efecto como si no, la propiedad `checked` se verá modificada con un nuevo valor booleano proveniente de la función `comprobarPublicidad` (ahora comprenderá mejor por qué hicimos así la función).

9.3.8. Ejecutar la validación

La manera más común de ejecutar una acción sobre un formulario es a través de alguno de sus botones, detectando cuándo se hace clic sobre él. Lamentablemente aún no hemos explicado cómo detectar estas acciones, que se conocen como eventos, aunque lo haremos en el capítulo siguiente como complemento de este. De todas formas no se preocupe que en JavaScript hay solución para (casi) todo. Vamos a solventar esta pequeña falta de conocimientos actual con algo que también le resultará útil: ejecutar funciones JavaScript desde hipervínculos o enlaces HTML.

Para hacer esto debemos utilizar el atributo `HREF` junto con una función en lugar de un URL, anteponiendo la cadena "javascript:" a la llamada de la función. La función se utiliza exactamente igual que dentro de un bloque `<SCRIPT>`, por lo que podrá incluir parámetros o llamadas a otras funciones.

Veamos un ejemplo, utilizando nuestra función para validar el formulario completo:

```
<A HREF="javascript:validarRegistro()">Validar
formulario</A>
```

Sencillo, ¿verdad? El resultado de hacer clic sobre este enlace será la ejecución de la función `validarRegistro`.

También podemos añadir otros enlaces que utilicen las funciones de validación de cada campo a modo de prueba. En estos casos, pasaremos el valor de un campo del formulario o uno escrito directamente (para que vea cómo se hace el paso de parámetros).

```
<A HREF="javascript:alert(comprobarUsuario(''))">Validar
usuario vacío</A>
<A HREF="javascript:alert(comprobarUsuario(document.
formRegistro.usuario.value))">Validar usuario del
formulario</A>
<A HREF="javascript:alert(comprobarClave('aa',
\"bb\"))">Validar contraseñas "aa" y "bb"</A>
<A HREF="javascript:alert(comprobarClave(document.
formRegistro.clave.value, document.formRegistro.
claveRepe.value))">Validar contraseña del formulario</A>
<A HREF="javascript:alert(comprobarEmail('nombre@
dominio'))">Validar email "nombre@dominio"</A>
<A HREF="javascript:alert(comprobarEmail('nombre@dominio.
com'))">Validar email "nombre@dominio.com"</A>
<A HREF="javascript:alert(comprobarEmail(document.
formRegistro.email.value))">Validar email
del formulario</A>
<A HREF="javascript:alert(comprobarPublicidad(false))">
Validar publicidad false</A>
<A HREF="javascript:alert(comprobarPublicidad(document.
formRegistro.publicidad.checked))">Validar publicidad
del formulario</A>
```

Ahora es cuando le animo, si no lo ha hecho ya, a escribir todo este ejemplo completo en su ordenador y probarlo. También puede añadir otros campos o modificar los arriba expuestos. Todo lo que practique ahora le vendrá bien para después hacer sus *scripts* mucho más completos y fiables.

10 Eventos

Un evento es una acción que realiza el usuario sobre algún elemento de nuestra página al interactuar con él, como por ejemplo pasar el puntero del ratón por encima de una imagen o hacer clic sobre un botón.

Gracias a los eventos, la interactividad de nuestra página aumenta considerablemente ya que podremos hacer que reaccione ante las acciones del usuario. Esta característica es especialmente explotada por DHTML (*Dynamic HTML*) para crear variedad de efectos y situaciones.

JavaScript es capaz de detectar estos eventos y a la vez nos permite asociarles unas instrucciones que se ejecutarán cuando se produzcan.

10.1. Eventos en JavaScript

En JavaScript existe una amplia variedad de eventos que podremos utilizar a nuestro antojo para crear las situaciones que queramos o conseguir un efecto concreto.

La tabla 10.1 le mostrará todos los eventos y la acción que los genera.

Tabla 10.1. Lista de eventos en JavaScript.

Evento	*Origen*
Abort	La carga de una imagen es interrumpida (detenemos la carga de la página, nos vamos a otra...).
Load	El navegador termina de cargar una página.

Evento	Origen
Unload	Se abandona la página actual (cerramos el navegador o nos vamos a otra).
Error	Se produce un error durante la carga de un documento o imagen.
Resize	Se modifica el tamaño de una ventana o marco.
Blur	Un elemento de la página pierde el foco.
Focus	Un elemento recibe el foco.
Change	El estado o valor de un elemento cambia. Normalmente se origina después de que el elemento haya perdido el foco (Blur).
Select	Se selecciona un elemento de la página.
KeyDown	Una tecla es pulsada sin soltarla.
KeyUp	Una tecla que estaba pulsada es soltada.
KeyPress	Una tecla es pulsada. Se genera después de KeyDown.
Click	Se hace clic sobre un elemento.
DblClick	Se hace doble clic sobre un elemento.
MouseDown	Se pulsa un botón cualquiera del ratón.
MouseUp	Se libera un botón del ratón que estaba pulsado.
MouseMove	Se mueve el puntero por la pantalla.
MouseOver	El puntero entra en el área que ocupa un elemento (se pasa por encima).
MouseOut	El puntero sale del área que ocupa un elemento (se quita de encima).
Submit	Un formulario está a punto de ser enviado.
Reset	Un formulario es limpiado o reiniciado.

10.2. Eventos en una página HTML

De momento ya conocemos los eventos que tenemos disponibles pero, ¿dónde puedo usar cada uno de ellos? Esto es lo que vamos a desvelar con la tabla 10.2, donde están indicados los principales elementos HTML que desencadenan cada evento.

Tabla 10.2. Elementos HTML que generan eventos.

Evento	Elementos
abort	<img>
load	<body>
unload	<body>
error	<body>, <img>
resize	<body>
blur	<body>, <input>, <select>, <textarea>
focus	<body>, <input>, <select>, <textarea>
change	<input>, <select>, <textarea>
select	<input>, <textarea>
keydown	Elementos de entrada de datos de formulario.
keyup	Elementos de entrada de datos de formulario.
keypress	Elementos de entrada de datos de formulario.
click	Elementos de selección de datos y botones de formulario, <a>, <img>
dblclick	Elementos de selección de datos y botones de formulario, <a>, <img>
mousedown	Elementos de selección de datos y botones de formulario, <a>, <img>
mouseup	Elementos de selección de datos y botones de formulario, <a>, <img>
mousemove	Elementos de formulario, <a>, <img>
mouseover	Elementos de formulario, <a>, <img>
mouseout	Elementos de formulario, <a>, <img>
submit	<form>
reset	<form>

10.3. Trabajar con eventos

Un evento como tal, carece de utilidad por sí mismo, así que se hace necesario asociarles una función o código JavaScript que se ejecutará cuando se produzca dicho evento. A estas funciones o código se les conoce como manejadores de eventos, ya que realizan una acción específica para ese evento. Los nombres de los manejadores se definen anteponiendo la palabra "on"

al nombre del evento. De esta forma, el manejador del evento `click` será `onclick`.

Para asociar los manejadores a los eventos disponemos de varias opciones, gracias a la flexibilidad de JavaScript:

1. Como un atributo HTML del elemento.
2. A través de una propiedad JavaScript del objeto que representa el elemento.

Ahora veremos con detalle cada una de estas opciones.

10.3.1. Manejadores como atributos HTML

Se trata de la manera más sencilla de incluir manejadores de eventos en un elemento. Para ello, basta con añadir un atributo con el nombre del manejador que necesitemos y a continuación el código JavaScript que queramos que se ejecute. Veamos un pequeño ejemplo que muestra un mensaje cuando hacemos clic sobre un botón genérico:

```
<HTML>
<BODY>
<H3>Formulario que muestra un mensaje</H3>
<FORM ID="formulario1" NAME="formMensaje">
<INPUT TYPE="button" ID="boton" NAME="mensaje"
VALUE="Púlsame" ONCLICK="alert('¡Me has pulsado!');" />
</FORM>
</BODY>
</HTML>
```

> **Nota**: *El nombre del manejador se puede escribir usando indistintamente mayúsculas y minúsculas ya que el código HTML no hace distinción entre ambas.*

Si recuerda, hicimos algo similar cuando utilizamos el atributo `HREF` de la etiqueta `<A>` para ejecutar una función JavaScript. Otra forma de hacer esto mismo sería usando un manejador:

```
<HTML>
<BODY>
<A HREF="javascript:void(0)" ONCLICK="alert('¡Mensaje
generado!');">Ver mensaje</A>
</BODY>
</HTML>
```

En este caso hemos incluido el código en el manejador y hemos dejado el atributo `HREF` con un código JavaScript que no hace nada. También es posible que vea muchos ejemplos

en otras publicaciones que utilizan en `HREF` el carácter de almohadilla (#) a modo de ancla al inicio de la página.

También podremos utilizar en el manejador la sentencia `this`, con la que accederemos a las propiedades del objeto que contiene ese manejador y así usarlas en nuestro código. Esto es más bien un atajo para acceder al objeto, ya que también podríamos obtenerlo mediante el método `getElementById` del objeto `document` (apartado 8.7.3 para los olvidadizos como yo), pero así nos ahorramos esa búsqueda y hacemos que nuestro código resulte un poco más óptimo.

```
<HTML>
<BODY>
<H3>Formulario que muestra propiedades</H3>
<FORM ID="formulario1" NAME="formMensaje">
<INPUT TYPE="button" ID="boton1" NAME="botonId"
VALUE="Ver mi ID" ONCLICK="alert('Mi ID es ' + this.
id);" /><BR />
<INPUT TYPE="button" ID="boton2" NAME="botonName"
VALUE="Ver nombre formulario" ONCLICK="alert('Mi
formulario se llama ' + this.form.name);" />
</FORM>
</BODY>
</HTML>
```

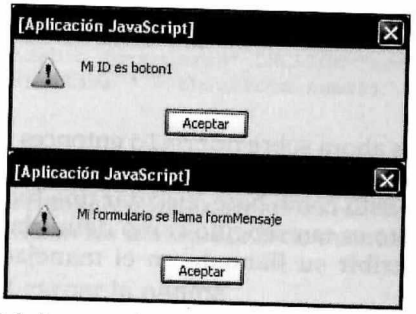


Figura 10.1. Acceso al objeto que contiene el manejador.

Con estos ejemplos podemos referirnos a los objetos que tiene asociado el manejador y acceder a sus propiedades o métodos. Si quisiéramos acceder a otro objeto debemos escribir el código de una forma normal. Veamos la manera de acceder a `botonName` desde `botonId`.

```
<HTML>
<BODY>
<H3>Formulario que muestra propiedades</H3>
```

```
<FORM ID="formulario1" NAME="formMensaje">
<INPUT TYPE="button" ID="boton1" NAME="botonId"
VALUE="Ver mi ID" ONCLICK="alert('El ID del otro botón
es ' + document.formMensaje.botonName.id);" /><BR />
<INPUT TYPE="button" ID="boton2" NAME="botonName"
VALUE="Ver nombre formulario" />
</FORM>
</BODY>
</HTML>
```

En este caso se mostrará el `ID` del `botonName`, que es "boton2".

Accediendo a los objetos de esta manera, no sólo podremos usar sus propiedades sino también sus métodos. Por ejemplo, en este caso vamos a simular que hacemos clic sobre el mismo botón.

```
<HTML>
<BODY>
<H3>Formulario que muestra propiedades</H3>
<FORM ID="formulario1" NAME="formMensaje">
<INPUT TYPE="button" ID="boton1" NAME="botonId"
VALUE="Ver mi ID" ONCLICK="document.formMensaje.
botonName.click();" /><BR />
<INPUT TYPE="button" ID="boton2" NAME="botonName"
VALUE="Ver nombre formulario" ONCLICK="alert('Mi
formulario se llama ' + this.form.name);" />
</FORM>
</BODY>
</HTML>
```

Si pulsamos ahora sobre `botonId` entonces se ejecutará el código asociado al manejador `ONCLICK` de `botonName`.

Tendiendo esto como base, ejecutar una función al originarse un evento es tan sencillo como definirla previamente y después escribir su llamada en el manejador que queramos.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra un mensaje
   function mostrarMensaje() {
      alert("¡Mensaje generado!");
   }
</SCRIPT>
</HEAD>
<BODY>
<A HREF="javascript:void(0)" ONCLICK="mostrarMensaje();">
Púlsame</A>
```



```
<FORM ID="formulario1" NAME="formMensaje">
<INPUT TYPE="button" ID="boton" NAME="mensaje"
VALUE="Pasa por encima" ONMOUSEOVER="mostrarMensaje();" />
</FORM>
</BODY>
</HTML>
```

Con este ejemplo conseguimos que se ejecute la función `mostrarMensaje` cuando se pulse el enlace y también cuando se pase el puntero del ratón por encima del botón.

Por supuesto, también es posible utilizar funciones con parámetros y asignarles valores usando la sentencia `this`.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra un texto
   function mostrarTexto(texto) {
      alert(texto);
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Formulario que muestra propiedades</H3>
<FORM ID="formulario1" NAME="formMensaje">
<INPUT TYPE="button" ID="boton1" NAME="botonId"
VALUE="Sumar" ONCLICK=" mostrarTexto(2+3);" /><BR />
<INPUT TYPE="button" ID="boton2" NAME="botonName"
VALUE="Ver nombre formulario" ONCLICK="mostrarTexto('Mi
formulario se llama ' + this.form.name);" />
</FORM>
</BODY>
</HTML>
```

Ahora veremos algunos ejemplos con situaciones típicas en las que se utilizan los manejadores de eventos.

Acciones al cargar la página

Este ejemplo trata de ejecutar una función cuando se complete la carga de la página que estamos viendo. Por tanto, el evento que nos interesa es `load`, siendo `ONLOAD` su manejador.

Nuestra página constará de un pequeño formulario y la función que asociaremos al manejador hará que dicho formulario se inicialice con unos valores por defecto. Es cierto que esto lo podríamos hacer directamente mediante atributos HTML, pero la finalidad de este ejemplo es que conozca el funcionamiento de los manejadores.

El formulario inicial sería este:

```
<H3>Formulario de conversión de moneda</H3>
<FORM ID="formulario1" NAME="formConversion">
<INPUT TYPE="text" ID="entrada1" NAME="cantidad" />
<SELECT ID="seleccion1" NAME="moneda1">
<OPTION VALUE="USD">Dólares</OPTION>
<OPTION VALUE="EUR">Euros</OPTION>
   <OPTION VALUE="GBP">Libras esterlinas</OPTION>
</SELECT> =>
<SELECT ID="seleccion2" NAME="moneda2">
   <OPTION VALUE="USD">Dólares</OPTION>
   <OPTION VALUE="EUR">Euros</OPTION>
   <OPTION VALUE="GBP">Libras esterlinas</OPTION>
</SELECT>
<INPUT TYPE="text" ID="entrada2" NAME="resultado" />
<BR /><BR />
<INPUT TYPE="submit" ID="boton1" NAME="botonConvertir" />
</FORM>
```

Después escribiremos la función JavaScript que se asociará posteriormente al manejador que nos interesa. Para evitar problemas, lo mejor es situarla dentro de <HEAD>.

```
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que inicializa un formulario
   function inicializarFormulario() {
      with(document.formConversion) {
         moneda1.selectedIndex = 1;
         moneda2.selectedIndex = 2;
         resultado.readOnly = true;
         botonConvertir.value = "Convertir >";
      }
   }
</SCRIPT>
</HEAD>
```

Por último, gracias al manejador ONLOAD colocado en la etiqueta <BODY> conseguiremos que:

1. Nuestras listas desplegables tengan unos valores concretos seleccionados.
2. La segunda caja de texto sea únicamente de lectura.
3. El botón de envío tenga el texto que nosotros queramos.

```
<BODY ONLOAD="inicializarFormulario();">
```

En las siguientes figuras puede ver la diferencia que existiría entre los formularios aplicando o no esta función.

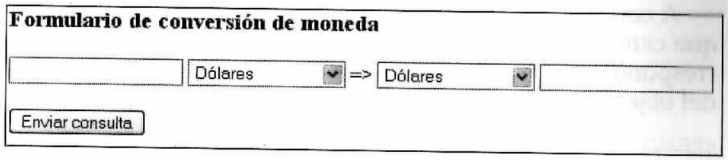


Figura 10.2. Formulario sin manejador onload.

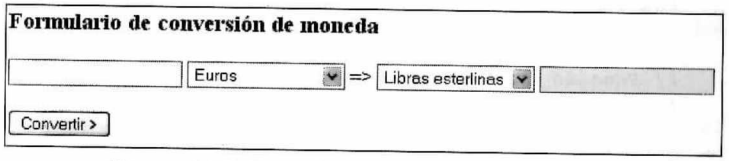


Figura 10.3. Formulario con manejador onload.

Cambiar imagen al pasar por encima de ella (rollover)

Ahora veremos cómo crear un pequeño efecto que le dará un toque de profesionalidad a nuestra página. El efecto *rollover* consiste en que al pasar el puntero del ratón por encima de una imagen ésta cambie y que después vuelva a aparecer la original al retirar el puntero. Esto es algo más propio de DHTML que de JavaScript, pero así puede ir viendo todo el potencial que tiene por delante.

En este caso, nuestra página mostrará simplemente una imagen que incluye dos manejadores:

1. ONMOUSEOVER, que cambiará la imagen por otra alojada en el mismo servidor al poner el puntero encima de ella.
2. ONMOUSEOUT, que volverá a mostrar la imagen que estaba al cargar la página, una vez quitemos el puntero.

Para realizar este ejemplo vamos a trabajar con las siguientes imágenes:

Figura 10.4. Imágenes de ejemplo.

Al principio mostraremos una de ellas, quedando así el código HTML:

```
<H3>Mueve el puntero sobre la imagen</H3>
<IMG ID="imagen1" NAME="flecha" SRC="derecha.gif" />
```

A continuación declaramos, en el <HEAD>, las funciones que cambiarán la imagen cuando se provoque el evento correspondiente. Para ello, debemos utilizar la propiedad `src` del objeto de la imagen.

```
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra la imagen original
   function mostrarOriginal() {
      document.flecha.src = "derecha.gif";
   }
   // Función que cambia la imagen original
   function cambiarOriginal() {
      document.flecha.src = "izquierda.gif";
   }
</SCRIPT>
</HEAD>
```

Para terminar, asociaremos estas funciones a los manejadores de la imagen:

```
<IMG ID="imagen1" NAME="flecha" SRC="derecha.
gif" ONMOUSEOVER="cambiarOriginal();"
ONMOUSEOUT="mostrarOriginal();" />
```

Si cargamos esta página y pasamos el puntero por encima de la imagen veremos cómo se produce el efecto deseado. Como puede ver, no es en absoluto complicado y sin embargo eleva la categoría de nuestra página.

Por aplicar un poco lo que hemos ido aprendiendo, me voy a poner un poco quisquilloso y voy a preguntarle qué pasaría si en nuestra página necesitásemos hacer este *rollover* en varias partes. Tal y como tenemos escritas las funciones, necesitaríamos crearlas para cada imagen ya que su código apunta a una en concreto (en el ejemplo, la que tiene el nombre `flecha`). ¿No cree que existe una manera más óptima de escribir estas funciones para que nos valgan en cualquiera de las imágenes? ¡Acertó! (Y si no, no se preocupe que no me voy a enterar...) Tenemos algo tremendamente útil para solucionar esto: la sentencia `this`. Como esta sentencia hace referencia al objeto que contiene el manejador, no tenemos más que trasladarla como parámetro a las funciones que cambian la imagen para poder usarlas tantas veces como queramos.

```
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra la imagen original
   function mostrarOriginal(imagen) {
      imagen.src = "derecha.gif";
   }
   // Función que cambia la imagen original
   function cambiarOriginal(imagen) {
      imagen.src = "izquierda.gif";
   }
</SCRIPT>
</HEAD>
```

Por supuesto, también debemos cambiar levemente los manejadores:

```
<IMG ID="imagen1" NAME="flecha" SRC="derecha.gif"
ONMOUSEOVER="cambiarOriginal(this);" ONMOUSEOUT="mostrar
Original(this);" />
```

Y ahora ya podemos añadir este rollover todas las veces que queramos en la misma página:

```
<IMG ID="imagen2" NAME="otraFlecha" SRC="derecha.gif"
ONMOUSEOVER="cambiarOriginal(this);" ONMOUSEOUT="mostrar
Original(this);" />
<IMG ID="imagen3" NAME="flechita" SRC="derecha.gif"
ONMOUSEOVER="cambiarOriginal(this);" ONMOUSEOUT="mostrar
Original(this);" />
```

Operaciones sobre formularios

El incluir manejadores en elementos de un formulario es uno de las usos más comunes de JavaScript ya que nos ofrece un control sobre ellos que es impensable a través de código HTML.

Para empezar por algo sencillo, vamos a imaginar que tenemos un formulario con varios tipos de campos y queremos validarlos antes de enviar la información al servidor.

```
<H3>Formulario de acceso</H3>
<FORM ID="formulario1" NAME="formLogin" ACTION="login.php"
METHOD="post">
Usuario <INPUT TYPE="text" ID="entrada1" NAME="usuario"
MAXLENGTH="8" /><BR />
Contraseña <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
<INPUT TYPE="button" ID="boton1" NAME="botonValidar"
VALUE="Validar" /> <INPUT TYPE="submit" ID="boton2"
NAME="botonEnviar"/>
</FORM>
```

Después tendremos nuestra función que validará el formulario comprobando que sus dos campos tienen un valor asociado, es decir, no están vacíos:

```
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que valida el formulario
   function validarFormulario() {
      // Operaciones de validación
      with(document.formLogin) {
         if (usuario.value == "") {
            usuario.focus();
         } else if (clave.value == "") {
            clave.focus();
         } else {
            alert("¡Todo correcto! Puedes enviar el
            formulario.");
         }
      }
   }
</SCRIPT>
</HEAD>
```

Hasta ahora habíamos visto cómo salir del paso utilizando un hipervínculo y su atributo `HREF` para lanzar la validación, pero para hacerlo a través de un botón debemos usar el manejador `ONCLICK`. De este modo, nuestro código quedaría tan simple como:

```
<INPUT TYPE="button" ID="boton1" NAME="botonValidar"
VALUE="Validar" ONCLICK="validarFormulario();" />
```

De esta forma obligamos al usuario a pulsar dos botones para enviar la información: uno para validar y otro para enviar. Entonces, como extensión de este ejemplo, podríamos unificar las acciones de los dos botones, en uno sólo haciendo que se envíe el formulario únicamente si la validación ha sido satisfactoria. Para ello debemos hacer uso del método `submit` del formulario dentro de la función que comprueba los campos.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que valida y envía el formulario
   function validarYEnviarFormulario() {
      // Operaciones de validación
      with(document.formLogin) {
         if (usuario.value == "") {
            usuario.focus();
         } else if (clave.value == "") {
            clave.focus();
         } else {
            submit();
         }
      }
```



```
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Formulario de acceso</H3>
<FORM ID="formulario1" NAME="formLogin" ACTION="login.php"
METHOD="post">
Usuario <INPUT TYPE="text" ID="entrada1" NAME="usuario"
MAXLENGTH="8" /><BR />
Contraseña <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
<INPUT TYPE="button" ID="boton1" NAME="botonEnviar"
VALUE="Validar y enviar"
ONCLICK="validarYEnviarFormulario();" />
</FORM>
</BODY>
</HTML>
```

Con esta función conseguimos que sólo se envíe el formulario si los dos campos tienen algún texto escrito. En otro caso, se fija el foco en el campo que esté vacío para que el usuario lo rellene.

Otro lugar típico del formulario donde incluir un manejador de eventos, en concreto `ONCHANGE`, es en las listas de selección. Con esto podremos ejecutar unas acciones cuando se seleccione una de sus opciones. En estos casos se hace bastante uso de la sentencia `this`, sobre todo para obtener el índice o el valor de la opción seleccionada.

Suponiendo este pequeño formulario:

```
<H3>Adivina la mezcla de colores</H3>
<FORM ID="formulario1" NAME="formColores">
Rojo +
<SELECT ID="seleccion1" NAME="color">
   <OPTION VALUE="Verde">Verde</OPTION>
   <OPTION VALUE="Azul">Azul</OPTION>
   <OPTION VALUE="Amarillo">Amarillo</OPTION>
</SELECT>
= Naranja
</FORM>
```

Y creando esta función que comprueba si hemos seleccionado la opción correcta, mostrando un mensaje al usuario:

```
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que comprueba la opción seleccionada
   function comprobarMezcla(indiceColor) {
      if (indiceColor == 2) {
         alert("¡Muy bien!");
      } else {
```

```
        alert("Lo siento, no es este color…
        Inténtalo de nuevo.");
      }
   }
</SCRIPT>
</HEAD>
```

Podremos finalmente asociarla al manejador ONCHANGE para dotar de "inteligencia" a nuestro formulario:

```
<SELECT ID="seleccion1" NAME="color"
ONCHANGE="comprobarMezcla(this.selectedIndex)">
```

Si el usuario elije la opción correcta (Amarillo, opción con índice 2) se le mostrará un mensaje de enhorabuena. En otro caso, se le anima a intentarlo otra vez.

Para ir un paso más allá con este ejemplo, podríamos pasar a la función comprobarMezcla directamente el objeto que representa la opción, para así poder obtener tanto su índice como su valor y así mostrar un mensaje más completo. El *script* nos quedaría algo similar a esto:

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que comprueba la opción seleccionada
   function comprobarMezcla(color) {
      if (color.index == 2) {
         alert("¡Muy bien!");
      } else {
         alert("Lo siento, no es " + color.value + "…
         Inténtalo de nuevo.");
      }
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Adivina la mezcla de colores</H3>
<FORM ID="formulario1" NAME="formColores">
Rojo +
<SELECT ID="seleccion1" NAME="color"
ONCHANGE="comprobarMezcla(this.options[this.
selectedIndex])">
   <OPTION VALUE="Verde">Verde</OPTION>
   <OPTION VALUE="Azul">Azul</OPTION>
   <OPTION VALUE="Amarillo">Amarillo</OPTION>
</SELECT>
= Naranja
</FORM>
</BODY>
</HTML>
```


Más profesional así, ¿verdad? Gracias a la sentencia this hemos podido acceder al *array* de opciones de la lista y, al mismo tiempo, hemos obtenido el índice seleccionado para usarlo como índice de ese *array* y así conseguir la opción correspondiente. Es un poco más complejo de escribir, pero en algunas situaciones es una de las mejores elecciones.

