Arkaitz Garro

JavaScript

Esta página se ha dejado vacía a propósito

JavaScript

Publication date: 14/06/2013

This book was published with *easybook v5.0-DEV*, a free and open-source book publishing application developed by Javier Eguiluz (http://javiereguiluz.com) using several Symfony components (http://components.symfony.com) .

Esta página se ha dejado vacía a propósito

Esta página se ha dejado vacía a propósito

Capítulo 1

JavaScript es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos (http://es.wikipedia.org/wiki/Programaci%C3%B3n_basada_en_prototipos) , imperativo (http://es.wikipedia.org/wiki/Programaci%C3%B3n_imperativa) , débilmente tipado y dinámico.

Se utiliza principalmente en el lado del cliente, implementado como parte de un navegador web permitiendo crear interacción con el usuario y páginas web dinámicas, aunque actualmente es posible ejecutar JavaScript en el propio servidor (NodeJS (http://nodejs.org/)). Su uso en aplicaciones externas a la web, por ejemplo en documentos PDF o aplicaciones de escritorio (mayoritariamente widgets) es también significativo.

JavaScript se diseñó con una sintaxis similar al lenguaje de programación C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo Java y JavaScript no están relacionados y tienen semánticas y propósitos diferentes.

A principios de los años 90, la mayoría de usuarios que se conectaban a Internet lo hacían con módems a una velocidad máxima de 28.8 kbps. En esa época, empezaban a desarrollarse las primeras aplicaciones web y por tanto, las páginas web comenzaban a incluir formularios complejos.

Con unas aplicaciones web cada vez más complejas y una velocidad de navegación tan lenta, surgió la necesidad de un lenguaje de programación que se ejecutara en el navegador del usuario. De esta forma, si el usuario no rellenaba correctamente un formulario, no se le hacía esperar mucho tiempo hasta que el servidor volviera a mostrar el formulario indicando los errores existentes.

Brendan Eich, un programador que trabajaba en Netscape, pensó que podría solucionar este problema adaptando otras tecnologías existentes (como ScriptEase) al navegador Netscape Navigator 2.0, que iba a lanzarse en 1995. Inicialmente, Eich denominó a su lenguaje LiveScript.

Posteriormente, Netscape firmó una alianza con Sun Microsystems para el desarrollo del nuevo lenguaje de programación. Además, justo antes del lanzamiento Netscape decidió cambiar el nombre por el de JavaScript. La razón del cambio de nombre fue exclusivamente por marketing, ya que Java era la palabra de moda en el mundo informático y de Internet de la época.

La primera versión de JavaScript fue un completo éxito y Netscape Navigator 3.0 ya incorporaba la siguiente versión del lenguaje, la versión 1.1. Al mismo tiempo, Microsoft lanzó JScript con su navegador Internet Explorer 3. JScript era una copia de JavaScript al que le cambiaron el nombre para evitar problemas legales.

Para evitar una guerra de tecnologías, Netscape decidió que lo mejor sería estandarizar el lenguaje JavaScript. De esta forma, en 1997 se envió la especificación JavaScript 1.1 al organismo ECMA European Computer Manufacturers Association).

ECMA creó el comité TC39 con el objetivo de "estandarizar de un lenguaje de script multiplataforma e independiente de cualquier empresa". El primer estándar que creó el comité TC39 se denominó **ECMA-262**, en el que se definió por primera vez el lenguaje ECMAScript.

Por este motivo, algunos programadores prefieren la denominación ECMAScript para referirse al lenguaje JavaScript. De hecho, JavaScript no es más que la implementación que realizó la empresa Netscape del estándar ECMAScript.

La organización internacional para la estandarización (ISO) adoptó el estándar ECMA-262 a través de su comisión IEC, dando lugar al estándar ISO/IEC-16262.

ECMA ha publicado varios estándares relacionados con ECMAScript. En Junio de 1997 se publicó la primera edición del estándar ECMA-262. Un año después, en Junio de 1998 se realizaron pequeñas modificaciones para adaptarlo al estandar ISO/IEC-16262 y se creó la segunda edición.

La quinte edición del estándar ECMA-262 (publicada en junio de 2011) es la versión que utilizan los navegadores actuales y se puede consultar gratuitamente en http://www.ecma-international.org/publications/standards/Ecma-262.htm

La integración de JavaScript y XHTML es muy flexible, ya que existen al menos tres formas para incluir código JavaScript en las páginas web.