10.3.2. Trabajar con eventos en JavaScript

Además de incluir manejadores de la manera que acabamos de ver, también es posible hacerlo directamente en nuestros *scripts* sin añadir atributos adicionales a los elementos HTML.

Los manejadores de cada objeto se utilizan como una propiedad más de éste, por lo que a estas alturas no le será muy difícil de comprender. En este caso, al manejador se le debe asignar siempre una función ya que si ponemos directamente una porción de código el resultado no será el esperado puesto que se ejecutará como parte del *script*. Veamos un ejemplo con el evento click.

```
<SCRIPT TYPE="text/javascript">
   // Función que muestra un mensaje
   function mostrarMensaje() {
      alert("¡Hola!");
   }
   // Asignación incorrecta de manejador
   objeto.onclick = alert("¡Hola!");
   // Asignaciones correctas de manejador
   objeto.onclick = mostrarMensaje;
   objeto.onclick = function mostrarMensaje() {
   alert("¡Hola!"); }
</SCRIPT>
```

En el código que acabamos de escribir verá que si asignamos directamente una llamada al método alert, el manejador no contendrá dicha instrucción sino su resultado (undefined para el caso de alert). Sin embargo, si asignamos una función o la definimos directamente sobre la marcha, será algo completamente válido y el manejador hará su papel tal y como esperamos.

> **Nota**: *Los nombres de los manejadores deben estar escritos siempre en minúsculas para trabajar con ellos desde código JavaScript.*

Algo que debe tener en cuenta al utilizar los manejadores de este modo es que las asignaciones deben hacerse al final de la página o después de que se haya cargado, para que los elementos HTML a los que hagamos referencia estén definidos.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra un mensaje
   function mostrarMensaje() {
      alert("¡Clic!");
   }
   document.formBotones.botonAntes.onclick =
   mostrarMensaje;
</SCRIPT>
</HEAD>
<BODY>
<H3>Pulsa en los botones</H3>
<FORM ID="formulario1" NAME="formBotones">
<INPUT TYPE="button" ID="boton1" NAME="botonAntes"
VALUE="Manejador erróneo" />
<INPUT TYPE="button" ID="boton2" NAME="botonDespues"
VALUE="Manejador válido" />
</FORM>
<SCRIPT TYPE="text/javascript">
   document.formBotones.botonDespues.onclick =
   mostrarMensaje;
</SCRIPT>
</BODY>
</HTML>
```

Si escribe este código y trata de pulsar en cada uno de los botones, comprobará que el primero no realiza ninguna acción, mientras que el segundo sí muestra el mensaje previsto. Esto es porque la asignación de la función `mostrarMensaje` al manejador `onclick` de `botonAntes` se hace al principio de la página donde, ni el botón ni el formulario estaban aún definidos en ella.

> **Nota:** *Un inconveniente que presenta esta manera de usar los manejadores, es que no podremos pasar parámetros a las funciones. La alternativa que nos queda es utilizar variables de ámbito global para poder recoger su valor dentro de las funciones asignadas a los eventos.*

Para utilizar la sentencia `this`, tendremos que hacer referencia a ella dentro de las funciones que asociemos al manejador de modo que `this` representará al objeto que ha

provocado el evento, tal y como ocurría al usarla dentro del atributo HTML.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra un ID
   function mostrarID() {
      alert("Mi ID es " + this.id);
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Pulsa en los botones</H3>
<FORM ID="formulario1" NAME="formBotones">
<INPUT TYPE="button" ID="boton1" NAME="botonId1"
VALUE="Mira mi ID" />
<INPUT TYPE="button" ID="boton2" NAME="botonId2"
VALUE="Mira mi ID" />
</FORM>
<SCRIPT TYPE="text/javascript">
   document.formBotones.botonId1.onclick = mostrarID;
   document.formBotones.botonId2.onclick = mostrarID;
</SCRIPT>
</BODY>
</HTML>
```

Este ejemplo muestra un poco todo lo que se ha ido diciendo hasta ahora:

- La función se asocia al manejador en la parte final de la página.
- La función hace referencia al objeto que provoca el evento y no siempre alude al mismo.

Para que vea mejor el contraste de incluir manejadores de esta forma, vamos a utilizar los mismos ejemplos que con los manejadores por HTML.

Acciones al cargar la página

Para refrescarle la memoria, aquí tratamos de ejecutar una función al completarse la carga de la página actual, utilizando el manejador `onload`.

La página contiene un formulario y la función lo inicializará con unos valores determinados.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
```

```
   // Función que inicializa un formulario
   function inicializarFormulario() {
      with(document.formConversion) {
         moneda1.selectedIndex = 1;
         moneda2.selectedIndex = 2;
         resultado.readOnly = true;
         botonConvertir.value = "Convertir >";
      }
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Formulario de conversión de moneda</H3>
<FORM ID="formulario1" NAME="formConversion">
<INPUT TYPE="text" ID="entrada1" NAME="cantidad" />
<SELECT ID="seleccion1" NAME="moneda1">
   <OPTION VALUE="USD">Dólares</OPTION>
   <OPTION VALUE="EUR">Euros</OPTION>
   <OPTION VALUE="GBP">Libras esterlinas</OPTION>
</SELECT> =>
<SELECT ID="seleccion2" NAME="moneda2">
   <OPTION VALUE="USD">Dólares</OPTION>
   <OPTION VALUE="EUR">Euros</OPTION>
   <OPTION VALUE="GBP">Libras esterlinas</OPTION>
</SELECT>
<INPUT TYPE="text" ID="entrada2" NAME="resultado" />
<BR /><BR />
<INPUT TYPE="submit" ID="boton1" NAME="botonConvertir" />
</FORM>
<SCRIPT TYPE="text/javascript">
   // Asignamos el manejador
   window.onload = inicializarFormulario;
</SCRIPT>
</BODY>
</HTML>
```

Para que la función se ejecute después de la carga de la página, debemos usar el manejador `onload` del objeto `window` y no de `document` como se podría pensar. En realidad podríamos haber asignado este manejador en el `<HEAD>` puesto que el objeto `window` ya existe en ese momento, pero es un caso excepcional y le recomiendo hacer estas asignaciones siempre al final para mantener una homogeneidad en su código.

Cambiar imagen al pasar por encima de ella (rollover)

Aquí se pretende crear un efecto de *rollover* mediante el uso de dos manejadores de un objeto de imagen:

1. `onmouseover`, para cambiar la imagen por otra al poner el puntero encima de ella.
2. `onmouseout`, para volver a mostrar la imagen que estaba, una vez quitemos el puntero.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que muestra la imagen original
   function mostrarOriginal() {
      this.src = "derecha.gif";
   }
   // Función que cambia la imagen original
   function cambiarOriginal() {
      this.src = "izquierda.gif";
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Mueve el puntero sobre la imagen</H3>
<IMG ID="imagen1" NAME="flecha" SRC="derecha.gif" />
<SCRIPT TYPE="text/javascript">
   // Asignamos los manejadores
   document.flecha.onmouseover = cambiarOriginal;
   document.flecha.onmouseout = mostrarOriginal;
</SCRIPT>
</BODY>
</HTML>
```

Como ve, nos quedan unas funciones bastante simples si utilizamos la sentencia `this` ya que, al provocarse los eventos, las funciones la enlazan con el objeto de imagen. También podríamos haber escrito la ruta completa (`document.flecha.src`) pero no estaríamos aprovechando el potencial de asignar los manejadores mediante código JavaScript.

Operaciones sobre formularios

El primer manejador que vamos a ver, en los ejemplos relacionados con los formularios, será `onclick` a través de un botón. Este manejador ejecutará una función que validará los campos existentes en el formulario.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que valida el formulario
   function validarFormulario() {
      // Operaciones de validación
```

```
    with(document.formLogin) {
       if (usuario.value == "") {
          usuario.focus();
       } else if (clave.value == "") {
          clave.focus();
       } else {
          alert("¡Todo correcto! Puedes enviar el
          formulario.");
       }
    }
  }
</SCRIPT>
</HEAD>
<BODY>
<H3>Formulario de acceso</H3>
<FORM ID="formulario1" NAME="formLogin" ACTION="login.php"
METHOD="post">
Usuario <INPUT TYPE="text" ID="entrada1" NAME="usuario"
MAXLENGTH="8" /><BR />
Contraseña <INPUT TYPE="password" ID="entrada2"
NAME="clave" /><BR />
<INPUT TYPE="button" ID="boton1" NAME="botonValidar"
VALUE="Validar" />
<INPUT TYPE="submit" ID="boton2" NAME="botonEnviar" />
<SCRIPT TYPE="text/javascript">
   // Asignamos el manejador
   document.formLogin.botonValidar.onclick =
   validarFormulario;
</SCRIPT>
</BODY>
</HTML>
```

Nada complicado, como puede comprobar. Únicamente cambia la forma de asignar el manejador respecto al método por HTML. En la versión extendida del ejemplo, donde unificamos las acciones de los dos botones en uno sólo (envío y validación), sería exactamente igual pero utilizando el manejador del botón correspondiente.

```
<SCRIPT TYPE="text/javascript">
   // Asignamos el manejador
   document.formLogin.botonEnviar.onclick =
   validarYEnviarFormulario;
</SCRIPT>
```

Ahora es el turno del manejador `onchange` de las listas desplegables. El uso de la sentencia `this` es casi el pan nuestro de cada día, pero recuerde que aquí siempre se debe usar dentro de las funciones y no como un parámetro de la misma, de modo que nos quedaría así:


```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que comprueba la opción seleccionada
   function comprobarMezcla() {
      if (this.selectedIndex == 2) {
         alert("¡Muy bien!");
      } else {
         alert("Lo siento, no es este color… Inténtalo
         de nuevo.");
      }
   }
</SCRIPT>
</HEAD>
<BODY>
<H3>Adivina la mezcla de colores</H3>
<FORM ID="formulario1" NAME="formColores">
Rojo +
<SELECT ID="seleccion1" NAME="color">
   <OPTION VALUE="Verde">Verde</OPTION>
   <OPTION VALUE="Azul">Azul</OPTION>
   <OPTION VALUE="Amarillo">Amarillo</OPTION>
</SELECT>
= Naranja
</FORM>
<SCRIPT TYPE="text/javascript">
   // Asignamos el manejador
   document.formColores.color.onchange = comprobarMezcla;
</SCRIPT>
</BODY>
</HTML>
```

Ocurre lo mismo cuando queremos acceder al objeto que representa la opción seleccionada para obtener su índice y valor, siempre dentro de la función.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
   // Función que comprueba la opción seleccionada
   function comprobarMezcla() {
      // Obtenemos la opción seleccionada
      var opcionSeleccionada = this.options[this.
      selectedIndex];
      if (opcionSeleccionada.index == 2) {
         alert("¡Muy bien!");
      } else {
         alert("Lo siento, no es " + opcionSeleccionada.
         value + "… Inténtalo de nuevo.");
      }
   }
```

```
</SCRIPT>
</HEAD>
<BODY>
<H3>Adivina la mezcla de colores</H3>
<FORM ID="formulario1" NAME="formColores">
Rojo +
<SELECT ID="seleccion1" NAME="color">
   <OPTION VALUE="Verde">Verde</OPTION>
   <OPTION VALUE="Azul">Azul</OPTION>
   <OPTION VALUE="Amarillo">Amarillo</OPTION>
</SELECT>
= Naranja
</FORM>
<SCRIPT TYPE="text/javascript">
   // Asignamos el manejador
   document.formColores.color.onchange = comprobarMezcla;
</SCRIPT>
</BODY>
</HTML>
```

Como ya no podemos usar parámetros, lo que hemos hecho ha sido obtener la opción seleccionada directamente dentro de la función gracias a que tenemos acceso completo al objeto que representa la lista desplegable, mediante la sentencia `this`.

10.4. El objeto event

Aún después de todo lo explicado, JavaScript nos tiene preparado un as más bajo la manga.

Cuando se produce un evento, se crea automáticamente un objeto (`event`) que contiene un conjunto de información acerca de ese evento. Este objeto podrá ser utilizado mientras se produce ese evento, de modo que desaparecerá cuando finalice, también de manera automática.

Siendo estrictos, este objeto tiene un mayor protagonismo en códigos relacionados con DHTML pero quiero presentárselo ya que creo que es importante conocerlo, aunque sea un poco, para tener una buena base de los eventos de JavaScript.

Ahora se estará preguntando cómo y dónde podemos usar este nuevo objeto. Bien, vamos a ello. El objeto `event` es accesible dentro de las funciones asignadas en los manejadores. ¿Cómo lo recuperamos? Veamos mejor un sencillo ejemplo con el evento `click`.


```
<SCRIPT TYPE="text/javascript">
   // Función que muestra un mensaje
   function mostrarEvento(evento) {
      alert(evento);
   }
   // Asignaciones al manejador (forma 1)
   document.onclick = mostrarEvento;
   // Asignaciones al manejador (forma 2)
   document.onclick = function mostrarEvento(evento) {
   alert(evento); }
</SCRIPT>
```

Si observamos el código, vemos que hemos puesto un parámetro a la función usada en el manejador. ¿Cómo es posible si habíamos dicho que esto no se podía hacer? Esto es así porque en realidad no estamos pasando un parámetro sino dando un nombre al objeto `event` que JavaScript está creando automáticamente. También se puede omitir ese nombre y recuperar el objeto con `arguments[0]` dentro de la función.

La asignación se puede hacer de las dos maneras mostradas, ya que consiguen el mismo resultado.

> **Advertencia:** *Estas son la formas estándar de recuperar el objeto* `event`. *En Internet Explorer también se puede acceder a él desde* `window.event`.

Si utilizásemos el código anterior, conseguiríamos un mensaje como el de la figura 10.5 al hacer clic sobre cualquier parte de la ventana de nuestro navegador. No nos dice mucho de momento, pero ahora veremos cómo sonsacarle la información que tiene guardada.

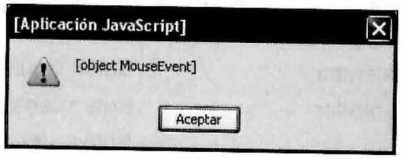


Figura 10.5. Recuperación del objeto event.

10.4.1. Propiedades

En la tabla 10.3 se mostrará un listado de las principales propiedades estándar que contiene el objeto `event`. Existen algunas más pero, al ser propias de algunos navegadores o muy específicas de DHTML, no las vamos a tratar en este libro.

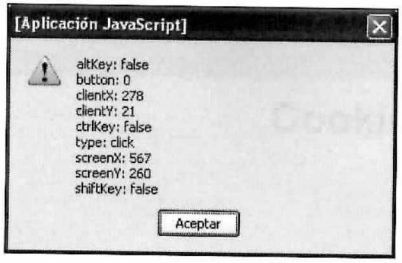
Tabla 10.3. Propiedades del objeto event

Propiedad	*Tipo*	*Descripción*
altKey	Booleano	Devuelve true cuando se ha presionado la tecla **Alt**.
button	Entero	Indica el botón del ratón que ha sido pulsado. Ver tabla 10.4.
clientX	Entero	Indica la posición horizontal del cursor donde se ha hecho clic, dentro del área disponible en el documento.
clienteY	Entero	Indica la posición vertical del cursor donde se ha hecho clic, dentro del área disponible en el documento.
ctrlKey	Booleano	Devuelve true cuando se ha presionado la tecla **Ctrl**.
type	String	Indica el tipo de evento que se ha producido (click, mouseover, etc.).
screenX	Entero	Indica la posición horizontal del cursor donde se ha hecho clic, dentro de la resolución de pantalla del usuario.
screenY	Entero	Indica la posición vertical del cursor donde se ha hecho clic, dentro de la resolución de pantalla del usuario.
shiftKey	Booleano	Devuelve true cuando se ha presionado la tecla **Shift**.

Tabla 10.4. Valores de la propiedad button

Valor	*Navegador*	*Descripción*
0	Estándar	Botón izquierdo del ratón
1	Estándar	Botón central del ratón
2	Estándar	Botón derecho del ratón
1	Internet Explorer	Botón izquierdo del ratón
2	Internet Explorer	Botón derecho del ratón
4	Internet Explorer	Botón central del ratón

La figura 10.6 muestra un ejemplo de la información contenida en el objeto `event` cuando se hace clic en alguna parte



del documento, según el código que vimos en el apartado anterior.

Figura 10.6. Propiedades de un evento.

Para terminar con este objeto, tenga siempre presente que en una página pueden lanzarse varios eventos al realizar una acción. Un caso muy claro es el que hemos estado viendo con `click`. Cuando pulsamos sobre una parte del documento, no sólo estamos provocando dicho evento sino que, en este caso, además se producen otros dos: `mousedown` y `mouseup`. Aunque no tengamos un manejador asociado a ellos, los eventos siempre van a producirse. Ahora la duda que nos queda es ¿en qué orden aparecen? Para esta acción los eventos siguen esta secuencia:

1. `mousedown`: al hacer la pulsación sobre el botón del ratón.
2. `mouseup`: al soltar el botón.
3. `click`: al soltar el botón también, pero después de `mouseup`.

Si nos construimos un sencillo código podremos ver esta secuencia.

```
<FORM NAME="formEventos">
   <TEXTAREA NAME="eventos" ROWS="9"></TEXTAREA>
</FORM>
<SCRIPT TYPE="text/javascript">
   // Función que añade un evento en el textarea
   function pintarEvento(evento) {
      document.formEventos.eventos.value += evento.type
      + "\n";
   }
   // Asignamos los manejadores (no importa el orden)
   document.onclick = pintarEvento;
   document.onmousedown = pintarEvento;
   document.onmouseup = pintarEvento;
</SCRIPT>
```

11 Cookies

Una *cookie* no es más que una porción de información que se almacena temporalmente en el ordenador del usuario con la finalidad de poder recuperarla en cada visita una misma página y así evitar tener que introducirla una y otra vez. Esto hace que la navegación del usuario sea más cómoda y pueda ir directamente a la sección que le interese de nuestra página.

La información que se suele guardar en las *cookies* es introducida por el usuario cuando entra por primera vez a la página, para poder mostrarle unos datos más personalizados o simplemente para ahorrarle tiempo en su próxima visita. Algunos ejemplos de la utilidad de las *cookies* son:

- Nombre de usuario. Crea la sensación de que la página "recuerda" nuestro nombre de usuario cuando volvemos a visitarla.
- Fecha última visita. Esto puede tener utilidad sobre todo en los foros para saber el tiempo que llevas sin entrar, por ejemplo.
- Colores de la página. Si la página nos permite personalizar el aspecto que puede tener, es de gran ayuda almacenar esas preferencias para que el usuario no tenga que volver a configurarlo una y otra vez.
- Tienda más cercana. A las páginas que muestren ofertas de productos les puede ser útil almacenar este dato para ofrecer al usuario una información más personalizada.

Hace tiempo, las *cookies* se usaban para almacenar los productos que se iban seleccionando en una Web de ventas, al estilo de un carrito de la compra, pero con el aumento del uso

de lenguajes de lado servidor, como PHP o ASP, esto ha caído en desuso y ahora se utiliza la sesión que mantiene el usuario con el servidor como medio para guardar esa cesta.

> **Advertencia**: *A la hora de crear nuestras propias cookies debemos evitar guardar en ellas información confidencial del usuario, como pueden ser contraseñas, número de DNI, número de tarjetas de crédito, etc. Hacerlo supondría un alto riesgo puesto que esos datos podrían ser recuperados por una página creada para tal propósito.*

Como hemos dicho, las *cookies* se almacenan en el ordenador de usuario, pero ¿dónde exactamente? Al igual que muchas otras cosas, cada navegador tiene definido un sitio y una manera de guardarlas:

- Internet Explorer: Para sistemas Windows, la ruta típica es `"C:\Documents and Settings\<su_usuario>\Cookies"`, donde cada cookie se almacena en un fichero distinto.
- FireFox: También para Windows, en la carpeta `"C:\Documents and Settings\<su_usuario>\Datos de programa\Mozilla\Firefox\Profiles\<su_perfil>"`. Ahí podremos encontrar los archivos `"cookies.txt"` y `"hostperm.1"` que almacenan todas las *cookies* juntas.

JavaScript guarda todas las *cookies* dentro de la propiedad `cookie` del objeto `document`.

Hay que señalar que las *cookies* no permanecen en nuestro ordenador hasta la eternidad sino que tienen una fecha de expiración o caducidad, que puede ser de unos segundos, horas, días o simplemente hasta cuando cerremos la ventana del navegador.

Opcionalmente, es posible activar o desactivar el almacenamiento de *cookies* desde las opciones del navegador, aunque por defecto suele venir activada.

- En Internet Explorer debemos pasar por Herramientas>Opciones de Internet>Privacidad>Avanzada. Una vez ahí tenemos la posibilidad de cambiar la administración de las *cookies*. Ver figura 11.1.
- En FireFox es más sencillo. Si vamos a Herramientas>Opciones>Privacidad veremos una sección dedicada a las *cookies*. Ver figura 11.2.

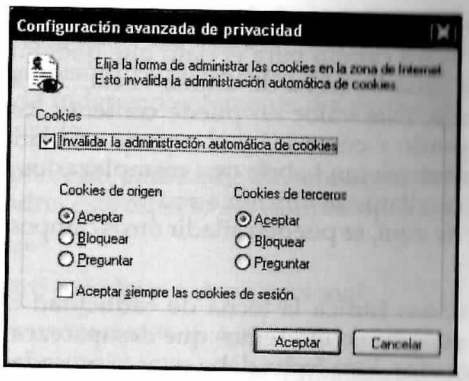


Figura 11.1. Administración de cookies en Internet Explorer.

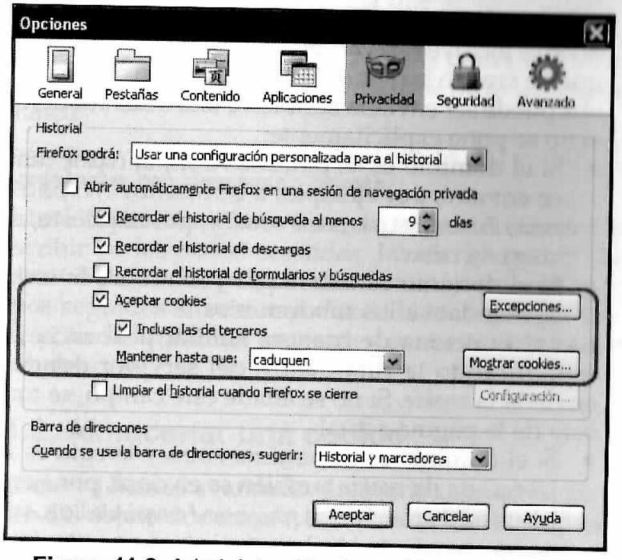


Figura 11.2. Administración de cookies en FireFox.

11.1. Trabajar con cookies

11.1.1. Estructura de una cookie

La manera de almacenar las *cookies* en JavaScript es en forma de pares campo=valor, separando cada par por un punto y coma (`;`).

La información mínima y obligatoria que debe contener cada *cookie* es un campo para el dato que queremos guardar (nombre de usuario, fecha de última visita...) y su valor correspondiente. Este valor no puede contener los caracteres punto (`.`), punto y coma (`;`) y el espacio en blanco, pero en caso de que estuvieran habría que reemplazarlos por valores UNICODE mediante la función `escape`.

A partir de aquí, se pueden añadir otros campos opcionales como son:

- `expires`: Indica la fecha de caducidad de la *cookie*, sólo cuando no queremos que desaparezca al cerrar el navegador. Esta fecha debe estar expresada en formato GMT, por lo que aquí nos será de mucha ayuda el método `toGMTString` del objeto `Date`.
- `domain`: Indica a qué servidor debe ser enviada la *cookie*. Si no se incluye este campo, entonces se toma el dominio que ha creado la *cookie*. Al especificar un dominio, la *cookie* puede ser enviada también a todos sus subdominios si no se pone explícitamente:
  - Si el dominio es `mipagina.com`, la *cookie* también se enviará, por ejemplo, a `buscador.mipagina.com`, `foro.mipagina.com` y, por supuesto, a `mipagina.com`.
  - Si el dominio es `www.mipagina.com`, la *cookie* no se mandará a los subdominios.
- `path`: Funciona de manera similar a `domain`, pero especificando la ruta dentro del servidor donde será enviada la *cookie*. Si no se añade este campo, se toma la ruta de la página actual.
  - Si el dominio es `mipagina.com` y la ruta es `/capitulos`, la *cookie* también se enviará, por ejemplo, a `buscador.mipagina.com/capitulos`, `mipagina.com/capitulos` o `mipagina.com/capitulos/JavaScript/`.
  - Si el dominio es `www.mipagina.com` y la ruta `/capitulos`, la *cookie* se mandará a `www.mipagina.com/capitulos` o `www.mipagina.com/capitulos/JavaScript`.
  - Si la ruta es `/` entonces la *cookie* se envía por todo el dominio.
- `secure`: Si este campo está incluido en la *cookie*, entonces ésta sólo se envía al servidor si la conexión se ha establecido a través de un protocolo seguro, como puede ser HTTPS (*HTTP Secure*). Este el único campo que no lleva un valor asociado ya que está representado por si mismo. Si no se especifica, entonces la *cookie* se manda independientemente del protocolo que se esté utilizando.

Dicho todo esto, una *cookie* que almacenase el nombre de usuario tendría este aspecto sin campos opcionales:

```
usuario=Pepe
```

Y este otro con algún campo opcional:

```
usuario=Pepe;expires=Tue, 3 Mar 2009 18:24:41 GMT;secure
```

Dado que todos los campos, obligatorios u opcionales, van separados por punto y coma (`;`), se hace difícil delimitar dónde acaba cada *cookie*. Una regla que se suele utilizar es la de insertar un espacio en blanco después del punto y coma de los campos opcionales. De esta forma podremos saber que una *cookie* comienza por aquel campo que no tenga ese espacio en blanco.

```
usuario=Pepe; expires=Tue, 3 Mar 2009 18:24:41 GMT; secure
;compra=leche,pan,manzanas; secure
```

Esta delimitación sólo va a ser visual para nosotros a la hora de escribir varias *cookies* seguidas. JavaScript puede manejar cada *cookie* de forma independiente aunque vayan todos los campos seguidos sin ese espacio en blanco, ya que es capaz de detectar las palabras especiales pertenecientes a campos opcionales.

11.1.2. Almacenar una cookie

Como ya hemos avanzado, JavaScript almacena las *cookies* dentro del objeto `document`, concretamente en su propiedad `cookie`. Por tanto, lo único que tendremos que hacer es construir una cadena que contenga un estructura de *cookie*, es decir, pares campo=valor y separados por punto y coma (`;`). Para facilitarnos este trabajo, nos crearemos una función que luego podremos reutilizar donde queramos. Esta función recibirá como parámetros cada uno de los posibles campos que pueden formar la *cookie* y los irá concatenando si es necesario.

```
<SCRIPT TYPE="text/javascript">
   function almacenarCookie(nombre, valor, caducidad,
   dominio, ruta, segura) {
      var cookie;
```

```
    // Añadimos el nombre de la cookie y su valor
    cookie = nombre + "=" + escape(valor);
    // Comprobamos si hemos recibido caducidad
    if (arguments[2] != undefined) {
       cookie += "; expires=" + caducidad.toGMTString();
    }
    // Comprobamos si hemos recibido dominio
    if (arguments[3] != undefined) {
       cookie += "; domain=" + dominio;
    }
    // Comprobamos si hemos recibido ruta
    if (arguments[4] != undefined) {
       cookie += "; path=" + ruta;
    }
    // Comprobamos si hemos recibido segura y si hay
    // que añadirla
    if (arguments[5] != undefined && segura) {
       cookie += "; secure";
    }
    // Añadimos la cookie
    document.cookie = cookie;
  }
</SCRIPT>
```

Viendo este código podría pensar que al guardar una *cookie* se sobrescribe el contenido completo de la propiedad `cookie` ya que no se añade el valor (operadores + o +=), sino que directamente se realiza una asignación (operador =). El ejemplo es correcto y no debe preocuparse puesto que JavaScript es el que se encarga de concatenar cada una de las *cookies* que añadamos a la propiedad de `document`.

> **Nota**: *Si el nombre de una cookie ya existe, no se crea una nueva sino que se reemplaza su valor con el nuevo.*

11.1.3. Recuperar una cookie

Al contrario de lo que pueda pensar, cuando obtenemos la información de una *cookie* que está almacenada en el objeto `document`, únicamente podremos extraer su valor, por lo que todo lo referente a los campos opcionales quedará oculto. Esto lo puede comprobar rápidamente si intenta mostrar las *cookies* guardadas:

```
<SCRIPT TYPE="text/javascript">
   // Mostramos las cookies
   alert(document.cookie);
</SCRIPT>
```

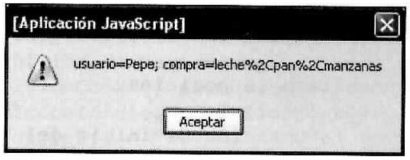



Figura 11.3. Colección de cookies almacenadas en JavaScript.

¿Ha visto? Sólo puede ver los pares nombre=valor de cada *cookie* que haya almacenada (dos en nuestro caso). Esto no nos supone ningún problema, puesto que los otros campos los gestionará el navegador para detectar cuándo caduca, y por tanto borrar la *cookie*, o saber a qué dominio o ruta pertenecen. Además puede ver sustituido el carácter coma (,) por su equivalente UNICODE (`%2C`) ya que ese carácter no está permitido como valor de la *cookie*.

Recapitulando, para leer el valor de una *cookie* en concreto tendremos que encontrar el par nombre=valor que nos interese dentro de la propiedad `document.cookie`, la cual veremos como una única cadena con todas las *cookies* concatenadas, así que necesitaremos usar las funciones de búsqueda de las cadenas. Lo mejor en este caso, al igual que para almacenar una *cookie*, será crearnos una función que nos permita obtener cualquier *cookie* indicando únicamente su nombre.

Lo primero que debemos comprobar es si tenemos alguna cookie almacenada para evitar lanzar una búsqueda que sabemos que no va a devolver nada.

```
// Comprobar si hay cookies
if (document.cookies.length > 0) {
   // Buscar y devolver el valor de la cookie
} else {
   // Devolver un valor vacío
   return "";
}
```

En caso de tener alguna *cookie* almacenada, debemos localizar en qué posición de la concatenación de *cookies* se encuentra la que nos interesa. Esto lo haremos buscando por el nombre de la misma.

Además comprobaremos si hemos encontrado esa posición o no, para detener de nuevo la búsqueda. Si hemos encontrado la *cookie* entonces calculamos la posición dónde comienza el valor de la misma, que será a continuación del carácter = del par nombre=valor.

```
// Buscar la posición de la cookie
posicionCookie = document.cookie.indexOf(nombreCookie +
"=");
// Si hemos encontrado la posición
if (posicionCookie != -1) {
   // Calculamos la posición de inicio del valor
   posicionInicio = posicionCookie + nombreCookie.
   length + 1;
   // Obtener valor de la cookie
}
```

Una vez localizada la posición donde comienza nuestra *cookie* queda por calcular donde termina, para así poder obtener su valor. Como sabemos que cada *cookie* está separada del resto por un punto y coma (;), si logramos identificar cuál es la posición de la más próxima justo después de la posición de inicio, conseguiremos delimitar el valor de la *cookie* que estamos buscando. Debemos tener en cuenta la posibilidad de que no exista ese punto y coma si coincide que justamente estamos leyendo la última *cookie*. En este caso, podremos obtener el valor desde la posición de inicio hasta el final del *string* que forma `document.cookie`.

```
// Buscar la posición del punto y coma más próximo
posicionPuntoComa = document.cookie.indexOf(";",
posicionCookie);
// Si estamos en la última cookie (no hay punto y coma)
if (posicionPuntoComa == -1) {
   // Leemos hasta el final de la cadena
   posicionFin = document.cookie.length;
} else {
   // En otro caso, leemos hasta el punto y coma
   posicionFin = posicionPuntoComa;
}
```

Nos está quedando bien la función... Vamos con el último paso: ¡devolver el valor de la cookie que buscamos! Esto resultará bastante sencillo gracias a todo el trabajo previo que hemos hecho. Bastará con devolver la parte de la cadena que esté situada entre la posición de inicio y fin, aplicándole además la función `unescape` para restaurar los caracteres que sustituimos al almacenar la *cookie*.

```
// Devolver el valor de la cookie
return unescape(document.cookie.
substring(posicionInicio, posicionFin));
```

Ahora que ya hemos explicado todos los pasos, veamos todo junto dentro de una función con algunos pequeños cambios para hacerla más bonita (aún) aunque no es completamente necesario para lograr nuestro objetivo.


```
<SCRIPT TYPE="text/javascript">
   function recuperarCookie(nombreCookie) {
      // Definición de variables
      var posicionCookie, posicionPuntoComa;
      var posicionInicio, posicionFin;
      var valorCookie = "";
      // Comprobamos si hay cookies
      if (document.cookie.length > 0) {
         // Buscamos la posición de la cookie
         posicionCookie = document.cookie.
         indexOf(nombreCookie + "=");
         // Si hemos encontrado la posición
         if (posicionCookie != -1) {
            // Calculamos la posición de inicio
            // del valor
            posicionInicio = posicionCookie +
            nombreCookie.length + 1;
            // Buscamos la posición del punto y
            // coma más próximo
            posicionPuntoComa = document.cookie.
            indexOf(";", posicionCookie);
            // Si estamos en la última cookie (no hay
            // punto y coma)
            if (posicionPuntoComa == -1) {
               // Leemos hasta el final de la cadena
               posicionFin = document.cookie.length;
            } else {
               // En otro caso, leemos hasta el punto
               // y coma
               posicionFin = posicionPuntoComa;
            }
            // Obtenemos el valor de la cookie
            valorCookie = document.cookie.
            substring(posicionInicio, posicionFin);
         }
      }
      // Devolvemos el valor de la cookie
      return unescape(valorCookie);
   }
</SCRIPT>
```

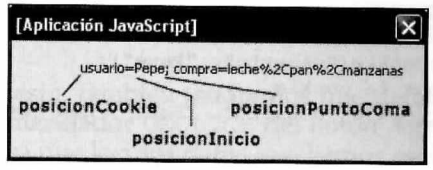


Figura 11.4. Representación visual de las posiciones de una cookie.

11.1.4. Modificar una cookie

Aunque ya avanzamos algo hace un par de apartados, modificar los campos de una *cookie* resulta bastante sencillo puesto que no necesitamos buscarla previamente dentro de la colección sino que será suficiente con volver a almacenarla con los nuevos valores. El único requisito, lógicamente, es que la *cookie* ya exista en la colección.

Esto quiere decir que con volver a usar nuestra función `almacenarCookie`, junto con los nuevos valores y el mismo nombre de *cookie*, habremos modificado todos los campos que hayamos indicado. Cuidado con esto porque en caso de usar un nombre de cookie que no esté en la colección almacenaremos una nueva.

```
<SCRIPT TYPE="text/javascript">
   // Creamos una cookie
   almacenarCookie("compra", "leche,pan,manzanas");
   // Modificamos la cookie
   almacenarCookie("compra",
"leche,pan,manzanas,queso");
</SCRIPT>
```

Así da gusto, ¿verdad? Las funciones que estamos creando nos van a ser de mucha utilidad para dar una claridad tremenda a nuestro código.

11.1.5. Borrar una cookie

Por suerte para nosotros, borrar una *cookie* resulta tan sencillo como modificarla. En concreto tendremos que cambiar su fecha de caducidad haciendo que no sea válida a partir del momento actual. La fecha que debemos fijar tiene que ser igual o anterior a la fecha y hora actuales. Lo más sencillo es usar la hora actual, ya que la *cookie* se invalidará en el siguiente milisegundo.

```
<SCRIPT TYPE="text/javascript">
   // Creamos una cookie
   almacenarCookie("usuario", "Pepe");
   // Borramos la cookie
   almacenarCookie("usuario", "Pepe", new Date());
</SCRIPT>
```

Tras hacer esto, el navegador detectará que la *cookie* ha caducado, la borrará de la colección y ya no será recuperable.

11.2. Ejemplos prácticos

Para comprender todas estas funciones y acciones que se han explicado, vamos a desarrollar unos pequeños ejemplos utilizando formularios y código JavaScript para que sean más cómodos de probar.

11.2.1. Almacenar y recuperar una cookie

Estas son las acciones más básicas y comunes que se pueden aplicar sobre las *cookies*. Para comprobarlo por nosotros mismos, crearemos un sencillo formulario que nos permita añadir *cookies* que constará de varias cajas de texto para introducir los valores y un botón para añadirla a la colección de *cookies* del navegador.

```
<HTML>
<BODY>
<H3>Formulario para añadir cookies</H3>
<FORM ID="formulario1" NAME="formCookie">
Nombre de la cookie: <INPUT TYPE="text" ID="entrada1"
NAME="nombre" /> <B>(obligatorio)</B><BR />
Valor: <INPUT TYPE="text" ID="entrada2" NAME="valor" />
<B>(obligatorio)</B><BR />
<INPUT TYPE="button" ID="boton1" NAME="botonAgregar"
VALUE="Agregar" ONCLICK="agregarCookie()" />
<INPUT TYPE="button" ID="boton2" NAME="botonLeer"
VALUE="Leer" ONCLICK="leerCookie()" />
</FORM>
</BODY>
</HTML>
```

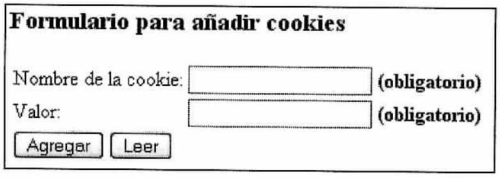


Figura 11.5. Formulario para administrar cookies.

Por supuesto, también tendremos que definir la función asociada al manejador `ONCLICK` del botón **Agregar**, que no hará otra cosa que leer los datos que hemos escrito en el formulario y llamar a la función `almacenarCookie` que hemos creado en el apartado anterior.

```
<SCRIPT TYPE="text/javascript">
   function agregarCookie() {
      // Obtenemos los campos
      with (document.formCookie) {
         // Comprobamos si hemos indicado el nombre
         // y valor
         if (nombre.value == "" || valor.value == "") {
            alert("Debe escribir el nombre y valor
            de la cookie.");
            nombre.focus();
            return;
         } else {
            // Llamamos a la función que crea la cookie
            almacenarCookie(nombre.value, valor.value);
            alert("Cookie " + nombre.value +
            " almacenada.");
         }
      }
   }
</SCRIPT>
```

Como ve, esta función no tiene mucha complicación. Solamente hemos ido comprobando cada uno de los campos del formulario para saber si tenían algo escrito y mostrar un mensaje al usuario en caso contrario, dejando el foco en la caja de texto correspondiente al nombre de la *cookie*.

En cuanto a recuperar los valores de las *cookies* nos tendremos que construir otra función (`leerCookie`), para asociarla al manejador `ONCLICK` del botón **Leer**, que se encargará de mostrarnos el valor de la *cookie* cuyo nombre coincida con el que esté escrito en el formulario. Aquí usaremos nuestra magnífica función `recuperarCookie`.

```
<SCRIPT TYPE="text/javascript">
   function leerCookie() {
      var nombreCookie;
      var valorCookie = "";
      // Obtenemos el nombre
      with (document.formCookie) {
         // Comprobamos si hemos indicado el nombre
         if (nombre.value == "") {
            alert("Debe escribir el nombre
            de la cookie.");
            nombre.focus();
            return;
         } else {
            // Guardamos el nombre de la cookie
            nombreCookie = nombre.value;
         }
         // Llamamos a la función que recupera la cookie
```



```
         valorCookie = recuperarCookie(nombreCookie);
      }
      // Mostramos el valor de la cookie
      if (valorCookie != "") {
         alert("El valor de la cookie " + nombreCookie +
         " es:\n" + valorCookie);
      } else {
         alert("No se ha encontrado la cookie " +
         nombreCookie);
      }
   }
</SCRIPT>
```

Bueno, pues después de tanto código llega el momento de probarlo (cruce los dedos). Le invito a recrear los siguientes casos en su ordenador:

1. Recuperar una *cookie* sin haber almacenado ninguna. Debe aparecer un mensaje indicando que no se ha encontrado una *cookie* con ese nombre.
2. Almacenar una *cookie* con un nombre de usuario. La *cookie* se almacenará en la colección.
3. Recuperar la *cookie* con el nombre de usuario. Debe aparecer un mensaje con el valor de la *cookie*.
4. Modificar el valor de la *cookie* con el nombre de usuario. La *cookie* se almacenará en la colección.
5. Recuperar la *cookie* con el nombre de usuario. Debe aparecer un mensaje con el nuevo valor de la *cookie*.

11.2.2. Expiración de cookies

Tomando como base el caso anterior, vamos a añadir un campo más al formulario para poder indicar cuándo queremos que caduque o expire nuestra *cookie*, expresado en segundos.

```
<HTML>
<BODY>
<H3>Formulario para expiración de cookies</H3>
<FORM ID="formulario1" NAME="formCookie">
Nombre de la cookie: <INPUT TYPE="text" ID="entrada1"
NAME="nombre" /> <B>(obligatorio)</B><BR />
Valor: <INPUT TYPE="text" ID="entrada2" NAME="valor" />
<B>(obligatorio)</B><BR />
Caducidad: <INPUT TYPE="text" ID="entrada3"
NAME="caducidad" /> (segundos)<BR />
<INPUT TYPE="button" ID="boton1" NAME="botonAgregar"
VALUE="Agregar" ONCLICK="agregarCookie()" />
```

```
<INPUT TYPE="button" ID="boton2" NAME="botonLeer"
VALUE="Leer" ONCLICK="leerCookie()" />
</FORM>
</BODY>
</HTML>
```

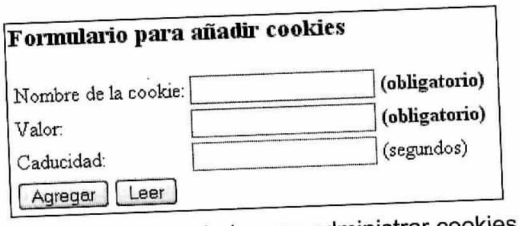


Figura 11.6. Formulario para administrar cookies.

Del mismo modo, debemos añadir unas líneas a la función del manejador `ONCLICK` del botón **Agregar** para poder almacenar la *cookie* con la caducidad que hayamos indicado en el formulario.

```
<SCRIPT TYPE="text/javascript">
   function agregarCookie() {
      var expiracion;
      var hoy = new Date();
      // Obtenemos los campos
      with (document.formCookie) {
         // Comprobamos si hemos indicado el nombre
         // y valor
         if (nombre.value == "" || valor.value == "") {
            alert("Debe escribir el nombre y valor
            de la cookie.");
            nombre.focus();
            return;
         }
         // Comprobamos si hemos indicado caducidad
         if (caducidad.value != "") {
            expiracion = new Date();
            expiracion.setTime(hoy.getTime() +
            (caducidad.value * 1000));
         }
         // Llamamos a la función que crea la cookie
         almacenarCookie(nombre.value, valor.value,
         expiracion);
         alert("Cookie " + nombre.value +
         " almacenada.");
      }
   }
</SCRIPT>
```


En el cálculo de la fecha de caducidad hay un pequeño detalle que explicar ya que es posible que le resulte extraña esa multiplicación por mil. En realidad lo que se ha hecho es guardar la fecha en forma de milisegundos. Para ello, hemos transformado la fecha y hora actuales en esa unidad mediante el método `getTime` y le hemos sumado el número de segundos indicados en el formulario, también expresado en milisegundos (de ahí la multiplicación).

Con este ejemplo podremos intentar crear dos *cookies* con distintas fechas de caducidad:

1. Para la primera (de nombre `cookie_sesion`) no indicaremos una cantidad de segundos como caducidad, por lo que expirará cuando cerremos la ventana del navegador. Esto significa que seremos capaces de recuperar su valor siempre y cuando no cerremos el navegador. Una vez que lo hagamos, la cookie habrá expirado y no permanecerá en la colección.
2. A la segunda (de nombre `cookie_minuto`) le pondremos 60 segundos como tiempo de expiración, así que podremos obtener su valor durante el siguiente minuto aunque cerremos el navegador. Pasado ese tiempo la cookie expirará.

Por último, sólo destacar que aunque únicamente calculemos la fecha de caducidad cuando se ha introducido un valor en el formulario, siempre se pasa la variable `expiracion` como parámetro a la función `almacenarCookie` ya que, si no hemos realizado el cálculo, su valor será `undefined` y provoca que no se añada ese parámetro opcional a la *cookie*.

11.2.3. Modificar una cookie

Para probar esta funcionalidad podemos utilizar el formulario y las funciones JavaScript asociadas a él de los ejemplos anteriores, puesto que la modificación de una *cookie* equivale a volver a almacenarla y el procedimiento de recuperación de su valor no varía. Realizaremos las siguientes operaciones para ver cómo funciona la modificación de una *cookie*:

1. Creamos una *cookie* y a continuación recuperamos su valor. Ahora modificaremos su valor ("creamos" la misma *cookie* con el nuevo valor) y volvemos a recuperarla para comprobar que el cambio se ha hecho efectivo.

2. Creamos una *cookie* sin indicar fecha de caducidad (expirará al cerrar el navegador). Seguidamente modificamos su fecha de caducidad a 60 segundos, por ejemplo. Entonces si cerramos el navegador e intentamos recuperar la *cookie* veremos su valor. Pasado ese tiempo, la *cookie* no debería seguir estando accesible.

11.2.4. Borrar una cookie

Por último, vamos a comprobar lo fácil que resulta eliminar una *cookie* de la colección usando el mismo formulario y código JavaScript que hasta ahora.

Para refrescarle la memoria, una *cookie* se borrará cuando haya expirado, es decir, cuando su fecha de caducidad sea anterior a la fecha actual. Como en nuestro formulario no indicamos una fecha concreta sino un plazo en segundos, bastará con poner un valor igual o menor a cero. El cero representa la hora actual (la *cookie* expira en el siguiente milisegundo) y los negativos indican una hora anterior a la actual (hace 3, 10 ó 60 segundos por ejemplo).

Por tanto, el ejemplo que nos ocupa consistirá en crear una nueva *cookie* y, tras comprobar que podemos recuperar su valor, modificaremos su caducidad con un valor -5 (o el primero que piense, pero siempre menor o igual que cero). Si a continuación intentamos recuperar el valor de esta *cookie* nos encontraremos con que ésta ha sido borrada.

Si queremos rizar el rizo, podríamos hacer una pequeña variación en la función `agregarCookie` para que nos indique cuándo borramos una *cookie*. Esto se puede hacer muy rápidamente si nos fijamos en el valor del campo caducidad del formulario: si no es vacío, y su valor es menor o igual a cero, entonces estamos borrando. En otro caso, la *cookie* se está almacenando.

```
// Llamamos a la función que crea la cookie
almacenarCookie(nombre.value, valor.value, expiracion);
if (caducidad.value != "" && caducidad.value <= 0) {
   alert("Cookie " + nombre.value + " borrada.");
} else {
   alert("Cookie " + nombre.value + " almacenada.");
}
```

Como ve, rápido, sencillo y para toda la familia.

12 Ejemplos prácticos

Este capítulo presenta una serie de ejercicios que intentan poner en práctica los diversos conocimientos que habrá adquirido con este libro. Aunque todos estos *scripts* vienen explicados y con su solución (que no la única, ya que siempre hay varias formas de llegar a lo mismo), le recomiendo que antes de mirarla intente pensar por su cuenta cómo lo resolvería, o al menos qué funciones u objetos harían falta para conseguir llegar al objetivo.

Sin más, le dejo que disfrute de nueve útiles ejemplos.

12.1. Mostrar un texto rotativo

Como primer ejemplo práctico vamos a simular el efecto de la etiqueta HTML `<MARQUEE>` (bastante obsoleta, por cierto) que consiste en mostrar un texto desplazándolo horizontalmente hacia la izquierda a modo de letrero informativo.

Los elementos que vamos a utilizar para crear este efecto son:

1. Una capa HTML que contendrá el mensaje.
2. Una función JavaScript que hará rotar el texto.

La capa HTML será muy sencilla y únicamente le fijaremos una anchura y un borde. El motivo de fijar estos dos atributos será para que el efecto se aprecie aunque el texto a rotar sea corto. Además, también centraremos el texto en la capa para que quede más bonito. No se preocupe si no conoce el lenguaje HTML a fondo, aquí va una ayuda de cómo debe estar definida la capa:

```
<DIV ID="capaTexto" STYLE="border: 1px solid #000000;
width: 250px; text-align: center">El texto que queramos
desplazar</DIV>
```

Seguidamente definiremos la función JavaScript que creará el efecto rotativo, cuyo mecanismo será el siguiente:

- Recoger el texto que hay dentro de la capa para saber cuál es. Esto es posible gracias a la propiedad `innerHTML`, que ya hemos explicado para objetos como `anchor` (ancla). Para acceder a la capa nos apoyaremos en el método `getElementById` del objeto `document`.
- Trasladar el primer carácter del texto al final, para simular que se ha desplazado una posición a la izquierda.
- Escribir el nuevo texto en la capa.

Con todo esto escribiremos la función que "dará vida" a nuestro texto.

```
function desplazarTexto() {
   // Obtenemos el texto de la capa
   var texto = document.getElementById("capaTexto").
   innerHTML;
   // Trasladamos el primer carácter al final
   texto = texto.substring(1, texto.length) + texto.
   substring(0, 1);
   // Escribimos el nuevo texto
   document.getElementById("capaTexto").innerHTML = texto;
}
```

Por último, necesitamos fijar un temporizador que ejecute repetidamente la función anterior para así crear el efecto deseado. Tenemos dos opciones:

- `setTimeout`: Nos ejecutará una única vez la expresión que le indiquemos.
- `setInterval`: Nos ejecutará continuamente la expresión indicada.

Lo vemos claro, ¿no? Nos conviene usar la función `setInterval` para arrancar de una sola vez el efecto del rotativo. En función del intervalo de tiempo que fijemos, el texto parecerá que se mueve más lento o más deprisa. Un valor medio pueden ser 150 milisegundos.

```
function arrancarRotativo() {
   // Fijamos el temporizador
   setInterval("desplazarTexto()", 150);
}
```


Para que el texto comience a desplazarse nada más cargar la página, asociaremos esta función al manejador `ONLOAD` del documento.

```
<BODY ONLOAD="arrancarRotativo()">
```

Pues bien, ¡ya tenemos preparado nuestro primer ejemplo!

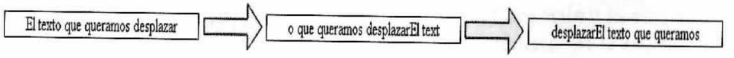


Figura 12.1. Efecto de texto rotativo.

Para hacer un poco más útil este texto rotativo, vamos a hacer que el efecto cese cuando pongamos el ratón encima de la capa, para poder leer el texto más cómodamente, y se reanude cuando lo quitemos. ¿Le suena difícil? ¡En absoluto! Ya verá que con cuatro pinceladas vamos a aumentar la funcionalidad de este rotativo.