El código JavaScript se encierra entre etiquetas `<script>` y se incluye en cualquier parte del documento. Aunque es correcto incluir cualquier bloque de código en cualquier zona de la página, se recomienda definir el código JavaScript dentro de la cabecera del documento (dentro de la etiqueta `<head>`):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Ejemplo de código JavaScript en el propio documento</title>
    <script type="text/javascript">
      alert("Un mensaje de prueba");
    </script>
</head>
<body>
    <p>Un párrafo de texto.</p>
</body>
</html>
```

Para que la página XHTML resultante sea válida, es necesario añadir el atributo `type` a la etiqueta `<script>`. Los valores que se incluyen en el atributo `type` están estandarizados y para el caso de JavaScript, el valor correcto es `text/javascript`.

Este método se emplea cuando se define un bloque pequeño de código o cuando se quieren incluir instrucciones específicas en un determinado documento HTML que completen las instrucciones y funciones que se incluyen por defecto en todos los documentos del sitio web.

El principal inconveniente es que si se quiere hacer una modificación en el bloque de código, es necesario modificar todas las páginas que incluyen ese mismo bloque de código JavaScript.

Las instrucciones JavaScript se pueden incluir en un archivo externo de tipo JavaScript que los documentos XHTML enlazan mediante la etiqueta `<script>`. Se pueden crear todos los archivos JavaScript que sean necesarios y cada documento XHTML puede enlazar tantos archivos JavaScript como necesite.

Ejemplo:

Archivo `codigo.js`

```
alert("Un mensaje de prueba");
```

Documento XHTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Ejemplo de código JavaScript en el propio documento</title>
    <script type="text/javascript" src="/js/codigo.js"></script>
</head>
<body>
    <p>Un párrafo de texto.</p>
</body>
</html>
```

Además del atributo `type`, este método requiere definir el atributo `src`, que es el que indica la URL correspondiente al archivo JavaScript que se quiere enlazar. Cada etiqueta `<script>` solamente puede enlazar un único archivo, pero en una misma página se pueden incluir tantas etiquetas `<script>` como sean necesarias.

Los archivos de tipo JavaScript son documentos normales de texto con la extensión `.js`, que se pueden crear con cualquier editor de texto como Notepad, Wordpad, EmEditor, UltraEdit, Vi, etc.

La principal ventaja de enlazar un archivo JavaScript externo es que se simplifica el código XHTML de la página, que se puede reutilizar el mismo código JavaScript en todas las páginas del sitio web y que cualquier modificación realizada en el archivo JavaScript se ve reflejada inmediatamente en todas las páginas XHTML que lo enlazan.

Este último método es el menos utilizado, ya que consiste en incluir instrucciones JavaScript dentro del código XHTML de la página:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Ejemplo de código JavaScript en el propio documento</title>
</head>
<body>
    <p onclick="alert('Un mensaje de prueba')">Un párrafo de texto.</p>
</body>
</html>
```

El mayor inconveniente de este método es que ensucia innecesariamente el código XHTML de la página y complica el mantenimiento del código JavaScript. En general, este método sólo se utiliza para definir algunos eventos y en algunos otros casos especiales, como se verá más adelante.

Algunos navegadores no disponen de soporte completo de JavaScript, otros navegadores permiten bloquearlo parcialmente e incluso algunos

usuarios bloquean completamente el uso de JavaScript porque creen que así navegan de forma más segura.

En estos casos, es habitual que si la página web requiere JavaScript para su correcto funcionamiento, se incluya un mensaje de aviso al usuario indicándole que debería activar JavaScript para disfrutar completamente de la página. El siguiente ejemplo muestra una página web basada en JavaScript cuando se accede con JavaScript activado y cuando se accede con JavaScript completamente desactivado.

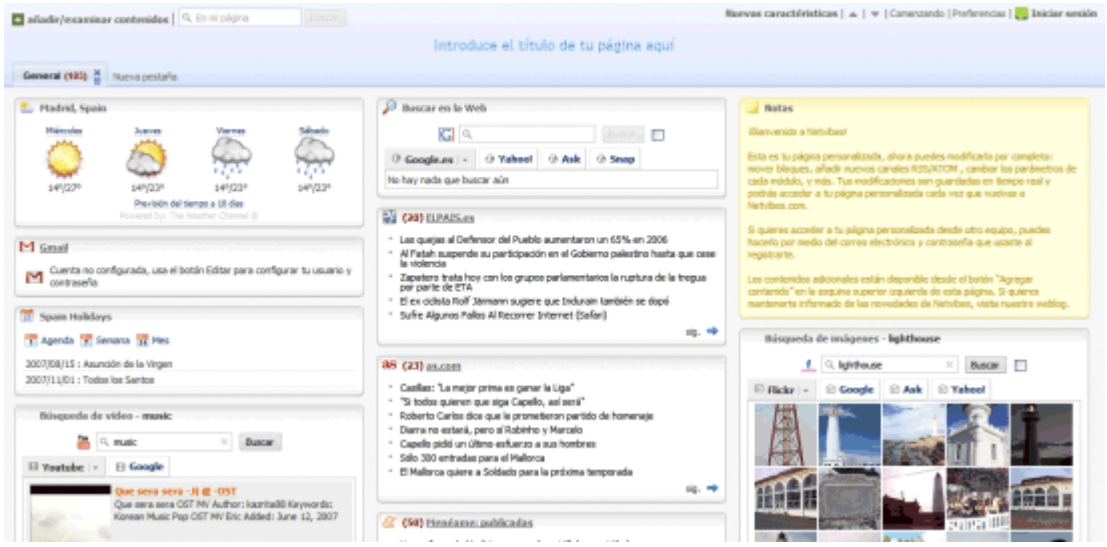


Figura 1.1 Imagen de www.netvibes.com con JavaScript activado

Figura 1.2 Imagen de www.netvibes.com con JavaScript desactivado

El lenguaje HTML define la etiqueta `<noscript>` para mostrar un mensaje al usuario cuando su navegador no puede ejecutar JavaScript. El siguiente código muestra un ejemplo del uso de la etiqueta `<noscript>`:

```
<head> ... </head>
<body>
    <noscript>
      <p>Bienvenido a Mi Sitio</p>
      <p>La página que estás viendo requiere para su funcionamiento el
      uso de JavaScript. Si lo has deshabilitado intencionadamente,
```

```
      por favor vuelve a activarlo.</p>
    </noscript>
</body>
```

La etiqueta `<noscript>` se debe incluir en el interior de la etiqueta `<body>` (normalmente se incluye al principio de `<body>`). El mensaje que muestra `<noscript>` puede incluir cualquier elemento o etiqueta XHTML.

- **Script**: cada uno de los programas, aplicaciones o trozos de código creados con el lenguaje de programación JavaScript. Unas pocas líneas de código forman un script y un archivo de miles de líneas de JavaScript también se considera un script.
- **Sentencia**: cada una de las instrucciones que forman un script.
- **Palabras reservadas**: son las palabras (en inglés) que se utilizan para construir las sentencias de JavaScript y que por tanto no pueden ser utilizadas libremente. Las palabras actualmente reservadas por JavaScript son: `break`, `case`, `catch`, `continue`, `default`, `delete`, `do`, `else`, `finally`, `for`, `function`, `if`, `in`, `instanceof`, `new`, `return`, `switch`, `this`, `throw`, `try`, `typeof`, `var`, `void`, `while`, `with`.

La sintaxis de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

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

- **No se tienen en cuenta los espacios en blanco y las nuevas líneas**: como sucede con XHTML, el intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.)
- **Se distinguen las mayúsculas y minúsculas**: al igual que sucede con la sintaxis de las etiquetas y elementos XHTML. Sin embargo, si en una página XHTML se utilizan indistintamente mayúsculas y

minúsculas, la página se visualiza correctamente, siendo el único problema la no validación de la página. En cambio, si en JavaScript se intercambian mayúsculas y minúsculas el script no funciona.

- **No se define el tipo de las variables**: al crear una variable, no es necesario indicar el tipo de dato que almacenará. De esta forma, una misma variable puede almacenar diferentes tipos de datos durante la ejecución del script.
- **No es necesario terminar cada sentencia con el carácter de punto y coma (;)**: en la mayoría de lenguajes de programación, es obligatorio terminar cada sentencia con el carácter ;. Aunque JavaScript no obliga a hacerlo, es conveniente seguir la tradición de terminar cada sentencia con el carácter del punto y coma (;).
- Se pueden incluir comentarios: los comentarios se utilizan para añadir información en el código fuente del programa. Aunque el contenido de los comentarios no se visualiza por pantalla, si que se envía al navegador del usuario junto con el resto del script, por lo que es necesario extremar las precauciones sobre la información incluida en los comentarios.

JavaScript define dos tipos de comentarios: los de una sola línea y los que ocupan varias líneas.

Ejemplo de comentario de una sola línea:

```
// a continuación se muestra un mensaje
alert("mensaje de prueba");
```

Los comentarios de una sola línea se definen añadiendo dos barras oblicuas (//) al principio de la línea.

Ejemplo de comentario de varias líneas:

```
/* Los comentarios de varias líneas son muy útiles
cuando se necesita incluir bastante información
en los comentarios */
alert("mensaje de prueba");
```

Los comentarios multilínea se definen encerrando el texto del comentario entre los símbolos /* y */.

Desde su aparición, JavaScript siempre fue utilizado de forma masiva por la mayoría de sitios de Internet. La aparición de Flash disminuyó su popularidad, ya que Flash permitía realizar algunas acciones imposibles de llevar a cabo mediante JavaScript.

Sin embargo, la aparición de las aplicaciones AJAX programadas con JavaScript le ha devuelto una popularidad sin igual dentro de los lenguajes de programación web.

JavaScript fue diseñado de forma que se ejecutara en un entorno muy limitado que permitiera a los usuarios confiar en la ejecución de los scripts. De esta forma, los scripts de JavaScript no pueden comunicarse con recursos que no pertenezcan al mismo dominio desde el que se descargó el script. Los scripts tampoco pueden cerrar ventanas que no hayan abierto esos mismos scripts. Las ventanas que se crean no pueden ser demasiado pequeñas ni demasiado grandes ni colocarse fuera de la vista del usuario (aunque los detalles concretos dependen de cada navegador).

Además, los scripts no pueden acceder a los archivos del ordenador del usuario (ni en modo lectura ni en modo escritura) y tampoco pueden leer o modificar las preferencias del navegador.

Por último, si la ejecución de un script dura demasiado tiempo (por ejemplo por un error de programación) el navegador informa al usuario de que un script está consumiendo demasiados recursos y le da la posibilidad de detener su ejecución.

A pesar de todo, existen alternativas para poder saltarse algunas de las limitaciones anteriores. La alternativa más utilizada y conocida consiste en firmar digitalmente el script y solicitar al usuario el permiso para realizar esas acciones.

Esta página se ha dejado vacía a propósito

Capítulo 2

A continuación, se muestra un primer script sencillo pero completo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>El primer script</title>

    <script type="text/javascript">
      alert("Hola Mundo!");
    </script>
</head>

<body>
    <p>Esta página contiene el primer script</p>
</body>
</html>
```

En este ejemplo, el script se incluye como un bloque de código dentro de una página XHTML. Por tanto, en primer lugar se debe crear una página XHTML correcta que incluya la declaración del `DOCTYPE`, el atributo `xmlns`, las secciones `<head>` y `<body>`, la etiqueta `<title>`, etc.

Aunque el código del script se puede incluir en cualquier parte de la página, se recomienda incluirlo en la cabecera del documento, es decir, dentro de la etiqueta `<head>`.

A continuación, el código JavaScript se debe incluir entre las etiquetas `<script>...</script>`. Además, para que la página sea válida, es necesario definir el atributo `type` de la etiqueta `<script>`. Técnicamente, el atributo `type` se corresponde con "el tipo MIME", que es un estándar para identificar los diferentes tipos de contenidos. El "tipo MIME" correcto para JavaScript es `text/javascript`.

Una vez definida la zona en la que se incluirá el script, se escriben todas las sentencias que forman la aplicación. Este primer ejemplo es tan sencillo que solamente incluye una sentencia: `alert("Hola Mundo!");`.

La instrucción `alert()` es una de las utilidades que incluye JavaScript y permite mostrar un mensaje en la pantalla del usuario. Si se visualiza la página web de este primer script en cualquier navegador, automáticamente se mostrará una ventana con el mensaje `"Hola Mundo!"`.