Lo primero es escribir una función que detenga el temporizador que lanza contínuamente el desplazamiento del texto. Para ello nos hace falta conocer su identificador, así que tendremos que crear una variable global y guardar en ella el resultado de la ejecución de `setInterval`.

```
// Variable global
var temporizador;
function arrancarRotativo() {
   // Fijamos el temporizador
   temporizador = setInterval("desplazarTexto()", 150);
}
```

Ahora que ya tenemos lo necesario para detener el efecto, vamos a escribir la función de parada.

```
function detenerRotativo() {
   // Detenemos el temporizador
   clearInterval(temporizador);
}
```

¿Le parece que falta algo? Efectivamente, ya sólo queda incluir en la capa los manejadores correspondientes a los movimientos del ratón para que se lancen las funciones oportunas... ¡y tendremos todo listo para probar!

```
<DIV ID="capaTexto" STYLE="border: 1px solid #000000;
width: 250px; text-align: center" ONMOUSEOVER=
"detenerRotativo()" ONMOUSEOUT="arrancarRotativo()">El
texto que queramos desplazar</DIV>
```

12.2. Calcular la letra del DNI

Gracias a este ejemplo seremos capaces de calcular automáticamente la letra de un DNI. Si utilizamos esto dentro de un formulario, podremos saber si el usuario ha introducido bien su DNI o si le hemos cazado mintiendo.

Primeramente nos crearemos un pequeño formulario, para recoger el valor escrito por el usuario, que constará de una caja de texto y un botón que lanzará la operación de validación.

```
<H3>Formulario para validar un DNI</H3>
<FORM ID="formulario1" NAME="formDNI">
DNI: <INPUT TYPE="text" ID="entrada1" NAME="dni"
MAXLENGTH="9" />
<INPUT TYPE="button" ID="boton1" NAME="botonValidar"
VALUE="Validar" />
</FORM>
```

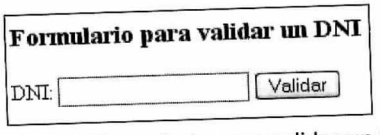


Figura 12.2. Formulario para validar un DNI.

Ahora definimos la función que calcula la letra que le corresponde a un número de DNI. Esta función es pública así que no crea que acabo de piratear nada:). Su funcionamiento es muy simple:

- Se dispone de una lista de 24 letras (de la A a la Z) almacenadas en un *string*.
- Para saber qué letra le corresponde a cada número, se calcula una posición a partir del número de DNI aplicando el módulo sobre 23 (`número_dni % 23`).
- Se extrae la letra que está en la posición calculada anteriormente.

Veamos esto ahora expresado en forma de código JavaScript:

```
function calcularLetra(numeroDNI) {
   // Declaración de variables
   var posicion, letra;
   var letrasDNI = "TRWAGMYFPDXBNJZSQVHLCKET";
   // Calculamos la posición
   posicion = numeroDNI % 23;
```



```
   // Extraemos la letra calculada
   letra = letrasDNI.substr(posicion, 1);
   // Devolvemos la letra
   return letra;
}
```

Puesto que este cálculo se realizará cuando presionemos el botón **Validar** del formulario, debemos crear otra función para asociarla a su manejador `ONCLICK`. Esta función recogerá el valor introducido en la caja de texto, que le eliminará la letra y llamará a la función `calcularLetra` con el valor resultante. Vayamos paso a paso.

Crearemos dos funciones: una para comprobar el formato del DNI introducido mediante un patrón y a la vez desechar la letra que contenga, y otra función, para asociarla al manejador, que nos dirá si la letra es correcta.

La primera función comprobará antes de nada que el DNI tiene un formato válido. Para realizar este proceso usaremos uno de los patrones que mostramos en el apartado 2.4 del capítulo 2: `/\d{8}[a-zA-Z]/`. Si se supera esta pequeña prueba, entonces separaremos la letra del DNI (que está justo al final del todo) y devolveremos el resto del número para su posterior validación. En otro caso, mostraremos un mensaje al usuario para advertirle que el DNI introducido no es válido.

```
function comprobarFormato(numeroDNI) {
   // Declaración de variables
   var patronDNI = new RegExp("\\d{8}[a-zA-Z]");
   // Comprobamos el formato
   if (patronDNI.test(numeroDNI)) {
      // Devolvemos los 8 números del DNI
      return numeroDNI.substr(0, 8);
   } else {
      // Mostramos un mensaje al usuario
      alert("El DNI " + numeroDNI + " no tiene un
      formato válido");
      return "";
   }
}
```

Ahora vamos con la función que vamos a asociar al manejador del botón **Validar**. Lo que tendrá que hacer es comprobar que el DNI introducido es válido, mediante la función `comprobarFormato`, y después comparar la letra introducida con la calculada en la función `calcularLetra`. Si ambas coinciden entonces se informa al usuario de que su DNI es correcto, y de lo contrario si no coinciden.

```
function validarDNI() {
   // Declaración de variables
   var DNIUsuario, numeroDNI, letraUsuario, letraCalculada;
   // Recogemos el valor del formulario
   DNIUsuario = document.formDNI.dni.value;
   // Comprobamos el formato y guardamos el número
   // sin la letra
   numeroDNI = comprobarFormato(DNIUsuario);
   // Si el formato es correcto
   if (numeroDNI != "") {
      // Guardamos la letra introducida por el usuario
      letraUsuario = DNIUsuario.substr(-1);
      // Calculamos la letra por nuestra cuenta
      letraCalculada = calcularLetra(numeroDNI);
      // Comparamos ambas letras mostrando un mensaje
      if (letraUsuario.toUpperCase() == letraCalculada) {
         alert("El DNI " + DNIUsuario + " es correcto");
      } else {
         alert("La letra del DNI " + DNIUsuario +
         " no es correcta");
      }
   }
}
```

Para terminar este ejemplo tan sólo queda asociar la función `validarDNI` al manejador `ONCLICK` del botón del formulario y ya habremos terminado todo el trabajo.

```
<INPUT TYPE="button" ID="boton1" NAME="botonValidar"
VALUE="Validar" ONCLICK="validarDNI()"/>
```

12.3. Un reloj despertador

Creo que todos conocemos estos simpáticos artilugios que se encargan de recordarnos todas las mañanas (de mejor o peor manera) que tenemos que resoplar y hacer un esfuerzo por salir de esa mullida "prisión" en forma de cama (qué poético ha quedado, ¿no?). En resumidas cuentas, el reloj despertador que nos vamos a construir en JavaScript se encargará de hacer saltar una alarma (silenciosa, por suerte para nosotros) cuando la hora actual sea la misma que la que hayamos fijado para tal fin.

Para hacer las cosas paso a paso, vamos a incluir primero un par de formularios en la página: uno para mostrar la hora actual y así saber cuándo va a llegar el momento que todos tememos, y otro para fijar la hora de la alarma y activarla o desactivarla.


```
<H3>Hora actual</H3>
<FORM ID="formulario1" NAME="formHora">
<INPUT TYPE="text" ID="entrada1" NAME="horaActual"
SIZE="2" READONLY /> :
<INPUT TYPE="text" ID="entrada2" NAME="minActual"
SIZE="2" READONLY /> :
<INPUT TYPE="text" ID="entrada3" NAME="segActual"
SIZE="2" READONLY />
</FORM><BR />
<H3>Alarma</H3>
<FORM ID="formulario2" NAME="formAlarma">
<INPUT TYPE="text" ID="entrada4" NAME="horaAlarma"
SIZE="2" /> :
<INPUT TYPE="text" ID="entrada5" NAME="minAlarma"
SIZE="2" />
<INPUT TYPE="button" ID="boton1" NAME="botonActivar"
VALUE="Activar" />
<INPUT TYPE="button" ID="boton2" NAME="botonDesactivar"
VALUE="Desactivar" />
</FORM>
```

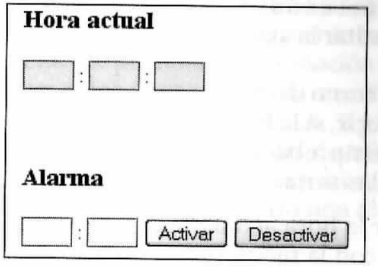


Figura 12.3. Aspecto de nuestro reloj despertador.

También incluiremos una simpática imagen (`alarma.jpg`) que sólo será mostrada cuando salte la alarma, de forma que quedará algo más vistoso que con un simple mensaje. En el momento de cargar la página no mostraremos nada para no revelar lo que está por llegar;).

```
<IMG ID="imagen1" NAME="imagenAlarma" SRC="" />
```

A continuación vamos a crearnos una función JavaScript que muestre la hora actual en su formulario correspondiente, para lo cual tendremos rellenar los tres campos con las horas, minutos y segundos respectivamente.

```
function mostrarHora() {
   // Obtenemos la fecha y hora actuales
   var ahora = new Date();
```

```
    // Colocamos los valores necesarios en el formulario
    with(document.formHora){
        horaActual.value = ahora.getHours();
        minActual.value = ahora.getMinutes();
        segActual.value = ahora.getSeconds();
    }
}
```

Además, como queremos que la hora se actualice para tener siempre la referencia en la pantalla, incluiremos un temporizador (`setInterval`) que se ejecute continuamente cada segundo. Esta sentencia la podemos situar bien dentro del manejador `ONLOAD` del documento o simplemente dentro de una etiqueta `<SCRIPT>` al final del todo. En este ejemplo seguiremos la primera opción por comodidad:

```
<BODY ONLOAD="setInterval('mostrarHora()', 1000);">
```

Antes de almacenar y activar la alarma, le quiero comentar que en nuestro código tendremos declarada una variable de ámbito global (`alarma`) que se encargará de guardar la hora a la que debe saltar la alarma. Dicho esto, sigamos declarando funciones.

Ahora es el turno de crear una que compruebe si debe saltar la alarma, es decir, si la hora actual coincide con la fijada como alarma. Esta comprobación puede hacerse de varias maneras, pero dos de ellas serían:

1. Obtener la hora actual mediante el objeto `Date` y compararla con la fijada en la variable `alarma`.
2. Obtener la hora actual desde el formulario `formHora` y compararla con la fijada en `alarma`.

Ambas son válidas y las diferencias en el código no son muy grandes. Nosotros seguiremos la primera opción, por decir alguna.

Si la hora actual hace que salte la alarma, entonces mostraremos nuestra imagen sorpresa al usuario modificando el atributo `SRC` de `imagenAlarma`. En otro caso, debemos forzar que se vuelva a realizar la comprobación un minuto después (60 segundos = 60000 milisegundos) mediante el uso del método `setTimeout`, del que guardaremos su identificador en otra variable global (`idComprobarAlama`). Veamos todo esto junto:

```
function comprobarAlarma() {
    // Obtenemos la hora actual
    var ahora = new Date();
    // Si hay una alarma fijada
    if (alarma != undefined)  {
        // Si coincide la hora actual con la alarma
        if (ahora.getHours() == alarma.getHours() &&
        ahora.getMinutes() == alarma.getMinutes()) {
            // Mostramos la imagen
            document.imagenAlarma.src = "alarma.jpg";
        } else {
            // En otro caso, volvemos a comprobar en 1 minuto
            idComprobarAlarma =
            setTimeout("comprobarAlarma()", 60000);
        }
    }
}
```

Ahora llega el turno de crear la función que fija la alarma con la hora y minutos que digamos. Como la alarma debe saltar justo cuando se cambia de minuto (pasamos del segundo 59 al 00) no podemos hacer la primera comprobación para dentro de un minuto a partir de la hora actual, ya que tenemos pocas probabilidades de estar en el segundo 00. Para solucionar esto, lo que haremos es calcular la diferencia entre los segundos actuales y 60 (que representa el comienzo del siguiente minuto). De este modo programaremos la primera comprobación para dentro de esa cantidad de segundos pero expresada en milisegundos, ya que es la unidad que maneja `setTimeout`. Y ya que estamos, para rematar la función, dejaremos vacío el atributo SRC de la imagen de alarma, después de fijar la hora y minutos, para evitar que esté visible después de haber saltado una alarma anterior.

```
function fijarAlarma(hora, minutos) {
    // Declaración de variables
    var ahora, primeraComprobacion;
    // Fijamos la alarma
    alarma = new Date();
    alarma.setHours(hora, minutos);
    // Preparamos la primera comprobación de alarma
    ahora = new Date();
    primeraComprobacion = (60 - ahora.getSeconds()) *
1000;
    idComprobarAlarma = setTimeout("comprobarAlarma()",
primeraComprobacion);
    // Dejamos de mostrar la imagen de alarma
    document.imagenAlarma.src = "";
}
```

> **Nota:** *Recuerde que el método `setHours` acepta hasta cuatro parámetros (horas, minutos, segundos y milisegundos), pero sólo es obligatorio el primero.*

Seguidamente nos ponemos `manos a la obra´ con la función que activa la alarma recogiendo los valores introducidos en el formulario para fijar la alarma. Además, antes de hacerlo, podemos aprovechar para establecer unos valores por defecto a los dos campos del formulario por si alguno no tuviese valor.

```
function activarAlarma() {
   // Fijamos valores por defecto
   var hora = 12, minutos = 00;
   // Comprobamos los valores del formulario
   with(document.formAlarma){
      // Guardamos el valor de la hora, si está escrito
      if (horaAlarma.value != "") {
         hora = horaAlarma.value;
      }
      //Guardamos el valor de los minutos, si está escrito
      if (minAlarma.value != "") {
         minutos = minAlarma.value;
      }
      // Fijamos la alarma
      fijarAlarma(hora, minutos);
   }
}
```

Como hemos incluido en nuestro reloj despertador un botón para poder desactivar la alarma antes de que ésta haya saltado, debemos crear otra función para tal tarea. No nos resultará muy complicada puesto que tan sólo debemos impedir que se siga comprobando la alarma, utilizando el método `clearTimeout` junto con el identificador `idComprobarAlarma` que hemos ido guardando en las otras funciones. Asimismo, debemos borrar la hora fijada en la variable `alarma` asignándole un valor nulo (`null`). Aquí también será conveniente ocultar la imagen de alarma puesto que no tiene sentido que aparezca si no hay una alarma fijada.

```
function desactivarAlarma() {
   // Dejamos de comprobar la alarma
   clearTimeout(idComprobarAlarma);
   // Borramos la hora fijada
   alarma = null;
   // Dejamos de mostrar la imagen de alarma
   document.imagenAlarma.src = "";
}
```

Para terminar ya con este ejemplo, sólo nos queda asociar las funciones de activación y desactivación de la alarma al manejador `ONCLICK` de los botones correspondientes.


```
<INPUT TYPE="button" ID="boton1" NAME="botonActivar"
VALUE="Activar" ONCLICK="activarAlarma()" />
<INPUT TYPE="button" ID="boton2" NAME="botonDesactivar"
VALUE="Desactivar" ONCLICK="desactivarAlarma()" />
```

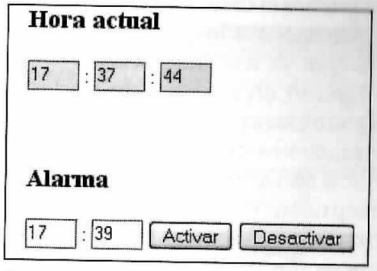


Figura 12.4. Nuestra alarma fijada para dentro de poco.

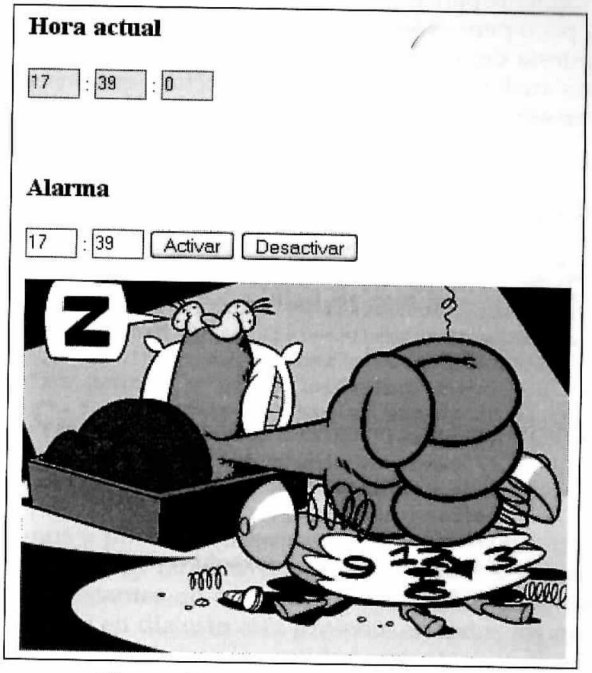


Figura 12.5. ¡La alarma ha saltado!.

Ya tenemos nuestro reloj despertador funcionando, pero todavía lo podemos hacer más real. Si le nombro la palabra *snooze*, ¿le suena de algo? A los madrugadores quizá no, pero

para el resto es el botón más utilizado de todo el despertador y casi siempre va asociado a la frase "bueno, un ratito más y me levanto". Efectivamente, se trata del botón que aplaza la alarma para que suene unos minutos más tarde sin tener que fijar una nueva hora. ¿Le parece si creamos esta nueva funcionalidad en nuestro *script*? Aunque no lo crea, resultará bastante sencillo gracias a todo lo que ya tenemos hecho.

Esta nueva opción en nuestro despertador debe preguntarnos si queremos aplazar la alarma justo cuando ésta salte a la hora programada, por lo que sólo tendremos que modificar la función `comprobarAlarma`. Este cambio consistirá en lanzar al usuario la propuesta de aplazar la alarma, a la que podrá responder afirmativa o negativamente. Si decide no aplazar la alarma, entonces desactivaremos la actual, pero si prefiere dormir un poco más entonces reprogramaremos la alarma automáticamente para dentro de un minuto (sí, soy consciente de que es poco pero es sólo para probar el código). Para recoger la respuesta del usuario usaremos el método `confirm` para después analizar el botón pulsado y actuar en consecuencia. Allá vamos:

```
function comprobarAlarma() {
   // Obtenemos la hora actual
   var ahora = new Date();
   // Si hay una alarma fijada
   if (alarma != undefined) {
      // Si coincide la hora actual con la alarma
      if (ahora.getHours() == alarma.getHours() &&
      ahora.getMinutes() == alarma.getMinutes()) {
         // Mostramos la imagen
         document.imagenAlarma.src = "alarma.jpg";
         // Si quiere aplazar la alarma
         if (confirm("¿Quiere dormir un rato más?")) {
            // Damos un minuto de tregua
            fijarAlarma(alarma.getHours(), alarma.
            getMinutes() + 1);
         } else {
            // En otro caso, desactivamos la alarma
            desactivarAlarma();
         }
      } else {
         // En otro caso, volvemos a comprobar
         // en 1 minuto
         idComprobarAlarma =
         setTimeout("comprobarAlarma()", 60000);
      }
   }
}
```

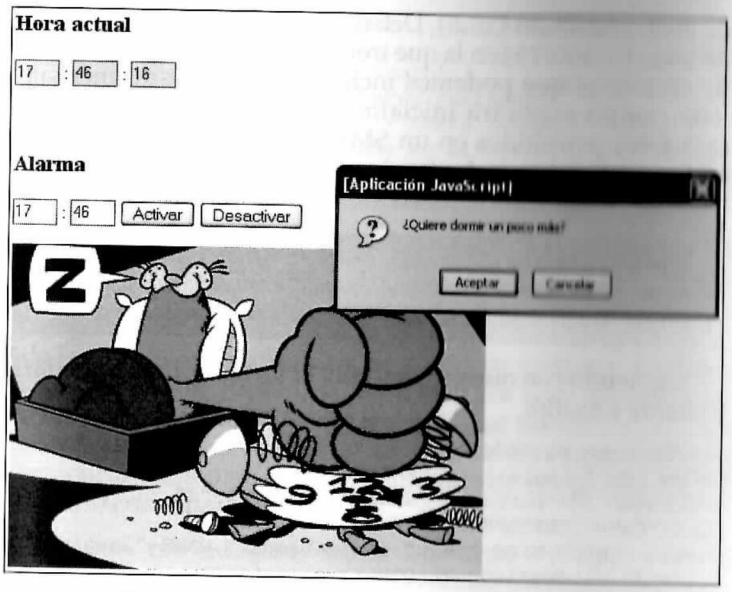



Figura 12.6. Pregunta para aplazar la alarma.

Después de todo lo expuesto, le dejo como ejercicio personal que intente implementar un número máximo de aplazamientos de la alarma, es decir, que podamos hacer que siga saltando la alarma durante una cantidad de minutos extra mediante "*snoozes*".

12.4. Calcular los caracteres restantes de un SMS

Estamos a punto de desarrollar un script cuya funcionalidad le será muy familiar a muchos: calcular y mostrar los caracteres restantes de un texto, centrándonos en el caso de un SMS. Hoy en día esto está presente en todos los móviles, aunque a veces muestran la cantidad restante cuando estamos llegando al límite.

Comenzaremos por simular, sin ningún tipo de alardes, la pantalla del móvil mediante una caja de texto (`<TEXTAREA>`) que sea lo suficientemente grande para ver el texto completo; por ejemplo, tres filas (atributo `ROWS`) y unos sesenta caracteres

de ancho (atributo COLS). Debajo de ella situaremos otra caja de texto (<INPUT>) en la que iremos actualizando la cantidad de caracteres que podemos incluir aún en nuestro mensaje. Este campo mostrará inicialmente el número máximo de caracteres permitidos en un SMS, que es 160. Por supuesto, ambas cajas deben ir dentro de un formulario para poder trabajar con ellas.

> **Nota**: *A modo de curiosidad, los SMS tienen esta limitación de longitud porque inicialmente se pensaron como mensajes de control y no como un servicio añadido para el usuario final.*

Siguiendo con nuestro ejemplo. El formulario nos quedará bastante sencillo:

```
<H3>Escriba su SMS</H3>
<FORM ID="formulario1" NAME="formSMS">
<TEXTAREA ID="entrada1" NAME="sms" ROWS="3"
COLS="60"></TEXTAREA><BR />
Quedan <INPUT TYPE="text" ID="entrada2" NAME="longitud"
SIZE="3" VALUE="160" READONLY> caracteres
</FORM>
```

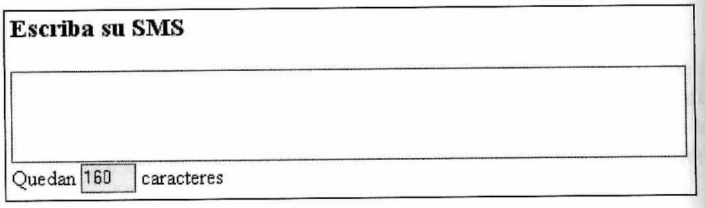


Figura 12.7. Formulario para calcular los caracteres restantes de un SMS.

En cuanto al código JavaScript, tendremos declarada una constante (LONGITUD_MAX) con el número máximo de caracteres que se admiten en la caja de texto. Si un día cambiase esa longitud, no nos costaría nada reflejarlo en nuestro código puesto que sólo habría que cambiar el valor de esta constante, siendo el resto de código totalmente válido.

También existirá en nuestro *script* una función que calculará los caracteres restantes en la caja de texto antes de llegar al límite, por tanto, para hacer ese cálculo se apoyará directamente sobre la constante declarada. El resultado de la operación se escribirá en la caja de texto `longitud` del formulario para que el usuario tenga constancia del espacio que le queda.


```
// Declaración de constante
var LONGITUD_MAX = 160;
function calcularRestantes(texto) {
   // Calculamos los caracteres restantes
   var resto = LONGITUD_MAX - texto.length;
   // Escribimos el resultado en el formulario
   document.formSMS.longitud.value = resto;
}
```

Bien, ya tenemos todo casi preparado. Sólo nos queda el "pequeño" detalle de saber cuándo llamar a la función `calcularRestantes` para que nos actualice el contador. En este caso, el mejor lugar sin duda es dentro del manejador ONKEYUP de la caja de texto `sms`. Gracias a esto, se realizará el cálculo cada vez que se suelte una tecla, bien sea para añadir caracteres o para borrarlos. Como tenemos que decirle a la función sobre qué texto debe realizar la operación, recurriremos a la sentencia `this` para pasar el contenido de la caja (propiedad `value`) como parámetro.

```
<TEXTAREA ID="entrada1" NAME="sms" ROWS="3" COLS="60"
ONKEYUP="calcularRestantes(this.value)"></TEXTAREA>
```

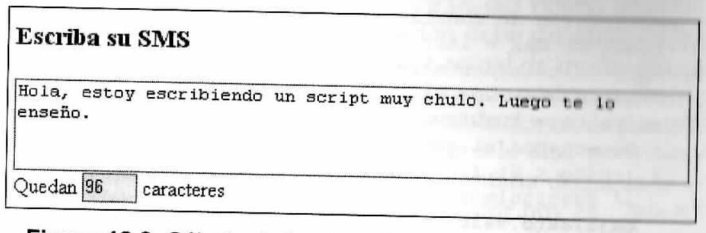


Figura 12.8. Cálculo de los caracteres restantes de un SMS.

Llegados a este punto, si nos ponemos a pensar por un momento nos daremos cuenta de que este *script* está incompleto ya que se muestra el número de caracteres restantes, que está muy bien, pero no limitamos de ninguna manera que el usuario pueda escribir más de 160. ¡Horror! Bueno, bueno... calma, la cosa no es tan complicada de solventar. Está claro que sólo hay que modificar la única función que tenemos (`calcularRestantes` por si no lo recuerda) para que detecte cuándo hemos llegado al límite de caracteres y no nos deje escribir más. Para impedir que el usuario introduzca caracteres extra recurriremos a un truco muy tonto: sustituir el valor de la caja de texto con los primeros 160 caracteres (o los que marque la constante `LONGITUD_MAX`).

```
// Declaración de constante
var LONGITUD_MAX = 160;
function calcularRestantes(texto) {
   // Calculamos los caracteres restantes
   var resto = LONGITUD_MAX - texto.length;
   // Detectamos si hemos llegado al límite
   if (resto < 0) {
      // Sustituimos el texto actual
      document.formSMS.sms.value = texto.
      substr(0, LONGITUD_MAX);
      // Forzamos la llegada al límite
      resto = 0;
   }
   // Escribimos el resultado en el formulario
   document.formSMS.longitud.value = resto;
}
```

Si quisiéramos evitar hacer referencia a la caja de texto del mensaje dentro de la función, tan sólo tendremos que pasar como parámetro la referencia directa al objeto (`this`, a secas) en lugar de su valor. Los cambios en la función serían mínimos.

```
<TEXTAREA ID="entrada1" NAME="sms" ROWS="3"
COLS="60" ONKEYUP="calcularRestantes(this)"></TEXTAREA>
// Declaración de constante
var LONGITUD_MAX = 160;
function calcularRestantes(cajaTexto) {
   // Calculamos los caracteres restantes
   var resto = LONGITUD_MAX - cajaTexto.value.length;
   // Detectamos si hemos llegado al límite
   if (resto < 0) {
      // Sustituimos el texto actual
      cajaTexto.value = cajaTexto.value.
      substr(0, LONGITUD_MAX);
      // Forzamos la llegada al límite
      resto = 0;
   }
   // Escribimos el resultado en el formulario
   document.formSMS.longitud.value = resto;
}
```

12.5. La ventana que escapa

El efecto que se consigue con este ejercicio me resulta particularmente gracioso ya que puede ser origen de alguna pequeña broma en nuestra página, dándole así un toque de humor. La finalidad del código será crear una nueva ventana (a modo de *pop-up*) sobre la que el usuario debe situar

el puntero para hacer clic sobre un botón situado dentro de ella, pero se encontrará con una sorpresa: ¡no será capaz de hacerlo! *Muahahaahaa* (risa malévola). La ventana escapará de él y cambiará de sitio. Ahora veremos cómo conseguir esto mediante JavaScript.

Lo primero es crear la página HTML que se mostrará en la nueva ventana (premio.html) y que tendrá el botón a pulsar.

```
<HTML>
<BODY>
<H3>¿Quieres ganar un coche?</H3>
<FORM ID="formulario1" NAME="formPremio">
<INPUT TYPE="button" ID="boton1" NAME="botonSi" VALUE="Sí">
<INPUT TYPE="button" ID="boton2" NAME="botonNo" VALUE="No">
</FORM>
</BODY>
</HTML>
```

De acuerdo, es un mensaje de lo más traicionero... pero lo que queremos es divertir al usuario.

Para impedir que se pueda hacer clic sobre el botón **Sí** crearemos una pequeña función que desplace la ventana entera, haciendo uso del método `moveTo` del objeto `window`. Para determinar a qué nueva posición debe dirigirse vamos a utilizar los milisegundos de la hora actual de modo que las coordenadas salgan un poco aleatorias ya que podemos obtener un valor entre 0 y 999. Dado que las resoluciones de pantalla suelen tener más píxeles a lo ancho que a lo alto, podríamos correr el riesgo de desplazar nuestra ventana demasiado en vertical (una resolución muy típica hoy por hoy es 1280x800 pixels), así que para evitarlo de un modo sencillo dividiremos los milisegundos entre 2, obteniendo un valor entre 0 y 500 (redondeando). Esta función también iría dentro de la página "`premio.html`", preferiblemente dentro de `<HEAD>`.

```
<HEAD>
<SCRIPT TYPE="text/javascript">
function esquivarUsuario() {
   // Obtenemos la hora actual
   var ahora = new Date();
   // Calculamos las nuevas coordenadas
   var x = ahora.getMilliseconds();
   var y = x/2;
   // Desplazamos la ventana
   window.moveTo(x, y);
}
</SCRIPT>
</HEAD>
```

En cuanto al botón **No**, también crearemos una sencilla función, también dentro de <HEAD>, que muestre un mensaje y seguidamente cierre la ventana:

```
function mostrarMensaje() {
   // Mostramos un mensaje
   alert("¡Nos sorprende su respuesta!");
   // Cerramos la ventana
   window.close();
}
```

Para terminar con esta página, queda asociar estas funciones a los manejadores ONMOUSEOVER del botón trampa, y al manejador ONCLICK del otro botón del formulario.

```
<INPUT TYPE="button" ID="boton1" ="botonSi"
VALUE="Sí" ONMOUSEOVER="esquivarUsuario()">
<INPUT TYPE="button" ID="boton2" NAME="botonNo"
VALUE="No" ONCLICK=" mostrarMensaje()">
```

Con esto tendremos preparada la página lista para ser traviesa con el usuario. Vamos entonces con la página que abrirá este *pop-up* (concurso.html).

En este caso, se mostrará en la pantalla un atractivo enlace que nos invitará a participar en un concurso. Cuando hagamos clic sobre él, se abrirá una nueva ventana que nos revelará el premio al que optamos.

De este modo, primeramente tenemos el enlace:

```
<HTML>
<BODY>
<H3>Concurso del día</H3>
Haga clic <A HREF="">aquí</A> para descubrir el gran
premio que le espera.
</BODY>
</HTML>
```

No hemos puesto una página destino en el enlace ya que queremos abrirla mediante JavaScript para poder fijar algunos atributos como el tamaño o la posición inicial. Ahora vamos con la función que abrirá una ventana y cargará la página "premio.html" en ella. Esta nueva ventana tendrá unas dimensiones concretas y ocultaremos la barra de estado (parámetro opcional status).

```
<HEAD>
<SCRIPT TYPE="text/javascript">
function abrirPremio() {
   // Abrimos una ventana con la página del premio
```



```
   open("premio.html", "concurso",
   "width=250,height=80,status=0");
}
</SCRIPT>
</HEAD>
```

En este ejemplo no necesitamos almacenar el identificador de la ventana devuelto por el método open puesto que el cierre se realiza mediante el botón **No** incluido en la página que abre. Por último, para poder comenzar la broma, asociamos la función al atributo HREF del enlace:

```
Haga clic <A HREF="javascript:abrirPremio()">aquí</A>
para descubrir el gran premio que le espera.
```

¿Listos para ver la reacción de los usuarios?

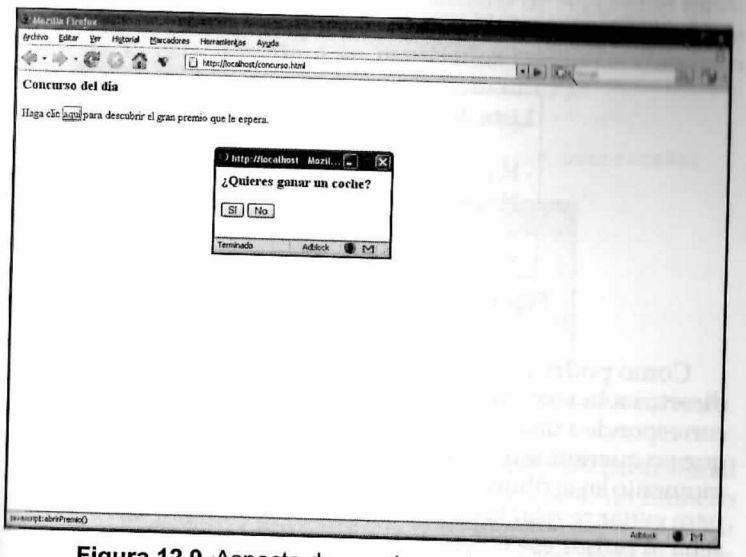


Figura 12.9. Aspecto de nuestra ventana escurridiza.

12.6. Acceso restringido mediante una contraseña

¿Alguna vez ha deseado que sólo un grupo de personas pueda ver ciertas páginas de su Web? Este ejercicio podrá resolver en parte esa necesidad, aunque no se trate de un método infalible.

Lo que queremos conseguir es que las personas que entren a nuestra Web tengan que introducir una contraseña (previamente proporcionada) para poder acceder a secciones concretas.

Vamos a imaginar que tenemos un enlace por cada sección que compone nuestro sitio Web y que la mayoría de ellos nos dirigen directamente a la página correspondiente, salvo uno que solicita primero una contraseña para comprobar si tenemos acceso o no a esa sección. Un conjunto posible de enlaces podría ser el siguiente:

```
<H3>Lista de secciones de mi Web</H3>
- <A HREF="fotos.html">Mis fotos</A><BR />
- <A HREF="canciones.html">Mis canciones preferidas
</A><BR />
- <A HREF="cosas.html">Cosas que me gustan</A><BR />
- <A HREF="JavaScript:void(0)">Zona privada</A>
(con contraseña)
```

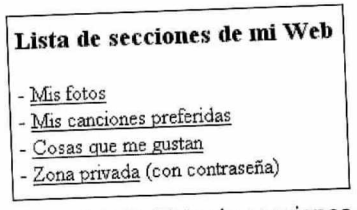


Figura 12.10. Lista de secciones.

Como podrá deducir, los tres primeros enlaces nos llevan directos a la sección que representan, mientras que el último corresponde a una zona privada donde mostramos información que no queremos que vea todo el mundo. Hemos rellenado de momento su atributo `HREF` con una llamada a la función `void` para evitar revelar la página de destino y así solicitar la contraseña al pulsar ese enlace antes de redirigirnos a la sección.

Para comprobar si el visitante actual conoce la contraseña, que le otorga el paso libre a la zona privada, vamos a construir una función JavaScript (no se lo esperaba, ¿verdad?) que recoge el texto introducido por el usuario y lo compara con la contraseña de acceso válida. Si coincide, entonces se le redirigirá hacia la página que alberga nuestro gran secreto. En otro caso, le mostraremos un mensaje advirtiéndole del error.

```
function comprobarAcceso() {
   // Declaración de variables
   var texto;
   // Pedimos la contraseña al usuario
```



```
   texto = prompt("Por favor, introduce la contraseña");
   // Si ha pulsado Cancelar o no ha escrito nada
   if (texto == null || texto == "") {
      // No hacemos nada
      return;
   }
   // Comparamos si la contraseña es válida
   if (texto == "ChachiQueSi") {
      // Redirigimos a la zona privada
      location.href = "zona_privada.html";
   } else {
      // En otro caso, mostramos un mensaje
      alert("Lo siento, la contraseña no es correcta.");
   }
}
```

Ahora que ya tenemos todo el sistema de seguridad montado, sólo queda completar el enlace de la zona privada asociando esta función a su manejador `ONCLICK` (también funcionaría si lo ponemos en el atributo `HREF`).

```
- <A HREF="javascript:void(0)" ONCLICK=
"comprobarAcceso()">Zona privada</A> (con contraseña)
```

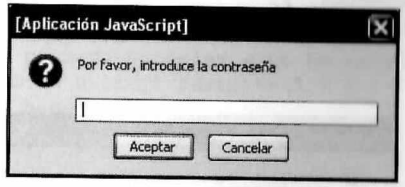


Figura 12.11. Petición de contraseña para entrar en la zona privada.

Opcionalmente, podemos darle un toque más profesional a esta restricción de acceso si le damos al usuario un número máximo de intentos para escribir la contraseña correcta, y así evitar que pruebe todas las que quiera y acabe dando con la buena. Vamos a añadir esta nueva condición a la función `comprobarAcceso`, dando hasta tres intentos para meter la contraseña. Para ello debemos tener una variable de ámbito global que nos sirva como contador de los intentos que se llevan.

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable global
   var numeroIntentos = 0;
   function comprobarAcceso() {
      // Declaración de variables
      var texto;
      // Si hemos llegado al máximo de intentos
```

```
    if (numeroIntentos == 3) {
       // No hacemos nada
       return;
    }
    // Pedimos la contraseña al usuario
    texto = prompt("Por favor, introduce
    la contraseña");
    // Si ha pulsado Cancelar o no ha escrito nada
    if (texto == null || texto == "") {
       // No hacemos nada
       return;
    }
    // Comparamos si la contraseña es válida
    if (texto == "ChachiQueSi") {
       // Redirigimos a la zona privada
       location.href = "zona_privada.html";
    } else {
       // Incrementamos el número de intentos que lleva
       numeroIntentos++;
       // Mostramos un mensaje
       alert("Lo siento, la contraseña no
       es correcta.\n" +
       "Te quedan " + (3 - numeroIntentos) +
       " intentos.");
    }
  }
</SCRIPT>
```

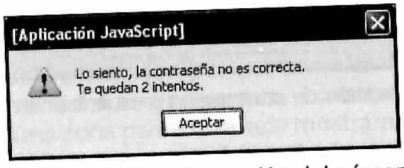


Figura 12.12. Mensaje con información del número de intentos restantes.

Esto tiene mejor pinta... Aunque se podría mejorar un poco más, para hacer más efectiva nuestra restricción.

Ahora mismo el usuario tiene tres intentos y una vez agotados todos ya no se le pedirá más veces la contraseña, pero si el usuario es un poco pícaro y recarga la página volverá a tener otros tres intentos puesto que se carga de nuevo todo el código y el contador se reinicia (¡ouch!). Una forma de controlar mejor si el usuario ha agotado sus intentos para entrar en la zona privada es crear una *cookie* que nos permita "marcarle" y así poder evitar concederle más oportunidades. Para esto debemos recurrir a las funciones `almacenarCookie` y `recuperarCookie` que creamos en el capítulo 8, dedicado a las *cookies*, para facilitarnos la tarea de manejarlas y realizar dos tareas adicionales en la función `comprobarAcceso`:

1. Comprobar que no tiene creada nuestra *cookie* antes de solicitarle la contraseña.
2. Crear nuestra *cookie* al agotar todos los intentos para entrar.

Veamos cómo integrar la comprobación de la cookie:

```
<SCRIPT TYPE="text/javascript">
   // Declaración de variable global
   var numeroIntentos = 0;
   function comprobarAcceso() {
      // Declaración de variables
      var texto;
      // Si está creada la cookie
      if (recuperarCookie("intentosAgotados") != "") {
         // No hacemos nada
         return;
      }
      // Pedimos la contraseña al usuario
      texto = prompt("Por favor, introduce
      la contraseña");
      // Si ha pulsado Cancelar o no ha escrito nada
      if (texto == null || texto == "") {
         // No hacemos nada
         return;
      }
      // Comparamos si la contraseña es válida
      if (texto == "ChachiQueSi") {
         // Redirigimos a la zona privada
         location.href = "zona_privada.html";
      } else {
         // Incrementamos el número de intentos que lleva
         numeroIntentos++;
         // Mostramos un mensaje
         alert("Lo siento, la contraseña no
         es correcta.\n" +
         "Te quedan " + (3 - numeroIntentos) +
         " intentos.");
         // Si hemos llegado al máximo de intentos
         if (numeroIntentos == 3) {
            // Creamos la cookie
            almacenarCookie("intentosAgotados", 1);
         }
      }
   }
</SCRIPT>
```

Ahora la *cookie* se crea con una caducidad de sesión, es decir, hasta que cerremos la ventana del navegador. Por lo tanto, si el usuario accede después a la página volverá a tener tres intentos. Esto se puede solucionar ampliando el tiempo de duración de la *cookie* mediante el tercer parámetro de la función `almacenarCookie`, donde podemos decir que la *cookie* se mantenga durante dos horas, cinco días, etc.

```
// Duración de 2 horas
var caducidad = new Date();
caducidad.setHours(hoy.getHours() + 2);
almacenarCookie("intentosAgotados", 1, caducidad);
// Duración de 5 días
var caducidad = new Date();
caducidad.setDate(hoy.getDate() + 5);
almacenarCookie("intentosAgotados", 1, caducidad);
```

12.7. Las tres en raya

En este último ejercicio vamos a recrear el famoso juego de las tres en raya o *tic-tac-toe* utilizando un poco de HTML y JavaScript. Como bien sabrá, la finalidad de este juego para dos personas es situar tres figuras en línea de forma vertical, horizontal o incluso diagonal dentro de un tablero cuadrado de tres casillas de ancho por tres de alto (haciendo un total de nueve). Obviamente, el primer jugador que lo consiga será el vencedor, aunque puede darse la posibilidad de que ninguno de ellos lo haga por estar todas las casillas rellenas sin haber logrado una línea.

Una vez claras las reglas, lo primero de todo es dibujar el tablero en nuestra página, para lo que recurriremos a lo más sencillo: una tabla de tres filas por tres columnas. También aprovecharemos para identificar cada celda (que harán de casillas del tablero) de acuerdo al siguiente esquema:

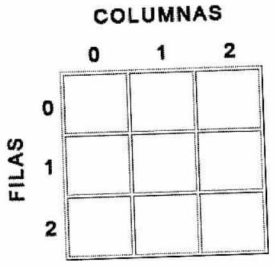


Figura 12.13. Esquema de nombrado de celdas.

Cada celda se identificará mediante la concatenación de su número de fila y su número de columna, por lo que la casilla situada arriba a la izquierda es la "00", la de su derecha la "01" y la siguiente la "02".

```
<H3>Juego de las 3 en raya</H3>
<TABLE BORDER="1">
<TR>
   <TD ID="casilla00" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla01" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla02" STYLE="width: 50px;
   height: 50px"></TD>
</TR>
<TR>
   <TD ID="casilla10" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla11" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla12" STYLE="width: 50px;
   height: 50px"></TD>
</TR>
<TR>
   <TD ID="casilla20" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla21" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla22" STYLE="width: 50px;
   height: 50px"></TD>
</TR>
</TABLE>
```

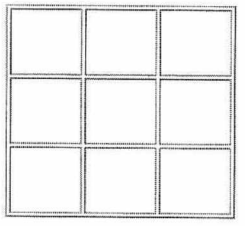


Figura 12.14. Nuestro tablero de juego.

Una buena forma de representar este tablero en nuestro script sería crear un objeto con sus propiedades y métodos específicos, de manera que podremos aplicar todo lo que

hemos ido aprendiendo a lo largo del libro. Haciendo un análisis de una partida real podemos identificar las siguientes propiedades:

- `casillas`: Representarán, obviamente, todas las posiciones en las que se puede situar una figura para lograr el objetivo del juego. La forma de recrearlo será a través de un *array* bidimensional que veremos más adelante. En cada posición se almacenará la figura que la ocupa.
- `turno`: Indica cuál de las dos figuras posibles tiene el turno actual. Las representaremos mediante los caracteres "X" y "O".
- `numMovimientos`: Es un contador del número de movimientos o figuras que ya se han situado sobre el tablero. Esto nos ayudará a determinar cuándo ha terminado la partida por no quedar huecos libres. Además, comprobar esta propiedad será más rápido que recorrer el tablero entero en busca del número de casillas sin utilizar.

En cuanto a los métodos, también tendremos los mismos que en la vida real:

- `inicializarTablero`: Deja el tablero vacío listo para empezar una nueva partida y también establece a qué figura corresponde el primer turno. Asimismo, establece que el número de movimientos realizados es cero puesto que vamos a comenzar a jugar.
- `detectarLinea`: Este método se encarga de analizar la situación actual del tablero para comprobar si hay una jugada vencedora, es decir, alguien ha conseguido las tres en raya. Para ello debe examinar las casillas en horizontal, vertical y diagonal en busca de tres figuras iguales seguidas. Con el fin de simplificar la apariencia del código, crearemos tres pequeños métodos auxiliares para cada uno de los posibles tipos de jugadas: `detectarLineaHorizontal`, `detectarLineaVertical` y `detectarLineaDiagonal`.
- `marcarMovimiento`: Guarda una jugada sobre el tablero, fijando en la casilla indicada la figura del turno actual e incrementando el número de movimientos realizados en la partida. Aquí también definiremos un método auxiliar (`continuarPartida`) que se encargará de comprobar si este movimiento causa la victoria del jugador actual mediante el método `detectarLinea`, dando por finalizada la partida. Si no es así, entonces debe cambiar el turno a la otra figura, siempre y cuando haya posiciones libres donde ponerla.

Después de este análisis, nuestro objeto quedaría formado así:

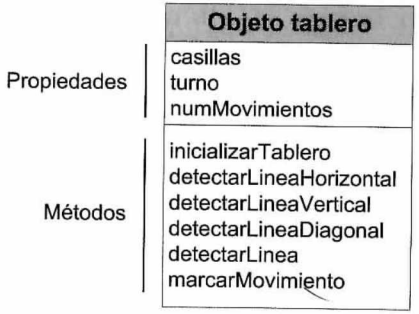


Figura 12.15. Nuestro objeto tablero.

Ahora ya podemos comenzar a trasladar todas estas palabras a código JavaScript. Empezaremos por lo más básico, crear la estructura del objeto `Tablero`:

```
function Tablero() {
   // Lista de propiedades
   // Lista de métodos
}
```

Después vendrán las propiedades que, recuerde, deben ir junto con la sentencia `this`.

```
// Lista de propiedades
this.casillas;
this.turno;
this.numMovimientos;
```

Como dijimos al definir la propiedad `casillas`, el tablero estará representado mediante un *array* bidimensional de 3x3 posiciones. Para conseguir esto debemos crear primero un *array* unidimensional de tres posiciones para después convertir cada una de ellas en otro *array* unidimensional de tres posiciones, con lo que obtenemos un total de nueve posiciones.

```
this.casillas = new Array(3);
for(var i=0; i<3; i++) {
   this.casillas[i] = new Array(3);
}
```

Hecho. No es necesario inicializar las otras propiedades ya que lo haremos desde su método `inicializarTablero`, el cual debe vaciar el tablero primeramente, acción que comprenderá dos fases:

1. Dejar el *array* `casillas` sin figuras en todas las posiciones, sustituyéndolas por una cadena vacía ("").
2. Limpiar la tabla HTML que representa el tablero para que los jugadores puedan volver a colocar sus figuras.

Como ambas fases están directamente relacionadas, las realizaremos en un único paso. Lo haremos definiendo dos bucles `for`: uno para recorrer las filas y otro para las columnas, de manera que pasaremos por todas las posiciones posibles. Además, como hemos creado el identificador de cada casilla de la tabla HTML como la combinación del número de fila y de columna, nos será realmente fácil acceder a ellos mediante estos bucles y el método `getElementById`. Para borrar la figura de la celda usaremos su propiedad `innerHTML` y el espacio en blanco de HTML (` `).

```
// Limpiamos las casillas
for(var fila=0; fila<3; fila++) {
   for(var col=0; col<3; col++) {
      this.casillas[fila][col] = "";
      document.getElementById("casilla" + fila + col).
      innerHTML = " ";
   }
}
```

Adicionalmente, dentro de este método tenemos que establecer a qué figura corresponde el primer turno. Para hacerlo de una forma original recurriremos al azar gracias al método `random` del objeto `Math`, de modo que, una vez redondeado el número aleatorio (método `round`), estableceremos el turno para la figura "O" si sale un cero, o por el contrario a la figura "X" si sale un uno. Sólo podremos obtener esos dos valores puesto que `random` da un número real comprendido entre ellos que además rendondearemos.

```
// Establecemos el turno
var azar = Math.round(Math.random());
if (azar == 0) {
   this.turno = "O";
} else {
   this.turno = "X";
}
```


Por último, el método también iguala a cero el número total de movimientos en la jugada actual. Esto es fácil:

```
// Reiniciamos el número de movimientos
this.numMovimientos = 0;
```

Veamos entonces la construcción completa del método para que tenga una visión global del mismo.

```
this.inicializarTablero = function (){
   // Declaramos las variables
   var azar = Math.round(Math.random());
   // Limpiamos las casillas
   for(var fila=0; fila<3; fila++) {
      for(var col=0; col<3; col++) {
         this.casillas[fila][col] = "";
         document.getElementById("casilla" + fila +
         col).innerHTML = " ";
      }
   }
   // Establecemos el turno
   if (azar == 0) {
      this.turno = "O";
   } else {
      this.turno = "X";
   }
   // Reiniciamos el número de movimientos
   this.numMovimientos = 0;
}
```

Los siguientes métodos que implementaremos serán los relacionados con la comprobación de una jugada ganadora. Empezaremos por la más sencilla: `detectarLineaDiagonal`. Para saber si existe una jugada en diagonal que finalice la partida tenemos que comprobar las dos únicas diagonales que existen en el tablero:

1. La que va desde la casilla superior izquierda (fila 0 y columna 0) hasta la inferior derecha (fila 2 y columna 2).
2. La que va desde la casilla superior derecha (fila 0 y columna 2) hasta la inferior izquierda (fila 2 y columna 0).

Como es indiferente el sentido dentro de la diagonal (arriba hacia abajo o abajo hacia arriba) no habrá que hacer otro tipo de validaciones.

Para que una jugada sea válida en diagonal, sus casillas no deben estar vacías y además contener la misma figura en todas ellas. En realidad nos basta con comprobar si la primera

casilla no está vacía para después comprobar que todas tienen la misma figura, ya que si contienen una figura no estaremos comparando con el valor vacío.

```
this.detectarLineaDiagonal  = function () {
   // Comprobamos la diagonal hacia la derecha
   if (this.casillas[0][0] != "" &&
      this.casillas[0][0] == this.casillas[1][1] &&
      this.casillas[1][1] == this.casillas[2][2]) {
         return true;
   }
   // Comprobamos la diagonal hacia la izquierda
   if (this.casillas[0][2] != "" &&
      this.casillas[0][2] == this.casillas[1][1] &&
      this.casillas[1][1] == this.casillas[2][0]) {
         return true;
   }
   // En otro caso, no hemos encontrado una línea
   return false;
}
```

Ahora nos iremos en las que realizan esta comprobación, pero de forma horizontal o vertical, cuyo funcionamiento es muy similar. El método `detectarLineaHorizontal` debe recorrer cada fila para comprobar si la figura de todas las columnas es la misma, mientras que `detectarLineaVertical` recorrerá cada columna comprobando las figuras en cada fila. En ambas también nos aseguraremos de que la primera casilla no está vacía, tal y como hicimos al escribir `detectarLineaDiagonal`. Estos métodos quedarían entonces así:

```
this.detectarLineaHorizontal  = function () {
   // Comprobamos cada fila
   for(var fila=0; fila<3; fila++) {
      if (this.casillas[fila][0] != "" &&
         this.casillas[fila][0] == this.casillas[fila]
         [1] &&
         this.casillas[fila][1] == this.casillas[fila]
         [2]) {
            return true;
      }
   }
   // En otro caso, no hemos encontrado una línea
   return false;
}
this.detectarLineaVertical  = function () {
   // Comprobamos cada columna
   for(var col=0; col<3; col++) {
      if (this.casillas[0][col] != "" &&
         this.casillas[0][col] == this.casillas[1][col] &&
```



```
         this.casillas[1][col] == this.casillas[2][col]) {
            return true;
      }
   }
   // En otro caso, no hemos encontrado una línea
   return false;
}
```

Con todos estos métodos auxiliares definidos, el método `detectarLinea` nos queda la mar de sencillo de implementar puesto que bastará con comprobar cada uno de los tres posibles tipos de líneas (horizontales, verticales y diagonales).