A continuación se muestra el resultado de ejecutar el script:

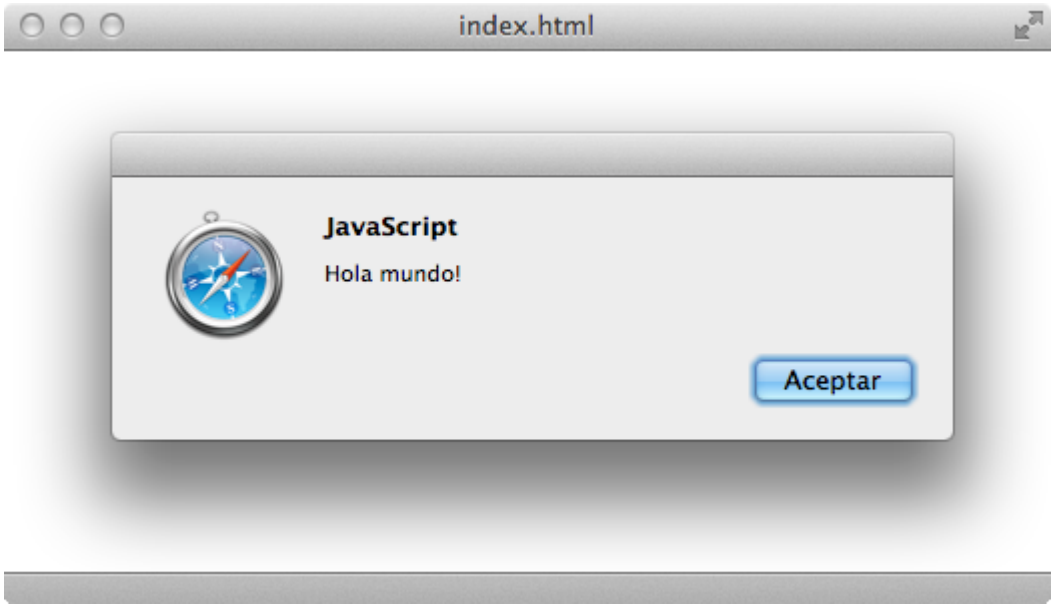


Figura 2.1 Mensaje mostrado con `alert()`

Ejercicio 1

Ver enunciado (#ej01)