```
this.detectarLinea = function () {
   return this.detectarLineaHorizontal() ||
      this.detectarLineaVertical() ||
      this.detectarLineaDiagonal();
}
```

Escrito de esta forma, en cuanto haya una jugada válida en alguna de las tres situaciones, el método `detectarLinea` revelará que existe.

¿Qué tal por ahora? Espero que bien. Vamos entonces con el método `continuarPartida` que comprueba si el último movimiento otorga la victoria al jugador actual (método `detectarLinea`), mostrando un mensaje de enhorabuena al ganador y dando por finalizada la partida. En otro caso, cambiará el turno si quedan posiciones libres en el tablero (o lo que es lo mismo, no hemos hecho el máximo número de movimientos) o muestra un mensaje indicando que la partida ha terminado si no es así.

```
this.continuarPartida = function () {
   // Si hay tres en raya
   if (this.detectarLinea()) {
      // Finalizamos la partida
      this.numMovimientos = 9;
      // Damos la enhorabuena
      alert("¡Tres en raya! Enhorabuena.");
   } else {
      /// Si no ha terminado la partida
      if (this.numMovimientos < 9) {
         // Cambiamos el turno
         if (this.turno == "O") {
            this.turno = "X";
         } else {
            this.turno = "O";
         }
      } else {
         // En otro caso, avisamos del final de la partida
```

```
            alert("La partida ha terminado sin vencedor.");
        }
    }
}
```

Escribamos finalmente el último método que nos queda por incluir: `marcarMovimiento`. Aquí debemos registrar la jugada realizada sobre el tablero, guardando la figura del turno actual en la casilla que reciba como parámetro (si no está ocupada y no ha terminado la partida, claro) y también sumando el movimiento al contador general (propiedad `numMovimientos`). Inmediatamente después hay que mirar si la partida sigue o no, que lo dejaremos en manos del método `continuarPartida`.

```
this.marcarMovimiento = function (fila, col) {
    // Si no ha terminado la partida
    if (this.numMovimientos < 9) {
        // Si la casilla no está ocupada
        if (this.casillas[fila][col] == "") {
            // La ocupamos
            this.casillas[fila][col] = this.turno;
            document.getElementById("casilla" + fila +
            col).innerHTML = this.turno;
            // Incrementamos el número de movimientos
            this.numMovimientos++;
            // Comprobar si sigue la partida
            this.continuarPartida();
        }
    }
}
```

Bueno, pues nuestro objeto `Tablero` ya está por fin completo. Tan sólo nos quedaría preparar la tabla HTML (que hace de tablero) para que el usuario pueda interactuar con él y echarse una partidita con alguien.

Primeramente hay que crear un nuevo tablero en una variable global (`tresEnRaya`) e implementar una función que se encargará de inicializarlo. Esta función la asociaremos al evento `ONLOAD` del documento para que realice las operaciones al cargar la página.

```
<HEAD>
<SCRIPT TYPE="text/javascript">
// Declaramos una variable global
var tresEnRaya;
function crearTablero() {
    // Creamos una instancia del tablero
    tresEnRaya = new Tablero();
```



```
    // Inicializamos el tablero
    tresEnRaya.inicializarTablero();
}
</SCRIPT>
</HEAD>
<BODY ONLOAD="crearTablero()">
```

Seguidamente, y ya para terminar, hay que asociar al evento `click` de cada una de las celdas (etiquetas `<TD>`) otra función que registre el movimiento realizado, ya que es donde el usuario va a pulsar para situar las figuras. Sin embargo, esta vez no serán manejadores a través de HTML, sino de Javascript puesto que vamos a asociar la misma función a todos ellos y una futura modificación, por ligera que sea, nos obligaría a cambiar los nueve manejadores. Haciéndolo a través de Javascript podemos asignar de `un plumazo´ la misma función a todos, evitando que una modificación en su código no afecte a nueve funciones sino a una sola.

De este modo, veamos qué debe hacer la función que registra el movimiento:

- Obtener la fila y la columna a la que corresponde la celda actual a través de su identificador, dado que esta información corresponde a sus dos últimos caracteres.
- Llamar al método `marcarMovimiento` del objeto `Tablero` para que el movimiento quede registrado, pasando como parámetros la fila y columna obtenidas en el paso anterior.

Antes de ver estos pasos transformados en código vamos a explicar cómo será su asignación a los manejadores. Después veremos ambas operaciones juntas.

Como todos los elementos HTML a los que tenemos que incluir un manejador son celdas de una tabla, podemos aprovecharnos del método `getElementsByTagName` que nos ofrece el objeto `document` gracias al cual podremos agrupar en un *array* todas las celdas para después asignarles el manejador una a una dentro de un bucle `for`. La única restricción que supone hacer esto es que en la página no puede haber más celdas que no correspondan al tablero, puesto que les asignaríamos el mismo manejador erróneamente. Si eso fuese inevitable podríamos solucionarlo haciendo un pequeño filtro a la hora de asignar el manejador, comprobando si el identificador de la celda comienza por "casilla" por ejemplo (es una idea de tantas).

Vamos, ahora sí, a escribir el código que dotará de vida al tablero de nuestra página:

```
// Obtenemos todas las celdas de la tabla
var celdas = document.getElementsByTagName("TD");
// Asignamos el manejador a todas las celdas
for(var i=0; i<celdas.length; i++) {
   // Asignamos el manejador
   celdas[i].onclick = function (){
      // Obtenemos la fila (penúltimo carácter)
      var fila = this.id.substr(-2, 1);
      // Obtenemos la columna (último carácter)
      var columna = this.id.substr(-1);
      // Registramos la jugada en el tablero
      tresEnRaya.marcarMovimiento(fila, columna);
   }
}
```

Sólo destacar que este último código debe situarse al final de la página ya que necesita que exista la tabla para poder asignarle un manejador. Si lo pusiéramos al principio no encontraría ningún elemento de celda puesto que se definirían justo a continuación. Otra opción sería incluirlo como parte del código que se ejecuta con el manejador `ONLOAD` del documento.

Pues dicho esto, ya estaría todo listo para jugar. ¿Qué, una partidita:)?

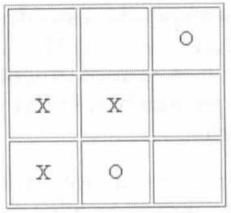


Figura 12.16. Transcurso de una partida.

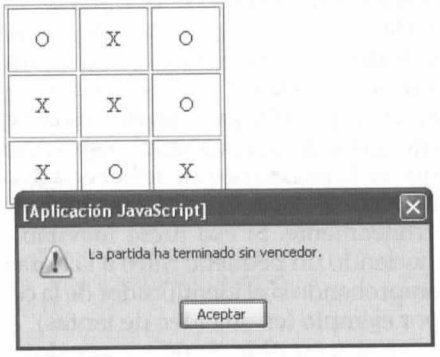


Figura 12.17. Final de partida sin vencedor.

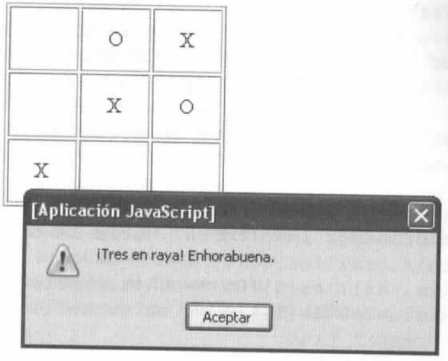


Figura 12.18. Final de partida con vencedor.

Me gustaría además aprovechar este ejemplo para mostrarle cómo puede reutilizar fácilmente el objeto tablero en otras páginas, sin tener que copiar el código en ella. Si recuerda del capítulo de introducción se comentó que se podía realizar esto utilizando el atributo `SRC` de la etiqueta `<SCRIPT>` junto con un fichero de extensión ".js". Para este ejemplo, el fichero podría llamarse "objeto_tablero.js" que incluiría el siguiente código (así de paso puede ver todo agrupado, en lugar de desperdigado):

```
// Declaración del objeto tablero
function Tablero() {
   // Lista de propiedades
   this.casillas = new Array(3);
   for(var i=0; i<3; i++) {
      this.casillas[i] = new Array(3);
   }
   this.turno;
   this.numMovimientos;

   // Lista de métodos
   this.inicializarTablero = function (){
      // Declaramos las variables
      var azar = Math.round(Math.random());
      // Limpiamos las casillas
      for(var fila=0; fila<3; fila++) {
         for(var col=0; col<3; col++) {
            this.casillas[fila][col] = "";
            document.getElementById("casilla" + fila +
            col).innerHTML = " ";
         }
      }
      // Establecemos el turno
```

```
    if (azar == 0) {
        this.turno = "O";
    } else {
        this.turno = "X";
    }
    // Reiniciamos el número de movimientos
    this.numMovimientos = 0;
}
this.detectarLineaDiagonal  = function () {
    // Comprobamos la diagonal hacia la derecha
    if (this.casillas[0][0] != "" &&
        this.casillas[0][0] == this.casillas[1][1] &&
        this.casillas[1][1] == this.casillas[2][2]) {
            return true;
    }
    // Comprobamos la diagonal hacia la izquierda
    if (this.casillas[0][2] != "" &&
        this.casillas[0][2] == this.casillas[1][1] &&
        this.casillas[1][1] == this.casillas[2][0]) {
            return true;
    }
    // En otro caso, no hemos encontrado una línea
    return false;
}
this.detectarLineaHorizontal  = function () {
    // Comprobamos cada fila
    for(var fila=0; fila<3; fila++) {
        if (this.casillas[fila][0] != "" &&
            this.casillas[fila][0] == this.
            casillas[fila][1] &&
            this.casillas[fila][1] == this.
            casillas[fila][2]) {
                return true;
        }
    }
    // En otro caso, no hemos encontrado una línea
    return false;
}
this.detectarLineaVertical  = function () {
    // Comprobamos cada columna
    for(var col=0; col<3; col++) {
        if (this.casillas[0][col] != "" &&
            this.casillas[0][col] == this.casillas[1]
            [col] &&
            this.casillas[1][col] == this.casillas[2]
            [col]) {
                return true;
        }
    }
    // En otro caso, no hemos encontrado una línea
    return false;
```



```
}
this.detectarLinea = function () {
    return this.detectarLineaHorizontal() ||
        this.detectarLineaVertical() ||
        this.detectarLineaDiagonal();
}
this.continuarPartida = function () {
    // Si hay tres en raya
    if (this.detectarLinea()) {
        // Finalizamos la partida
        this.numMovimientos = 9;
        // Damos la enhorabuena
        alert("¡Tres en raya! Enhorabuena.");
    } else {
        /// Si no ha terminado la partida
        if (this.numMovimientos < 9) {
        // Cambiamos el turno
            if (this.turno == "O") {
                this.turno = "X";
            } else {
                this.turno = "O";
            }
        } else {
            // En otro caso, avisamos del final
            // de la partida
            alert("La partida ha terminado sin
            vencedor.");
        }
    }
}
this.marcarMovimiento = function (fila, col) {
    // Si no ha terminado la partida
    if (this.numMovimientos < 9) {
        // Si la casilla no está ocupada
        if (this.casillas[fila][col] == "") {
            // La ocupamos
            this.casillas[fila][col] = this.turno;
            document.getElementById("casilla" + fila +
            col).innerHTML = this.turno;
            // Incrementamos el número de movimientos
            this.numMovimientos++;
            // Comprobar si sigue la partida
            this.continuarPartida();
        }
    }
}
}
```

Gracias a esto, el código de la página quedará más despejado puesto que sustituiríamos todo lo anterior por esta única línea, dentro de <HEAD>, quedando la página completa como sigue:

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript" SRC="objeto_tablero.
js"></SCRIPT>
<SCRIPT TYPE="text/javascript">
// Declaramos una variable global
var tresEnRaya;
function crearTablero() {
   // Creamos una instancia del tablero
   tresEnRaya = new tablero();
   // Inicializamos el tablero
   tresEnRaya.inicializarTablero();
}
</SCRIPT>
</HEAD>
<BODY ONLOAD="crearTablero()">
<H3>Juego de las 3 en raya</H3>
<TABLE BORDER="1">
<TR>
   <TD ID="casilla00" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla01" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla02" STYLE="width: 50px;
   height: 50px"></TD>
</TR>
<TR>
   <TD ID="casilla10" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla11" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla12" STYLE="width: 50px;
   height: 50px"></TD>
</TR>
<TR>
   <TD ID="casilla20" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla21" STYLE="width: 50px;
   height: 50px"></TD>
   <TD ID="casilla22" STYLE="width: 50px;
   height: 50px"></TD>
</TR>
</TABLE>
<SCRIPT TYPE="text/javascript">
// Obtenemos todas las celdas de la tabla
var celdas = document.getElementsByTagName("TD");
// Asignamos el manejador a todas las celdas
for(var i=0; i<celdas.length; i++) {
   // Asignamos el manejador
   celdas[i].onclick = function (){
```



```
      // Obtenemos la fila (penúltimo carácter)
      var fila = this.id.substr(-2, 1);
      // Obtenemos la columna (último carácter)
      var columna = this.id.substr(-1);
      // Registramos la jugada en el tablero
      tresEnRaya.marcarMovimiento(fila, columna);
   }
}
</SCRIPT>
</BODY>
</HTML>
```

No quiero resultarle pesado, ya que este ejemplo ha sido bastante largo, pero no me gustaría despedirme sin antes darle otra pequeña idea que mejoraría a nivel visual nuestro maravilloso *script*: poner en negrita la jugada ganadora;).

Si lo piensa detenidamente y analizando el código que tenemos, seguro que encuentra un sitio donde se puede incluir este cambio sin mucho esfuerzo: las funciones que detectan si hay una jugada ganadora. Dentro de ellas podemos aprovechar para saber la combinación ganadora y ponerla en negrita al mismo tiempo. Y pensará ¿cómo lo ponemos en negrita? Pues muy fácilmente, sustituyendo el contenido de las celdas correspondiente con la misma figura pasando por el método `bold` del objeto `String`.

```
this.detectarLineaDiagonal  = function () {
   // Comprobamos la diagonal hacia la derecha
   if (this.casillas[0][0] != "" &&
      this.casillas[0][0] == this.casillas[1][1] &&
      this.casillas[1][1] == this.casillas[2][2]) {
         // Ponemos las celdas en negrita
         document.getElementById("casilla00").
         innerHTML = this.casillas[0][0].bold();
         document.getElementById("casilla11").
         innerHTML = this.casillas[1][1].bold();
         document.getElementById("casilla22").
         innerHTML = this.casillas[2][2].bold();
         return true;
   }
   // Comprobamos la diagonal hacia la izquierda
   if (this.casillas[0][2] != "" &&
      this.casillas[0][2] == this.casillas[1][1] &&
      this.casillas[1][1] == this.casillas[2][0]) {
         // Ponemos las celdas en negrita
         document.getElementById("casilla02").
         innerHTML = this.casillas[0][2].bold();
         document.getElementById("casilla11").
         innerHTML = this.casillas[1][1].bold();
```

```
            document.getElementById("casilla20").
            innerHTML = this.casillas[2][0].bold();
            return true;
        }
        // En otro caso, no hemos encontrado una línea
        return false;
    }
    this.detectarLineaHorizontal  = function () {
        // Comprobamos cada fila
        for(var fila=0; fila<3; fila++) {
            if (this.casillas[fila][0] != "" &&
                this.casillas[fila][0] == this.casillas[fila]
                [1] &&
                this.casillas[fila][1] == this.casillas[fila]
                [2]) {
                    // Ponemos las celdas en negrita
                    document.getElementById("casilla" + fila +
                    "0").innerHTML = this.casillas[fila][0].
                    bold();
                    document.getElementById("casilla" + fila +
                    "1").innerHTML = this.casillas[fila][1].
                    bold();
                    document.getElementById("casilla" + fila +
                    "2").innerHTML = this.casillas[fila][2].
                    bold();
                    return true;
            }
        }
        // En otro caso, no hemos encontrado una línea
        return false;
    }
    this.detectarLineaVertical  = function () {
        // Comprobamos cada columna
        for(var col=0; col<3; col++) {
            if (this.casillas[0][col] != "" &&
                this.casillas[0][col] == this.casillas[1][col] &&
                this.casillas[1][col] == this.casillas[2][col]) {
                    // Ponemos las celdas en negrita
                    document.getElementById("casilla0" + col).
                    innerHTML = this.casillas[0][col].bold();
                    document.getElementById("casilla1" + col).
                    innerHTML = this.casillas[1][col].bold();
                    document.getElementById("casilla2" + col).
                    innerHTML = this.casillas[2][col].bold();
                    return true;
            }
        }
        // En otro caso, no hemos encontrado una línea
        return false;
    }
```

13

Un paso más allá de JavaScript

Con todo lo que ha visto en esta guía, tengo la convicción de que está preparado para enfrentarse a muchos retos e incluso seguir avanzando en el dominio de JavaScript porque, aunque no lo crea, esto es casi sólo la punta de iceberg. Pero no tema puesto que ya tiene unos buenos cimientos para comprender rápidamente las múltiples técnicas que se basan en este lenguaje.

En este capítulo pretendo mostrarle las más inmediatas cuando se quiere subir el nivel de nuestras páginas Web.

13.1. DHTML

Seguro que le suena haberlo leído alguna vez por está guía, junto a otro montón de palabros del mismo calibre. DHTML, o HTML Dinámico, es una poderosa combinación de HTML, hojas de estilo CSS y JavaScript que nos va a permitir incorporar variedad de efectos en nuestras páginas, así como la posibilidad de aumentar la interacción con el usuario que la visita. Por tanto, nuestras páginas dejarán de ser un simple y aburrido conjunto de texto, imágenes, tablas y otros elementos estáticos.

Además, se suele hacer un uso muy intensivo del DOM (es amplísimo) llegando a poder manipular el contenido de una página casi por completo.

Voy a exponer un ejemplo sin adentrarme mucho, que para eso hay libros específicos muy buenos, con el objetivo de que vea un poco el potencial de DHTML. No creo que le suponga problema el comprender la gran mayoría del código.

Partiremos con una página en la que habrá varias capas con palabras en inglés que, al pulsar sobre ellas, aparecerá otro recuadro con la traducción al español. Estas traducciones estarán previamente cargadas en nuestra página como capas ocultas, de modo que la pulsación sobre una caja en inglés hará visible su correspondiente traducción.

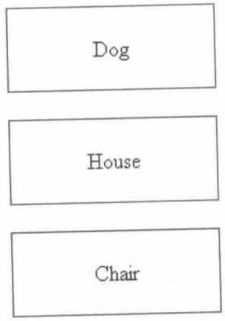


Figura 13.1. Página inicial.

También estará presente algo que no hemos utilizado en este libro: estilos CSS. Serán muy sencillos para no complicar el ejemplo.

```
<HTML>
<HEAD>
   <STYLE>
      .visible {
         display: block;
      }
      .oculto {
         display: none;
      }
      .caja {
         border: 1px solid red;
         width: 150px;
         height: 60px;
         cursor: pointer;
      }
      .traduccion {
         position: absolute;
         border: 2px solid green;
         background-color: #F0F0F0;
         width: 150px;
         height: 60px;
      }
   </STYLE>
</HEAD>
<BODY>
<DIV ID="traduccion1" CLASS="traduccion oculto">Perro</DIV>
<DIV ID="traduccion2" CLASS="traduccion oculto">Casa</DIV>
<DIV ID="traduccion3" CLASS="traduccion oculto">Silla</DIV>
<DIV CLASS="caja">Dog</DIV><br>
<DIV CLASS="caja">House</DIV><br>
<DIV CLASS="caja">Chair</DIV>
</BODY>
</HTML>
```

Le paso a comentar brevemente cada estilo que se ha incluido, por si no conoce estas definiciones.

- visible: Como puede imaginar, nos permite hacer visible una capa.
- oculto: Muy obvia también, nos permite ocultar una capa.
- caja: Es el estilo que aplicaremos a las cajas con el texto en inglés. Se ajustarán sus dimensiones, se pondrá un borde de color rojo y haremos que aparezca el cursor en forma de mano cuando nos pongamos sobre una de estas cajas.
- traduccion: Muy similar al anterior, pero aplicada a las traducciones en español. Como añadido, se colorea el fondo de la capa con un gris claro (`#F0F0F0`) y se fija su posición como absoluta, es decir, nos permitirá cambiar su posición respecto al documento.

Por lo demás, nada raro excepto el uso del atributo `CLASS` que sirve para asignar estilos a un elemento de la página.

¡Pero vamos a la "chicha"! Tendremos que crear dos funciones para mostrar u ocultar las traducciones. Para ello tendremos que cambiar la propiedad `className` de la capa (nueva para nosotros) con los estilos que correspondan en cada caso. Además, para darle un poco de estilo, vamos a hacer que la capa aparezca cerca de la posición del cursor cuando se hace clic. Si recuerda un poco el capítulo 10 referente a los eventos, se mencionó que existía un objeto `event` que almacena información acerca del evento que se acababa de producir. Pues bien, aquí sí lo usaremos para saber la posición del cursor y colocar nuestra traducción muy cerca con las propiedades `top` y `left`, que pertenecen a su vez a `style` (así que el acceso sería mediante `style.top` y `style.left`)

```
<SCRIPT TYPE="text/javascript">
   function mostrarTraduccion(evento, idCapa) {
      // Obtenemos la posición del cursor
      var topCapa = evento.clientY;
      var leftCapa = evento.clientX + 20;
      // Hacemos visible la capa
      document.getElementById(idCapa).className =
      "traduccion visible";
      // Colocamos la capa
      document.getElementById(idCapa).style.top =
      topCapa;
      document.getElementById(idCapa).style.left =
      leftCapa;
   }
   function ocultarTraduccion(idCapa) {
      // Ocultamos la capa
      document.getElementById(idCapa).className =
      "traduccion oculto";
   }
</SCRIPT>
```

No es demasiado difícil, ¿verdad? Ya sólo queda el remate final, asignando manejadores de eventos a las cajas con textos en inglés. Usaremos el evento `MOUSEDOWN` en lugar del clásico `CLICK` para poder capturar el objeto `event`.

```
<DIV CLASS="caja" ONMOUSEDOWN="mostrarTraduccion(e
vent, 'traduccion1')" ONMOUSEOUT="ocultarTraduccion
('traduccion1')">Dog</DIV><br>
<DIV CLASS="caja" ONMOUSEDOWN="mostrarTraduccion(e
vent, 'traduccion2')" ONMOUSEOUT="ocultarTraduccion
('traduccion2')">House</DIV><br>
<DIV CLASS="caja" ONMOUSEDOWN="mostrarTraduccion(e
vent, 'traduccion3')" ONMOUSEOUT="ocultarTraduccion
('traduccion3')">Chair</DIV>
```

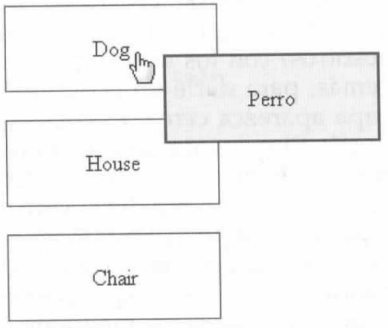


Figura 13.2. Traducción mostrada junto al cursor.

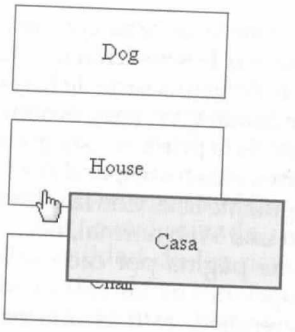


Figura 13.3. Otra traducción mostrada junto al cursor.

El mayor inconveniente de este ejemplo es que nos obliga a tener las traducciones escritas en la página (dentro de capas ocultas) desde un principio por lo que, si tenemos muchas palabras, resulta muy laborioso de mantener. Lo ideal sería que pudiéramos lanzar una consulta en una base de datos cuando alguien pulse sobre una de las palabras y mostrar después el resultado en la caja con la traducción. Esto sería totalmente posible con la técnica del apartado que sigue, convirtiéndose en un complemento perfecto.

13.2. AJAX

No, no hablamos de un producto de limpieza ni de un equipo de fútbol. Se trata de una técnica que nos permite aumentar la interactividad de nuestras páginas y de comunicarnos con un servidor desde el código JavaScript, entre otras ventajas. Su nombre proviene de las siglas de *Asynchronous JavaScript And XML*.

¿Qué es eso de comunicarnos con el servidor? A estas alturas todos conocemos los enlaces de una página Web y sabemos que, cuando pinchamos sobre ellos, el navegador cargará una nueva página borrando antes todo el contenido de la pantalla. En cada carga o petición, el servidor donde está alojada la Web nos manda el código HTML de la página solicitada, el cual debe interpretar y pintar nuestro navegador.

AJAX permite cargar una página o una serie de datos directamente sobre un elemento existente de nuestra página, de manera que no se nos estará redirigiendo, sino que una parte contenido visible se actualiza de manera independiente. Esto

es muy útil para evitar la recarga continua de una página, a la vez que aumentamos la sensación de velocidad de nuestra página. Imagine que tiene una serie de listas desplegables para elegir la población donde vive: país, provincia y municipio. Al seleccionar un valor de la primera lista (país), se preguntaría al servidor por las provincias correspondientes a la opción elegida y se rellenaría la siguiente lista. Con la última lista se procedería de igual forma. En una Web normal, necesitaríamos refrescar o repintar la misma página por cada selección, únicamente para mostrar las opciones de las listas y sin que cambie nada más. Como comprenderá, esto es una pérdida de eficiencia muy grande ya que nos estamos trayendo datos de más desde el servidor en cada recarga. Con AJAX, podemos actualizar cada una de las listas desplegables de forma independiente al resto de la página, con lo que sólo estamos obteniendo los datos justos y necesarios.

Figura 13.4. Petición de datos tradicional.

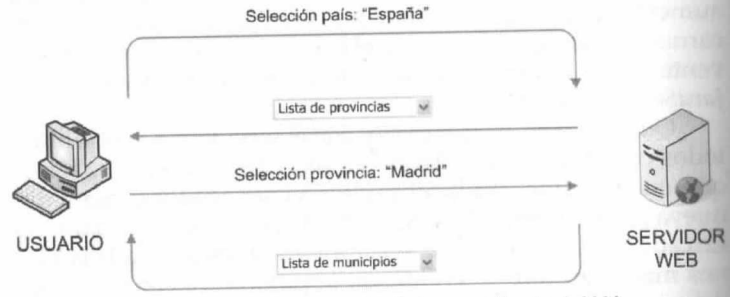


Figura 13.5. Petición de datos mediante AJAX.

Como puede apreciar en las figuras 13.4 y 13.5, la diferencia es bastante grande en cuanto a volumen de datos que se transfieren entre nuestro ordenador y el servidor Web.

13.3. jQuery

En realidad no es una técnica basada en JavaScript sino que se trata de una librería o biblioteca (de tantas), no obstante me apetece que la conozca porque le será de gran ayuda cuando quiera manejar rápidamente el DOM o crear vistosas animaciones y, además, consiguiendo un código compatible con la práctica totalidad de los navegadores, así que ya sólo por los dolores de cabeza que nos ahorrará merece la pena:). No sólo nos permite realizar acciones propias de DHTML sino que también incluye funciones para trabajar con AJAX.

Lo mejor de todo es que es gratuita y con licencia GNU (`http://www.gnu.org`) por lo que podrá utilizarla en su proyecto Web. No use "mulas" ni "torrentes", la tiene completamente disponible en su sitio Web `http://www.jquery.com` junto a una extensa documentación y varios ejemplos.

> **Nota:** *No me llevo comisión, lo prometo. Sólo soy un fan de esta librería.*

Existe una versión específica para dispositivos móviles llamada jQuery Mobile (`http://www.jquerymobile.com`) y mantiene una gran similitud con su "hermana mayor".

¿Por qué se la recomiendo? Veámoslo con un ejemplo. Imaginemos que tenemos una tabla de 3 columnas por 3 filas, por ejemplo, con los atributos `ID` y `CLASS` asignados.

```
<TABLE ID="tabla">
<TR>
   <TD ID="celda1" CLASS="importante">Texto 1</TD>
   <TD ID="celda2" CLASS="normal">Texto 2</TD>
   <TD ID="celda3" CLASS="normal">Texto 3</TD>
</TR>
<TR>
   <TD ID="celda4" CLASS="normal">Texto 4</TD>
   <TD ID="celda5" CLASS="importante">Texto 5</TD>
   <TD ID="celda6" CLASS="normal">Texto 6</TD>
</TR>
<TR>
   <TD ID="celda7" CLASS="importante">Texto 7</TD>
   <TD ID="celda8" CLASS="normal">Texto 8</TD>
   <TD ID="celda9" CLASS="importante">Texto 9</TD>
</TR>
</TABLE>
```

Ahora supongamos que queremos obtener una de las celdas para cambiarle el texto. Con DHTML tendremos que buscar la celda por su identificador mediante `getElementById` para después modificar el texto a través de la propiedad `innerHTML`.

```
<SCRIPT TYPE="text/javascript">
// Obtenemos la celda
var celda = document.getElementById("celda4");
// Modificamos el texto
celda.innerHTML = "Texto 4 modificado";
</SCRIPT>
```

También lo podemos simplificar en una única línea sin utilizar la variable.

```
<SCRIPT TYPE="text/javascript">
// Obtenemos la celda y modificamos el texto
document.getElementById("celda4").innerHTML = "Texto 4
modificado";
</SCRIPT>
```

Antes de mostrarle el equivalente usando jQuery voy a comentarle muy brevemente el manejo básico. Para buscar un elemento en el documento tendremos que usar el carácter de dólar `$` seguido del identificador que deseemos, precedido por una almohadilla (#). Esto nos permite tener acceso a las propiedades y métodos del elemento que coincida con ese identificador.

Esta librería añade varias propiedades y métodos nuevos a todos los elementos para aumentar la funcionalidad y hacer el código más sencillo. Por tanto, para modificar el texto vamos a usar uno de esos métodos, concretamente `html` que es muy fácil de utilizar:

- Si no lleva parámetros, nos devuelve el texto que tiene el elemento.
- Si le pasamos una cadena, sustituye el texto del elemento.

> **Advertencia:** *No olvide incluir la librería en el HEAD de su página, indicando la ruta hasta el fichero* `<SCRIPT TYPE="text/javascript" SRC="librerias/jquery.js"></SCRIPT>`

Vamos, ahora sí, a saber cómo se haría la misma acción usando la librería.


```
<SCRIPT TYPE="text/javascript">
// Obtenemos la celda y modificamos el texto
$("#celda4").html("Texto 4 modificado");
</SCRIPT>
```

Quizá esté un poco desilusionado porque no es un cambio tan radical como se lo había vendido. Bueno, esto es de lo más básico que se puede hacer:). Vamos a subir de nivel para que vea el potencial real de esta librería.

Pongamos que queremos poner en negrita los textos de todas las celdas que tengan aplicado el estilo `importante` en su atributo `CLASS`. Con DHTML, primero buscaremos todas las celdas (método `getElementsByTagName`) para después recorrerlas una a una en busca de las que contengan el estilo `importante` (propiedad `className`). A las que cumplan esa condición les modificaremos la propiedad CSS `fontWeight`, dentro de la propiedad `style`, que cambia la densidad o grosor del texto. En nuestro ejemplo, lo fijaremos como `bold` o negrita.

```
<SCRIPT TYPE="text/javascript">
// Obtenemos todas las celdas
var celdas = document.getElementsByTagName("TD");
for(var i = 0; i < celdas.length; i++) {
   // Comprobamos si contiene el estilo
   if (celdas[i].className == "importante") {
      // Ponemos su texto en negrita
      celdas[i].style.fontWeight = "bold";
   }
}
</SCRIPT>
```

Parece bastante sencillo. ¿Y cómo se podrá hacer con jQuery? Antes, una pequeña explicación más. Con jQuery también podemos obtener todas las celdas usando el dólar (`$`) y el nombre de la etiqueta de celda (`TD` en este caso)

```
var celdas = $("TD");
```

Pero podemos ir aún más allá y seleccionar de un plumazo todos los elementos que tengan un estilo determinado usando su nombre precedido por un punto (`.`).

```
var importantes = $(".importante");
```

Y por último, podemos combinar ambas condiciones: celdas que contengan el estilo `importante`. De este modo evitamos obtener otro tipo de elementos que también tengan aplicado el mismo estilo (una capa, una imagen, etc.)

```
var celdasImportantes = $("TD.importante");
```

¿Va viendo cómo se simplifica el código? Demos ahora el paso final. Para cambiar el texto a negrita tan sólo debemos fijar la propiedad `fontWeight` a `bold` como hicimos con DHTML, pero usando el método `css`, que se encarga de alterar cualquier propiedad CSS del elemento. Para deleite de los programadores, incluso podremos aplicar este cambio a todas las celdas a la vez sin usar un bucle (se hace internamente) y, por tanto, prescindiendo también de una variable que almacene el conjunto de celdas.

```
<SCRIPT TYPE="text/javascript">
// Ponemos en negrita todas las celdas con el estilo
$("TD.importante").css("font-weight", "bold");
</SCRIPT>
```

No sé qué efecto le habrá causado, pero le puedo asegurar que cuando lo descubrí quedé fascinado por su potencial y simplicidad. Por supuesto, hay mucho más detrás de jQuery y se pueden conseguir unos efectos muy llamativos dignos de Flash o PowerPoint.

Esta es sólo una de todas las librerías que existen para JavaScript, así que le animo a descubrir y utilizar la que más sea de su agrado. Una pequeña pista: `http://www.mootools.net` (muy similar a jQuery).

```
confirm("Esto se acabó. ¿Nos vemos en la próxima?");
```

A

Palabras reservadas

En todo lenguaje de programación existen una serie de palabras que no pueden ser utilizadas por el programador para nombrar sus variables, funciones, objetos o cualquier otro elemento, puesto que están reservadas por el propio lenguaje para nombrar sus sentencias u operadores. Si se permitiese emplear estas palabras, el lenguaje no sabría diferenciar el uso que queremos darle.

Tabla A.1. Lista de palabras reservadas de JavaScript (I).

Palabra			
break	else	instanceof	true
case	false	new	try
catch	finally	null	typeof
continue	for	return	var
default	function	switch	void
delete	if	this	while
do	in	throw	with

Aparte de la lista de la tabla A.1, JavaScript reservó de forma anticipada otra buena lista de palabras puesto que se preveía su uso en posteriores versiones.

Tabla A.2. Lista de palabras reservadas por JavaScript (II).

Palabra			
abstract	enum	int	short
boolean	export	interface	static

Palabra			
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

B

Precedencia de operadores

Todos los operadores de JavaScript pueden combinarse entre ellos y utilizarse cuantas veces se quiera por lo que se hace necesario establecer un orden de ejecución o precedencia entre ellos, el cual puede consultar en la tabla B.1.

Tabla B.1. Precedencia de operadores

Precedencia	*Operadores*
1	. []
2	() new
3	! ~ - (signo menos) + (signo más) ++ -- typeof void delete
4	* / %
5	+ (suma) - (resta)
6	<< >> >>>
7	< <= > >= in instanceof
8	== != === !==
9	&
10	^
11	|
12	&&
13	||
14	?:
15	= += -= *= /= %= &= ^= |= <<= >>= >>>=
16	,

C

Referencia rápida a los objetos de JavaScript

En este apéndice podrá encontrar una referencia a todos los objetos de JavaScript, con sus propiedades y métodos, junto con una breve descripción.

C.1. Boolean

Crea valores booleanos a partir de un valor. Su constructor es el siguiente:

```
<SCRIPT TYPE="text/javascript">
   // Objeto Boolean
   var nombre_variable = new Boolean(valor);
</SCRIPT>
```

El valor booleano que se devuelve depende de su parámetro:

- Si el parámetro es omitido o es 0, -0, `null`, cadena vacía (""), valor booleano `false`, valor `undefined` o valor `NaN`, entonces el objeto devuelve `false`.
- En cualquier otro caso, devuelve `true`.

Este objeto no tiene propiedades o métodos.

C.2. Number

Crea instancias con un valor numérico (entero o real). Su constructor es el siguiente:

```
<SCRIPT TYPE="text/javascript">
   // Objeto Number
   var nombre_variable = new Number(valor);
</SCRIPT>
```

En función del valor que se reciba se obtienen distintos resultados:

- Si no recibe un valor, devuelve un 0.
- Si es un número (expresado como tal o en forma de *string*), entonces inicializa el objeto con el número que recibe. La cadena vacía equivale a un 0.
- En caso de que reciba un valor no numérico, devuelve `NaN`.
- El valor `false` se interpreta como un 0 y `true` como un 1.

C.2.1. Propiedades

- `MAX_VALUE`: El mayor número que representable en JavaScript.
- `MIN_VALUE`: El menor número representable.
- `NaN` (*Not A Number*): Valor especial que indica que el valor no es un número.
- `POSITIVE_INFINITY`: Valor especial (`Infinity`) que indica el número es mayor que el máximo representable (`MAX_VALUE`).
- `NEGATIVE_INFINITY`: Valor especial (`-Infinity`) que indica el número es menor que el mínimo representable (`MIN_VALUE`).

C.2.2. Métodos

- `toExponential(decimales)`: Convierte a una expresión exponencial (e o E), con el número de posiciones decimales especificada en el parámetro `decimales`.
- `toFixed(decimales)`: Ajuste del número de posiciones decimales de un valor a las definidas por `decimales`, redondeando si es necesario.
- `toPrecision(digitos)`: Ajusta el número de dígitos de un valor a los especificados por `digitos`, redondeando si es necesario.

C.3. String

Este objeto de JavaScript crea instancias de cadenas. Su constructor debe escribirse así:


```
<SCRIPT TYPE="text/javascript">
   // Objeto String
   var nombre_variable = new String(valor);
</SCRIPT>
```

El valor que le pasemos al constructor será convertido a una cadena de texto en todos los casos.

C.3.1. Propiedades

- length: Número de caracteres que contiene la cadena.

C.3.2. Métodos

- `anchor(nombre)`, `link(direccion)`: Crean un ancla HTML y un enlace HTML respectivamente.
- `big()`, `blink()`, `bold()`, `fixed()`, `italics()`, `small()`, `strike()`, `sub()`, `sup()`: Encierran el valor de la cadena entre una etiqueta HTML de formato.
- `fontcolor(color)`, `fontsize(tamaño)`: Cambian el color y el tamaño de la cadena.
- `charAt(posicion)`, `charCodeAt(posicion)`: Devuelven el carácter y el código UNICODE del carácter que se encuentre en la posición indicada por su parámetro.
- `indexOf(texto, inicio)`, `lastIndexOf(texto, inicio)`: Devuelven la posición de la primera y última ocurrencia, respectivamente, de `texto` dentro la cadena. El parámetro `inicio` es opcional e indica la posición en la que se debe empezar a buscar.
- `concat(cadena1, cadena2, ..., cadenaN)`: Une el valor de la cadena con el de las cadenas pasadas como parámetros.
- `fromCharCode(codigo1, codigo2, ..., codigoN)`: Construye una cadena a partir de una lista de valores UNICODE. Es exclusivo del objeto `String`.
- `split(separador, limite)`: Devuelve un array con el subconjunto de cadenas resultantes de dividir la cadena tomando el parámetro `separador` como delimitador. El parámetro opcional `limite` indica el número de subconjuntos que se devuelven.
- `substring(inicio, fin)`, `substr(inicio, longitud)`, `slice(inicio, fin)`: Extrae y devuelve la porción de la cadena que se encuentra entre las

posiciones `inicio` y `fin` o una cantidad de caracteres (`longitud`) a partir de la posición `inicio`.
- `match(expresion)`, `replace(expresion, reemplazo)`, `search(expresion)`: Buscan una coincidencia de `expresion` dentro de la cadena o, en el caso de `replace`, sustituye la primera ocurrencia por el valor de `reemplazo`.
- `toLowerCase()`, `toUpperCase()`: Pasa la cadena a minúsculas o mayúsculas, respectivamente.

C.4. Array

Para crear un *array* existen varios constructores:

1. `new Array()`: crea un *array* vacío, sin huecos donde almacenar nuestros datos.
2. `new Array(longitud)`: en este caso el *array* se crea con tantos huecos como indique el parámetro `longitud`.
3. `new Array(valor1, valor2, ..., valorN)`: con este constructor crearemos un *array* con tantos huecos o posiciones como valores hayamos pasado como parámetros y, además, dichos huecos irán rellenos con el valor correspondiente.

Se puede almacenar cualquier tipo de datos en las posiciones del *array* y además pueden ser distintos en cada una.

C.4.1. Propiedades

- `length`: Indica la longitud del *array*.

C.4.2. Métodos

- `concat(array1, array2, ... , arrayN)`: Une el *array* actual con los que se pasen como parámetros, devolviendo el resultado en un único *array*.
- `join(separador)`: Traslada todos los valores de un *array* a una cadena, intercalando el carácter `separador` entre ellos.
- `pop()`: Elimina y devuelve el último valor de un *array*
- `push(valor1, valor2, ..., valorN)`: Añade uno o más valores al final del *array* y devuelve la longitud del mismo.

- `shift()`: Realiza la misma operación que `pop`, pero con el primer valor.
- `unshift(valor1, valor2, ..., valorN)`: Se comporta igual que `push`, pero insertando valores al principio del *array*.
- `slice(inicio, fin)`: Devuelve un nuevo *array* con los valores que estén entre las posiciones `inicio` y `fin-1`.
- `splice(inicio, longitud_borrado, valor1, valor2, ..., valorN)`: Añade y/o elimina valores de un array.
- `reverse()`: Invierte el orden de todos los valores del *array*.
- `sort(funcion)`: Ordena alfabéticamente los valores del *array*, pudiendo pasar una función para determinar como deben ordenarse los valores.

C.5. Date

El objeto `Date` almacena una fecha como los milisegundos que han pasado desde una fecha base (1 de enero de 1970 a las 00:00:00).

Se pueden crear instancias de la siguiente forma:

```
<SCRIPT TYPE="text/javascript">
   // Objeto Date
   var nombre_variable = new Date(fecha);
</SCRIPT>
```

Siendo el parámetro `fecha` cualquiera de estas expresiones:

- Vacío: Si no se pasa este parámetro, entonces se crea una instancia con la fecha y hora actuales.
- Una cantidad de milisegundos, que se sumarán a la fecha base.
- "Mes_en_letras día, año horas:minutos:segundos": El mes debe estar escrito en inglés, y si se omite las horas, minutos o segundos, tomarán valor 00 por defecto.
- "Año, mes, día": Estos valores deben ser numéricos enteros.
- "Año, mes, día, hora, minuto, segundos": Exactamente igual que el anterior pero ampliado para incluir la hora.

C.5.1. Propiedades

Este objeto no dispone de propiedades.

C.5.2. Métodos

Están agrupados según su funcionalidad.

Métodos de obtención

- `getTime()`: Con esto obtenemos la fecha expresada como el número de milisegundos que han pasado desde la fecha base.
- `getMilliseconds()`, `getSeconds()`, `getMinutes()`, `getHours()`: Devuelven el campo correspondiente a los milisegundos, segundos, minutos u hora, respectivamente.
- `getDate()`, `getDay()`, `getMonth()`: Nos permiten obtener el día del mes, el día de la semana y el mes.
- `getYear()`, `getFullYear()`: Devuelven el año con dos o cuatro cifras.
- `getUTCMilliseconds()`, `getUTCSeconds()`, `getUTCMinutes()`, `getUTCHours()`, `getUTCDate()`, `getUTCDay()`, `getUTCMonth()`, `getUTCFullYear()`: Funcionan exactamente igual que sus análogos pero devolviendo el campo de acuerdo al UTC.
- `getTimezoneOffset()`: Devuelve la cantidad de minutos de diferencia que existen entre la fecha de la instancia actual y el GMT.

Métodos de asignación

- `setTime(milisegundos)`: Crea una fecha añadiendo o restando los milisegundos indicados a la fecha base.
- `setMilliseconds(mseg)`, `setSeconds(seg, mseg)`, `setMinutes(min, seg, mseg)`, `setHours(hora, min, seg, mseg)`: Fija los campos correspondientes a los milisegundos (mseg), segundos (seg), minutos (min) y hora. Sólo es obligatorio el primer parámetro.
- `setDate(día)`, `setMonth(mes, día)`, `setYear(año)`, `setFullYear(año, mes, día)`: Fija los campos que corresponden al día del mes, el mes y el año. Sólo es obligatorio el primer parámetro.
- `setUTCMilliseconds(mseg)`, `setUTCSeconds(seg, mseg)`, `setUTCMinutes(min, seg, mseg)`, `setUTCHours(hora, min, seg, mseg)`, `setUTCDate(día)`, `setUTCMonth(mes, día)`, `setUTCFullYear(año, mes, día)`: Realizan lo mismo que sus equivalentes pero ajustadas al UTC.

Métodos de transformación

- `parse(fecha_cadena)`: Recibe una fecha expresada como una cadena y calcula el número de milisegundos que han pasado desde la fecha base. Es exclusivo del objeto `Date`.
- `UTC(año, mes, día, hora, minuto, segundo, milisegundo)`: Toma la fecha indicada por los parámetros y halla el número de milisegundos que han pasado desde la fecha base de acuerdo al UTC. Es exclusivo del objeto `Date`.
- `toDateString()`, `toLocaleDateString()`: Devuelven la fecha sin información sobre la hora en forma de texto.
- `toTimeString()`, `toLocaleTimeString()`: Similar a los anteriores pero devuelven únicamente la parte correspondiente a la hora.
- `toGMTString()`, `toUTCString()`, `toLocaleString()`: Expresan la fecha y la hora en base al GMT, UTC y configuración local respectivamente.

C.6. Math

Este objeto nos proporciona una serie de propiedades y métodos que nos permitirán realizar operaciones matemáticas complejas.

Este objeto no permite la creación de instancias por lo que todas sus propiedades y métodos deben utilizarse directamente a través del objeto `Math`.

C.6.1. Propiedades

- `E`: Representa la constante de Euler (2,718).
- `PI`: Nos da el número Pi (3,14159).
- `SQRT2`: Con esta obtendríamos la raíz cuadrada de 2 (1,414).
- `SQRT1_2`: La raíz cuadrada de ½ (0,707).
- `LN2`: Devuelve el logaritmo neperiano de 2 (0,693).

- `LN10`: El logaritmo neperiano de 10 (2,302).
- `LOG2E`: Devuelve el resultado del logaritmo de E en base 2 (1,442).
- `LOG10E`: Igual que el anterior, pero en base 10 (0,434).

C.6.2. Métodos

- `abs(numero)`: Devuelve el valor absoluto de `numero`.
- `sin(angulo)`, `cos(angulo)`, `tan(angulo)`: Funciones trigonométricas para calcular el seno, coseno y tangente de un ángulo expresado en radianes.
- `asin(numero)`, `acos(numero)`, `atan(numero)`: Realizan los cálculos trigonométricos arcoseno, arcocoseno y arcotangente.
- `atan2(posicionY, posicionX)`: Devuelve el ángulo que forma, respecto al eje X, el punto situado en las coordenadas indicadas.
- `exp(numero)`, `log(numero)`: Devuelven, respectivamente, el resultado de E elevado al parámetro `numero` (E^{numero}) y el logaritmo neperiano (base E) del parámetro.
- `ceil(numero)`: Devuelve el entero inmediatamente superior de `numero`, si éste tiene decimales, o directamente `numero` si no es así.
- `floor(numero)`: Igual que el anterior, pero devolviendo el entero inmediatamente inferior.
- `min(numero1, numero2)`, `max(numero1, numero2)`: Proporcionan el menor o mayor valor de sus dos parámetros.
- `pow(base, exponente)`: Calcula una potencia.
- `random()`: Este método devuelve un número aleatorio entre 0 y 1.
- `round(numero)`: Redondea el parámetro al entero más cercano.
- `sqrt(numero)`: Calcula la raíz cuadrada del parámetro `numero`.

C.7. RegExp

Este objeto permite trabajar con expresiones regulares mediante el siguiente constructor:


```
<SCRIPT TYPE="text/javascript">
   // Objeto RegExp
   var miPatron = new RegExp(patrón, modificadores);
</SCRIPT>
```

C.7.1. Propiedades

Las propiedades de este objeto se activan o desactivan mediante un valor booleano.

- `global`: Fija el modificador "g".
- `ignoreCase`: Fija el modificador "i".
- `multiline`: Activa el modificador "m".
- `lastIndex`: Representa la posición dentro del texto donde se empezará a buscar la próxima coincidencia.

Las siguientes propiedades sólo están accesibles desde el objeto `RegExp`, aunque hace referencia a valores de la instancia.

- `input`: Fija u obtiene el texto sobre el que se aplica el patrón.
- `lastMatch`: Devuelve la última cadena encontrada mediante el patrón.
- `lastParen`: Devuelve la última coincidencia que está encerrada entre paréntesis.
- `leftContext`, `rightContext`: Permite conocer los caracteres que están a la izquierda o derecha de la última coincidencia.

C.7.2. Métodos

- `compile(patron, modificadores)`: Permite fijar un nuevo patrón en una instancia de `RegExp` y opcionalmente también sus modificadores.
- `exec(texto)`: Aplica el patrón sobre el texto pasado como parámetro y devuelve la cadena encontrada.
- `test(texto)`: Aplica el patrón sobre el parámetro `texto` pero únicamente nos dirá si se encontraron coincidencias.

En el apéndice F tiene información sobre cómo escribir expresiones regulares.

D

Objetos del navegador (DOM)

La figura D.1 representa a modo resumen la estructura DOM de los navegadores Web.

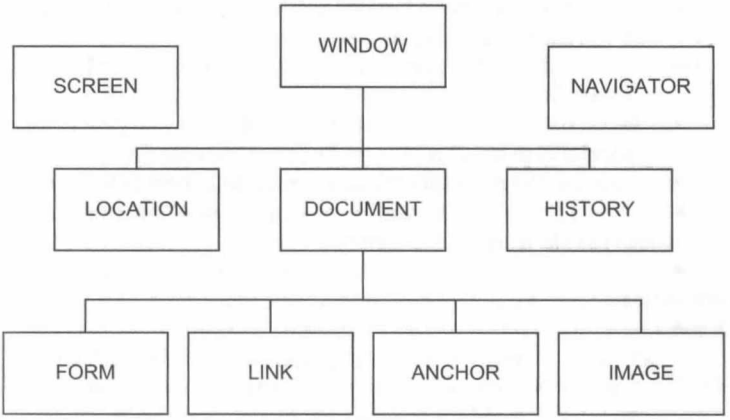


Figura D.1. Estructura DOM.

A continuación encontrará una referencia rápida a todos los objetos del DOM, con sus colecciones, propiedades y métodos junto a una breve descripción.

D.1. Objeto window

Es el objeto principal del DOM que representa la ventana del navegador donde se está visualizando la página.

D.1.1. Colecciones de objetos

El objeto `window` contiene a su vez otros objetos:

- `frames`: Se trata de un *array* con la colección de *frames* o marcos que contenga la página. Se accede a ellos a través de su posición o su nombre.