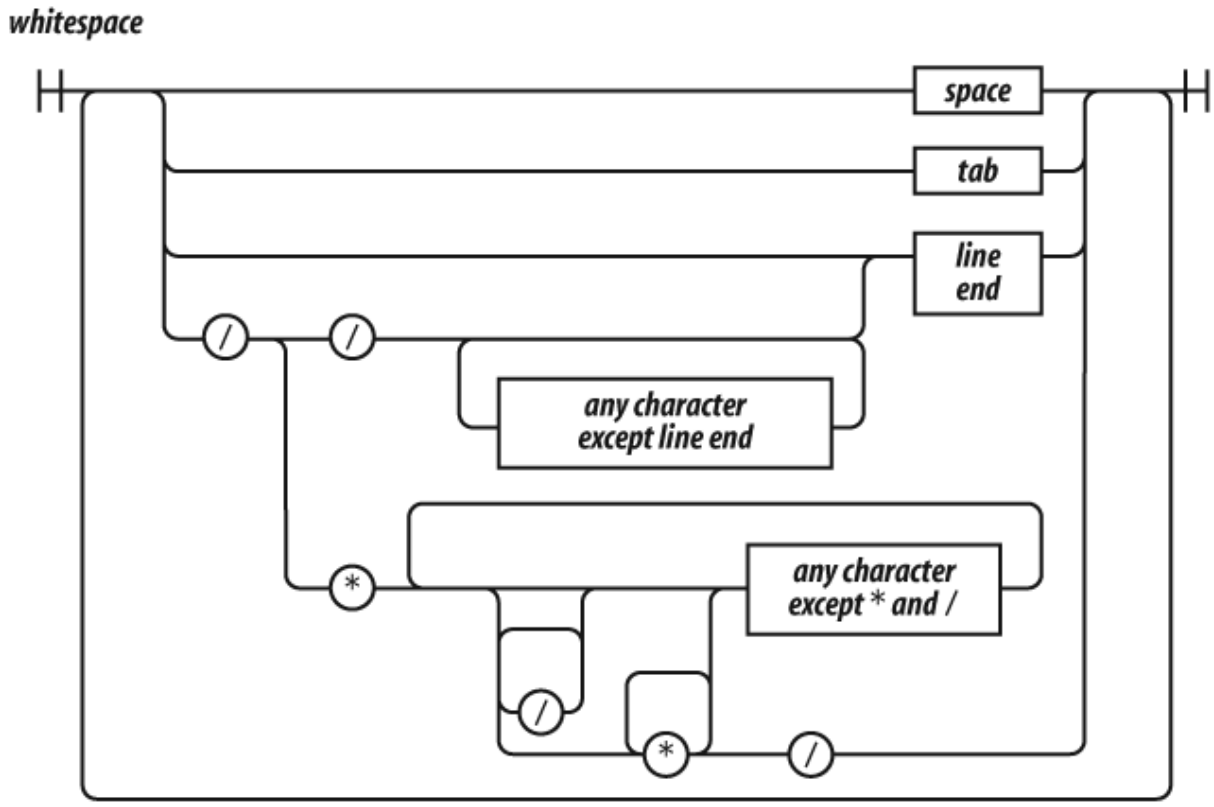
Capítulo 3

Antes de comenzar a desarrollar programas y utilidades con JavaScript, es necesario conocer los elementos básicos con los que se construyen las aplicaciones. Este capítulo explica en detalle y comenzando desde cero los conocimientos básicos necesarios para poder comprender la sintaxis básica de Javascript. En el próximo capítulo veremos aspectos más avanzados como objetos, herencia, arrays o expresiones regulares.

No se tienen en cuenta los espacios en blanco y las nuevas líneas: como sucede con XHTML, el intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.) Sin embargo, en ocasiones estos espacios en blanco son totalmente necesarios, por ejemplo, para reparar nombres de variables o palabras reservadas. Por ejemplo:

```
var that = this;
```

Aquí el espacio en blanco entre `var` y `that` no puede ser eliminado, pero el resto sí.



JavaScript ofrece dos tipos de comentarios, de bloque gracias a los caracteres `/* */` y de línea comenzando con `//`. El formato `/* */`de comentarios puede causar problemas en ciertas condiciones, como en las expresiones regulares, por lo que hay que tener cuidado al utilizarlo. Por ejemplo:

```
/*
    var rm_a = /a*/.match(s);
*/
```

provoca un error de sintaxis. Por lo tanto, suele ser recomendable utilizar únicamente los comentarios de línea, para evitar este tipo de problemas.

Las variables en JavaScript se crean mediante la palabra reservada `var`. De esta forma, podemos declarar variables de la siguiente manera:

```
var numero_1 = 3;
var numero_2 = 1;
var resultado = numero_1 + numero_2;
```

La palabra reservada `var` solamente se debe indicar al declarar por primera vez la variable. Cuando se utilizan las variables en el resto de instrucciones del script, solamente es necesario indicar su nombre. En otras palabras, en el ejemplo anterior sería un error indicar lo siguiente:

```
var numero_1 = 3;
var numero_2 = 1;
var resultado = var numero_1 + var numero_2;
```

En JavaScript no es obligatorio inicializar las variables, ya que se pueden declarar por una parte y asignarles un valor posteriormente. Por tanto, el ejemplo anterior se puede rehacer de la siguiente manera:

```
var numero_1;
var numero_2;

numero_1 = 3;
numero_2 = 1;

var resultado = numero_1 + numero_2;
```