D.1.2. Propiedades

Este objeto dispone de un buen número de propiedades:

- `length`: Fija o devuelve el número de marcos de la ventana actual.
- `name`: Fija o devuelve el nombre de la ventana, que nos servirá para referirnos a ella en el código.
- `menubar`: Devuelve un objeto que representa la barra de menú del navegador.
- `toolbar`: Devuelve un objeto que representa la barra de herramientas del navegador.
- `statusbar`: Devuelve un objeto que representa la barra de estado.
- `defaultStatus`: Permite fijar u obtener el mensaje por defecto que se muestra en la barra de estado.
- `status`: Permite modificar el texto de la barra de estado.
- `scrollbars`: Devuelve el objeto que simboliza las barras de desplazamiento.
- `location`, `history`, `document`: Devuelven los objetos `location`, `history` y `document`.
- `outerHeight`, `outerWidth`: Establece o devuelve el tamaño en pixels del espacio de toda la ventana en vertical u horizontal, respectivamente, incluyendo las barras de desplazamiento, de herramientas, etc.
- `innerHeight`, `innerWidth`: Devuelve la altura o anchura, en pixels, del espacio donde se muestra la página incluyendo las barras de desplazamiento, si hubiera, pero excluyendo todas las demás (menú, herramientas, estado, etc.).
- `parent`: Se refiere a la ventana donde está situado el marco en el que estamos trabajando. En caso de no haber marcos, equivale a la propia ventana.
- `top`: Devuelve la ventana que se encuentra un nivel por encima del marco actual. Si no hay marcos, se devuelve la propia ventana.
- `self`: Representa a la ventana actual.

- `opener`: Hace referencia a la ventana que abrió la actual. Si la ventana no fue abierta por otra, obtendremos un valor `undefined`.
- `closed`: Indica si la ventana está cerrada o no.

D.1.3. Métodos

Entre los métodos del objeto `window` encontraremos:

- `alert(mensaje)`: Muestra un mensaje en un cuadro de diálogo con el botón **Aceptar**.
- `confirm(mensaje)`: Muestra un mensaje en un cuadro de diálogo con los botones **Aceptar** y **Cancelar**. Devuelve un booleano (`true` para **Aceptar** y `false` para **Cancelar**).
- `prompt(mensaje, valor_por_defecto)`: Similar a `confirm` pero incluyendo una caja de texto para recoger un valor escrito por el usuario, el cual es devuelto. Opcionalmente podemos indicar un valor que será mostrado por defecto en la caja de texto. Si el usuario hace clic sobre **Cancelar** obtendremos un valor `null`.
- `focus(),blur()`: Establece o retira el foco de un objeto.
- `moveBy(x, y)`: Mueve la ventana el número de píxeles indicados. La `x` indica el desplazamiento horizontal y la `y` el vertical.
- `moveTo(x, y)`: Mueve la ventana a una posición concreta.
- `resizeBy(x, y)`: Redimensiona la ventana tantos píxeles como indiquen los parámetros. La `x` se refiere a la anchura y la `y` a la altura.
- `resizeTo(x, y)`: Establece una anchura y altura concretas a la ventana.
- `scrollBy(x, y)`: Realiza un desplazamiento horizontal y/o vertical de tantos píxeles como marquen los parámetros.
- `scrollTo(x, y)`: Realiza un desplazamiento horizontal y/o vertical hasta una posición concreta.
- `open(URL, nombre, opciones)`: Crea una nueva ventana en la que se carga un URL. Las opciones se pasan como una cadena con pares propiedad=valor separados por comas, siendo `yes` o `1` valores positivos y `no` o `0` los negativos.
  - `toolbar`: Permite indicar si la ventana tendrá barra de herramientas o no.

- `menubar`: Muestra u oculta la barra de menús de la nueva ventana.
- `location`: Permite mostrar u ocultar la barra de direcciones en la nueva ventana.
- `directories`: Deja decidir si la ventana tendrá botones de dirección o no.
- `scrollbars`: Deja indicar si la nueva ventana tendrá barras de desplazamiento o no.
- `status`: Permite controlar la visibilidad de la barra de estado.
- `resizable`: Permite establecer si la nueva ventana podrá ser cambiada de tamaño (con el ratón) o no.
- `fullscreen`: Deja indicar si la ventana se verá a pantalla completa o no.
- `width`, `height`: Establece el ancho o alto que tendrá la ventana, en pixels.
- `top`, `left`: Indica la distancia, en pixels, a la que estará la ventana respecto al lado superior o izquierdo de la pantalla.

Este método devuelve un identificador para la ventana que hemos abierto o un valor `null` si la ventana no pudo abrirse.

- `close()`: Cierra la ventana actual.
- `print()`: Manda imprimir la página que estemos viendo.
- `setInterval(expresion, milisegundos)`: Evalúa una expresión continuamente después de que hayan pasado el número de milisegundos especificados. Devuelve un identificador para poder cancelar su ejecución.
- `setTimeout(expresion, milisegundos)`: Evalúa una expresión una única vez después de que hayan pasado el número de milisegundos indicados. Devuelve un identificador para poder cancelar su ejecución.
- `clearInterval(identificador)`, `clearTimeout(identificador)`: Cancelan respectivamente la ejecución de `setInterval` y `setTimeout` a través del identificador pasado como parámetro.

D.2. Objeto navigator

Con este objeto podremos acceder a información acerca del navegador que está utilizando el usuario para ver la página.

D.2.1. Propiedades

Gracias a esta lista de propiedades podemos obtener bastante información sobre el navegador del usuario:

- `appCodeName`: Devuelve el nombre del código del navegador.
- `appName`: Devuelve el nombre del navegador.
- `appMinorVersion`: Indica la versión de la última revisión del navegador.
- `appVersion`: Proporciona información acerca de la versión del navegador.
- `userAgent`: Devuelve la cabecera completa del navegador que se envía en cada petición HTTP (`appCodeName + appVersion`).
- `browserLanguage`: Devuelve el idioma del navegador.
- `systemLanguage`: Indica el idioma por defecto del sistema operativo del usuario.
- `userLanguage`: Indica el idioma preferido del usuario para el sistema operativo que está utilizando.
- `language`: Devuelve el idioma del navegador.
- `cpuClass`: Ofrece información acerca del tipo de procesador que tiene el usuario.
- `platform`: Indica la plataforma del sistema operativo.
- `cookieEnabled`: Permite saber si el navegador tiene activadas las *cookies* o no.

D.2.2. Métodos

Como no podemos modificar ninguna de las características del navegador, sólo tenemos un método a nuestro alcance:

- `javaEnabled()`: Indica si el navegador tiene activado el soporte para Java (no confunda con JavaScript).

D.3. Objeto screen

Mediante este objeto podremos obtener información acerca de la configuración de pantalla del usuario que está viendo nuestra página. Sólo dispone de propiedades.

D.3.1. Propiedades

Conocer algunos parámetros de la configuración del usuario será útil para realizar algún ajuste en nuestro código.

- `width`, `height`: Indican los pixels de ancho y alto que tiene la resolución actual.
- `availWidth`, `availHeight`: Devuelven la anchura y altura que queda libre en el área de trabajo del escritorio, es decir, resta al ancho o alto total el espacio que ocupan las barras de herramientas que tengamos en nuestro sistema operativo.
- `colorDepth`: Permite conocer la profundidad de color, en bits, de la pantalla del usuario.

D.4. Objeto history

Este objeto almacena una lista de los sitios por los que hemos estado navegando.

D.4.1. Propiedades

Tan sólo tiene una propiedad de interés:

- `length`: Devuelve la cantidad de sitios que hay dentro del historial.

D.4.2. Métodos

Los métodos de `history` nos ayudarán a desplazarnos por las páginas del historial.

- `back()`, `forward()`: Carga la página inmediatamente anterior o posterior a la actual.
- `go(posicion)`: Carga una página específica que está en la posición indicada dentro del historial. Un valor del parámetro `posicion` negativo representa páginas anteriores y uno positivo las posteriores.

D.5. Objeto location

Este objeto contiene el URL actual.

D.5.1. Propiedades

Con estas propiedades podremos separar el URL en varias partes para usar la que más nos interese.

- `href`: Devuelve el URL completo. También podemos reemplazarlo para dirigirnos a otra página.
- `hostname`: Devuelve únicamente la parte del URL que hace referencia al nombre o dirección IP del servidor donde está alojada la página.
- `pathname`: Contiene la ruta que se sigue, dentro del servidor Web, hasta alcanzar la página actual.
- `hash`: Devuelve la parte del URL que hay después de la almohadilla (`#`), representando un ancla (*anchor*).
- `search`: Devuelve la parte del URL que hay después del signo de interrogación (`?`), conocido como consulta o *query string*.
- `port`: Indica el puerto por el que hemos accedido al servidor.
- `host`: Devuelve el nombre del servidor (`hostname`) y el número del puerto (`port`).
- `protocol`: Indica el protocolo que se ha usado para acceder al servidor (normalmente HTTP).

D.5.2. Métodos

Su función principal es la de interactuar con un URL.

- `assign(URL)`: Carga un nuevo URL, creando una nueva entrada en el objeto `history`.
- `replace(URL)`: Reemplaza el URL actual por otro, pero reemplazando la entrada actual en `history`.
- `reload()`: Recarga la página actual.

D.6. Objeto document

Todos los elementos de una página (texto, imágenes, enlaces, formularios...) quedan representados y accesibles dentro de este objeto.

D.6.1. Colecciones de objetos

Este objeto contiene a su vez otros objetos.

- `anchors`: Es un *array* con todas las anclas (*anchors*) del documento (se mantiene por temas de compatibilidad).
- `forms`: Contiene un array con todas las referencias a los formularios de la página.
- `images`: Tiene ecopiladas todas las imágenes representadas con la etiqueta `<IMG>`.
- `links`: Contiene todos los enlaces de la página, dentro de un *array*.

D.6.2. Propiedades

Las propiedades de este objeto nos proporcionarán alguna información de interés:

- `cookie`: Devuelve, en forma de cadena, los valores de las *cookies* que tenga el documento.
- `domain`: Indica el nombre del servidor donde está alojado el documento.
- `lastModified`: Indica la fecha de la última modificación del documento.
- `referrer`: Indica el URL del documento que llamó al actual, a través de un enlace.
- `title`: Devuelve título del documento que aparece en la parte superior izquierda de la ventana del navegador.
- `URL`: Devuelve el URL del documento actual.

D.6.3. Métodos

Estos métodos nos permiten modificar ligeramente el contenido de la página.

- `write(texto)`,`writeln(texto)`: Escribe texto HTML en el documento. El método `writeln` añade además un salto de línea al final.
- `open()`: Permite escribir sobre el documento una vez que el documento ha sido cargado.
- `close()`: Finaliza la escritura sobre el documento.
- `getElementById(identificador)`: Devuelve el primer elemento del documento cuyo atributo `ID` coincida con el parámetro `identificador`.
- `getElementsByName(nombre)`: Devuelve un *array* de elementos cuyo atributo `NAME` coincida con el indicado.
- `getElementsByTagName(etiqueta)`: Devuelve un *array* de elementos cuya etiqueta sea la misma que la especificada por el parámetro.

D.7. Objeto anchor

Este objeto engloba todos los enlaces internos al documento, también llamados anclas o *anchor*.

D.7.1. Propiedades

Las propiedades de este objeto representan básicamente todos los atributos HTML que puede tener.

- `id`: Identicador del ancla.
- `name`: Nombre del ancla.
- `target`: Ventana o marco donde se cargará el ancla.
- `text`: Permite saber el texto que contiene el ancla (sólo lectura).
- `innerHTML`: Permite obtener o modificar el texto que contiene el ancla.

D.7.2. Métodos

Sólo contamos con dos métodos para este objeto.

- `focus()`, `blur()`: Hacen que reciba o pierda el foco, respectivamente.

D.8. Objeto link

Este objeto permite tener acceso a todos los enlaces externos del documento.

D.8.1. Propiedades

La lista de propiedades que tiene este tipo de objeto es muy similar a la que vimos para `location`.

- `id`: Identicador del enlace.
- `href`: Devuelve el URL completo del enlace.
- `target`: Ventana o marco donde se cargará el enlace.
- `hostname`: Devuelve la parte del URL que hace referencia al nombre o dirección IP del servidor donde está alojada la página destino.
- `pathname`: Contiene la ruta que se sigue hasta alcanzar la página destino.

- `hash`: Devuelve la parte del URL que hay después de la almohadilla (#), llamada ancla (*anchor*).
- `search`: Devuelve la parte del URL que hay después del signo de interrogación (?), conocida como consulta o *query string*.
- `port`: Indica el puerto por el que vamos a acceder al servidor destino.
- `host`: Devuelve el nombre del servidor (`hostname`) y el número del puerto (`port`).
- `protocol`: Indica el protocolo que vamos a usar para acceder al servidor.
- `text`: Permite saber el texto del enlace (sólo lectura).
- `innerHTML`: Permite obtener y modificar el texto del enlace

D.8.2. Métodos

Este objeto sólo dispone de dos métodos.

- `focus()`, `blur()`: Hacen que reciba o pierda el foco, respectivamente.

D.9. Objeto image

Mediante este objeto seremos capaces de acceder y manipular ciertas características de una imagen contenida en la página. Este objeto no cuenta con ningún método.

D.9.1. Propiedades

- `id`: Se corresponde con el identificador de la imagen.
- `name`: Proporciona acceso al nombre del objeto.
- `src`: Permite manipular el URL de la imagen que se muestra.
- `width`, `height`: Permiten conocer y cambiar la anchura y altura de la imagen.
- `alt`: Corresponde con el texto alternativo que se muestra al usuario cuando la imagen no puede ser cargada.
- `title`: Representa el texto que se muestra sobre la imagen al poner el cursor sobre ella.

E

Formularios en JavaScript

Este apéndice pretende recopilar todo lo referente al manejo de formularios a través de código JavaScript para que tenga una referencia rápida a todos los objetos, con sus colecciones de objetos, propiedades y métodos, junto con una breve descripción.

E.1. Formulario

El objeto `document` nos otorga un acceso directo a todos los formularios de la página, a través de su colección `forms`, aunque también se puede acceder a ellos mediante su nombre.

```
// Acceso a un formulario
var miForm = document.forms[0];
var miForm = document.formRegistro;
```

E.1.1. Colecciones de objetos

- `elements`: *Array* con todos los campos que componen el formulario.

E.1.2. Propiedades

- `id`: Representa el identificador del formulario (atributo `ID`).
- `name`, `action`, `method`, `target`, `enctype`: Representan cada uno de los atributos HTML correspondientes.
- `length`: El número de campos que tiene el formulario. Es lo mismo que `elements.length`.

E.1.3. Métodos

- `submit()`: Envía el formulario con toda la información que haya introducido el usuario en él.
- `reset()`: Deja el formulario en blanco, tal y como estaba al cargar la página.

E.2. Campos de entrada de datos

Nos permiten obtener datos directamente del usuario. Están representados por las etiquetas `<INPUT>` (`text`, `password`, `file` y `hidden`) y `<TEXTAREA>`.

E.2.1. Propiedades

Las siguientes propiedades son comunes para todos los elementos.

- `id`: Representa el identificador del formulario (atributo `ID`).
- `disabled`, `name`, `readOnly`, `type`, `value`: Devuelven el valor del atributo HTML correspondiente.
- `form`: Nos indica el objeto de formulario donde está contenido el campo actual.

Las cajas de texto, definidas con `<INPUT>`, tienen además estas propiedades:

- `maxLength`: Nos permite obtener y modificar la longitud máxima del campo.
- `size`: Devuelve el número de caracteres visibles en el campo.

Como no podía ser de otra manera, los `<TEXTAREA>` también tienen algunas propiedades más:

- `cols`, `rows`: Nos dejan obtener o modificar el ancho o el alto del cuadro de texto.

E.2.2. Métodos

- `focus()`, `blur()`: Establece o retira el foco de la caja.
- `select()`: Selecciona todo el texto que esté escrito dentro de la caja de texto.

E.3. Campos de selección de datos

También nos permiten recoger información del usuario dándole a escoger un valor dentro de una serie de opciones. Aquí entran las etiquetas `<INPUT>` (`radio` y `checkbox`), `<SELECT>` y `<OPTION>`.

E.3.1. Colecciones de objetos

Sólo aplicable a los elementos `<SELECT>`. Se trata de una colección que representa los elementos `<OPTION>`.

- `options`: Contiene un *array* todas las opciones que muestra la lista desplegable.

E.3.2. Propiedades

Las propiedades listadas a continuación son comunes para todos los elementos.

- `id`: Equivale al identificador del campo (atributo `ID`).
- `name`, `type`: Representan el nombre y el tipo del elemento (atributos `NAME` y `TYPE`), excepto para los `<OPTION>`, que no disponen de estos atributos.
- `value`: Nos devuelve el valor que tenga asociado el campo en su atributo `VALUE`. En el caso de los `<SELECT>` nos devolverá el valor de la opción que esté seleccionada.
- `disabled`: Nos permite habilitar o deshabilitar el campo.
- `form`: Nos devuelve el objeto de formulario en el que se encuentra el campo.

Adicionalmente, los dos tipos de elementos `<INPUT>` que pueden existir en un formulario (`radio` y `checkbox`) tienen esta propiedad:

- `checked`: Permite saber y establecer cuándo está seleccionado el campo.

Las listas desplegables (`<SELECT>`) tienen además estas propiedades:

- `multiple`: Nos permite conocer y establecer cuando la lista admite selección múltiple.

- `size`: Nos devuelve el número de opciones visibles al mismo tiempo en la lista desplegable.
- `length`: Nos indica el número de opciones de las que dispone. Equivale a la longitud de su colección de datos, es decir, `options.length`.
- `selectedIndex`: Nos permite obtener o establecer la opción que está seleccionada. El valor 0 representa la primera opción. Si esta propiedad toma el valor especial -1 significa que no hay ninguna opción seleccionada.

Por último, las propiedades de los elementos `<OPTION>`:

- `selected`: Nos permite saber o fijar cuándo la opción estará seleccionada.
- `text`: Obtiene el texto que se encuentra junto a la etiqueta `<OPTION>` y también nos deja modificarlo.
- `index`: Devuelve la posición que ocupa la opción dentro de la lista desplegable.

E.3.3. Métodos

Todos los campos de selección tiene estos dos métodos definidos:

- `focus()`,`blur()`: Nos dejan establecer o retirar el foco del campo.

Además los `<INPUT>` disponen de otro más:

- `click()`: Simula un clic de ratón sobre el campo, de forma que lo dejará seleccionado o deseleccionado dependiendo de su estado anterior.

E.4. Botones

Con ellos podremos ejecutar una serie de operaciones con los datos introducidos en el formulario. Todos los botones se definen con la etiqueta `<INPUT>` (`button`, `submit` y `reset`).

E.4.1. Propiedades

Todos estos elementos tienen las mismas propiedades.

- `id`, `disabled`, `name`, `type`, `value`: Nos permiten manipular los atributos que representan con el mismo nombre.

- `form`: Nos indica el formulario en el que se encuentran los botones.

E.4.2. Métodos

- `focus()`,`blur()`: Sirven para colocar o quitar el foco del botón.
- `click()`: Simula un clic de ratón sobre el botón como si lo hubiera hecho el usuario.

E.5. Resumen de tipos de campos

La tabla E.1 le permitirá tener a mano los valores que utiliza JavaScript para identificar todos los posibles valores de la propiedad `type` de los campos de un formulario.

Tabla E.1. Valores de la propiedad type.

Campo	*Valores type*
Entrada de datos	text, password, file, hidden, textarea
Selección de datos	radio, checkbox, select-one (sin atributo MULTIPLE), select-multiple (con atributo MULTIPLE)
Botones	button, submit, reset

F

Caracteres y modificadores de las expresiones regulares

Para hacer más útiles las expresiones regulares, se utilizan una serie de estructuras y caracteres especiales para crear patrones más complejos y adaptables a nuestros propósitos. Además pueden combinarse entre ellos las veces que sean necesarias.

F.1. Caracteres de repetición

Permiten indicar que se busquen coincidencias en nuestro patrón teniendo en cuenta que algunas partes de la cadena pueden repetirse un número determinado de veces.

- Asterisco (`*`): Indica que el carácter que le precede puede aparecer cero o más veces.
- Más (`+`): Indica que el carácter que le precede puede repetirse una o más veces.
- Interrogación (`?`): Indica que el carácter que le precede puede aparecer ninguna o una vez.
- `{n}`: Siendo `n` un entero positivo, indica que el carácter que le precede se debe repetir exactamente el número de veces especificado con `n`.
- `{n,}`: Similar al anterior, pero el carácter debe aparecer al menos `n` veces.
- `{n, m}`: Siendo `n` y `m` enteros positivos, indica que el carácter que preceda a esta estructura debe repetirse un número de veces entre las definidas por `n` y `m`.

F.2. Caracteres especiales

A veces es necesario incluir en un patrón ciertos caracteres, como el tabulador o el salto de línea, que deben escribirse de una forma concreta para que JavaScript pueda reconocerlos.

- Punto (`.`): Coincidirá con cualquier carácter simple excepto con el de salto de línea.
- `\n`: Nueva línea (o salto de línea).
- `\r`: El retorno de carro.
- `\t`: Carácter de tabulado.
- `\v`: El de tabulado vertical.
- `\f`: Avance de página.
- `\uxxxx`: Carácter UNICODE que tiene como código cuatro dígitos hexadecimales representados por `xxxx`.
- `\b`: Separador de palabra, como puede ser el espacio o el carácter de nueva línea.
- `\B`: Carácter que no sea separador de palabra.
- `\cX`: Carácter de control en una cadena siendo `X` dicho carácter (combinación de teclas **Control-X**).
- `\d`: Dígito entre cero y nueve.
- `\D`: Carácter que no sea un dígito.
- `\s`: Un carácter de separación (espacio, tabulado, avance de página o nueva línea).
- `\S`: Un carácter que no sea de separación.
- `\w`: Cualquier carácter alfanumérico (letras de la "a" a la "z", minúsculas o mayúsculas y números del cero al nueve) o el subrayado.
- `\W`: Cualquier carácter no alfanumérico ni subrayado.
- `\0` (cero): Carácter nulo o de fin de cadena. No puede tener ningún dígito a continuación.

F.3. Agrupación de valores

Para encontrar una coincidencia en palabras que son muy similares diferenciándose en una pequeña parte, podemos usar estas estructuras para fusionar todo bajo un mismo patrón.

- [`xxx`]: Coincide con uno de los caracteres que están entre los corchetes. También es posible especificar un rango de caracteres usando un guión donde los valores deben ser contiguos.

- [`^xxx`]: Coincide con cualquier carácter salvo los que estén indicados en el conjunto.
- Barra vertical (`x | y`): Coincide con `x` o `y`, pero no con los dos.
- [`\b`]: Coincide con el carácter de retroceso o *backspace*. No confundir con el carácter especial `\b` (separador de palabra).

F.4. Caracteres de posición

Los caracteres de posición nos permiten especificar en qué parte de la línea debe existir la coincidencia dentro de nuestro patrón.

- Circunflejo (`^`): Hace que coincida desde el inicio de la línea.
- Dólar (`$`): Hace que coincida con la parte final de la línea.

F.5. Modificadores

Permiten definir algunas características a tener en cuenta a la hora de buscar coincidencias.

Podemos aplicar a un patrón tantos modificadores como queramos y deben escribirse a continuación de la última barra (`/`) que delimita el patrón.

- g: Fuerza que se sigan buscando coincidencias después de encontrar la primera.
- i: Elimina la distinción entre mayúsculas y minúsculas.
- m: Permite usar varios caracteres de posición (`^` y `$`) en el patrón.
- s: Incluye el salto de línea en el comodín punto (`.`).
- x: Fuerza que los espacios sean ignorados.

F.6. Expresiones regulares útiles

Los patrones que nos presenta la siguiente tabla F.1 se suelen utilizar con frecuencia.

Tabla F.1. Expresiones regulares de uso frecuente.

Expresión regular	*Descripción*
/\d{9}/	Número de teléfono
/\d{8}[a-zA-Z]/	Número de DNI
/\d{2}-\d{2}-\d{4}/	Fecha (día-mes-año o mes-día-año)
/[0-3]\d-[01]\d-\d{4}/	Fecha (día-mes-año)
/([0][1-9]|1[0-2]):[0-5]\d:[0-5]\d/	Hora (hora:minutos:segundos en formato de 12 horas)
/([01]\d|2[0-3]):[0-5]\d:[0-5]\d/	Hora (hora:minutos:segundos en formato de 24 horas)
/\w+@\w+\.\w{2,3}/	Dirección de correo electrónico

Índice alfabético

Símbolos

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

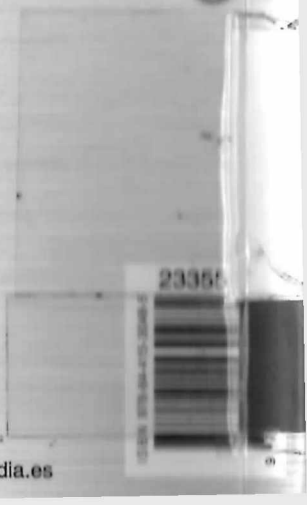
X

Y

Las **Guías Prácticas**
de Anaya Multimedia son los libros
más apreciados por los usuarios
que se inician en el estudio de un tema
informático. Ofrecen una visión rápida
y rigurosa de todo lo relacionado
con ordenadores personales, lenguajes
de programación, Internet y nuevas
tecnologías en general.

El objetivo es que todos los interesados
encuentren aquí una forma práctica
y solvente de resolver sus dudas.
El lenguaje es claro e incluye ejemplos
y ejercicios. Además, las capturas
de pantalla e ilustraciones permiten
fijar las ideas de modo eficaz,
actualizar conocimientos y profundizar
en el aprendizaje.

En definitiva, aquí encontrará todo
lo que necesita saber. Lo esencial
de la informática al alcance de todos.



http://www.AnayaMultimedia.es