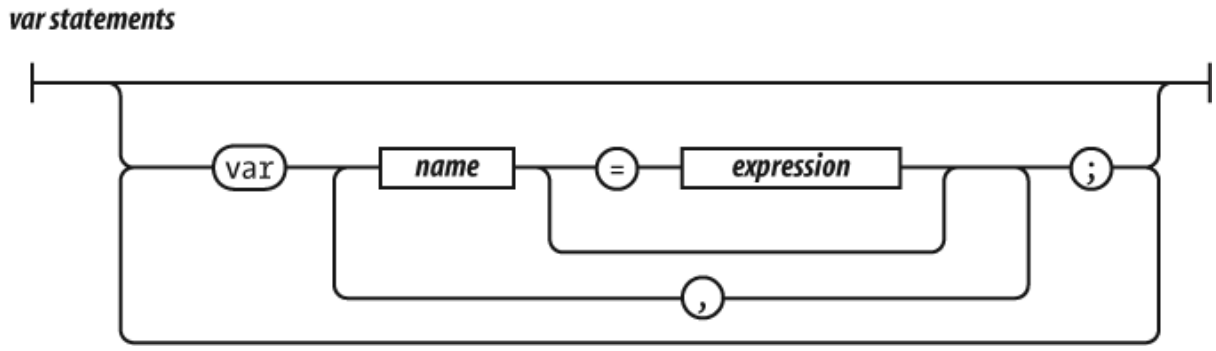
Una de las características más sorprendentes de JavaSript para los programadores habituados a otros lenguajes de programación es que tampoco es necesario declarar las variables. En otras palabras, se pueden utilizar variables que no se han definido anteriormente mediante la palabra reservada `var`. El ejemplo anterior también es correcto en JavaScript de la siguiente forma:

```
var numero_1 = 3;
var numero_2 = 1;
resultado = numero_1 + numero_2;
```

La variable `resultado` no está declarada, por lo que JavaScript crea una variable global (más adelante se verán las diferencias entre variables locales y globales) y le asigna el valor correspondiente. De la misma forma, también sería correcto el siguiente código:

```
numero_1 = 3;
numero_2 = 1;
resultado = numero_1 + numero_2;
```

En cualquier caso, se recomienda declarar todas las variables que se vayan a utilizar.



El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:

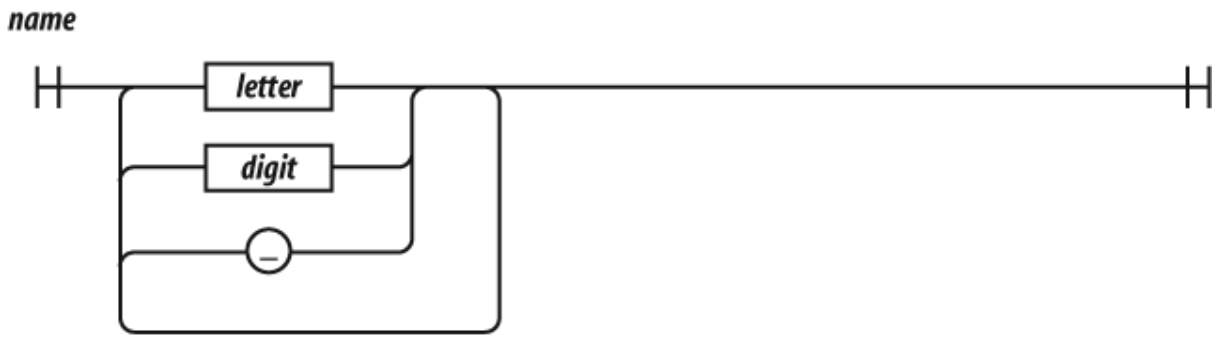
- Sólo puede estar formado por letras, números y los símbolos `$` (dólar) y `_` (guión bajo).
- El primer carácter no puede ser un número.

Por tanto, las siguientes variables tienen nombres correctos:

```
var $numero1;
var _$letra;
var $$$otroNumero;
var $_a__$4;
```

Sin embargo, las siguientes variables tienen identificadores incorrectos:

```
var 1numero;        // Empieza por un número
var numero;1_123;   // Contiene un carácter ";"
```



A continuación se indica el listado de palabras reservadas en JavaScript, y que no podremos utilizar para nombrar nuestras variables, parámetros, funciones, operadores o etiquetas:

- `abstract`

- `boolean break byte`
- `case catch char class const continue`
- `debugger default delete do double`
- `else enum export extends`
- `false final finally float for function`
- `goto`
- `if implements import in instanceof int interface`
- `long`
- `native new null`
- `package private protected public`
- `return`
- `short static super switch synchronized`
- `this throw throws transient true try typeof`
- `var volatile void`
- `while with`

JavaScript divide los distintos tipos de variables en dos grupos: tipos primitivos y tipos de referencia o clases.

JavaScript define cinco tipos primitivos: `undefined`, `null`, `boolean`, `number` y `string`. Además de estos tipos, JavaScript define el operador `typeof` para averiguar el tipo de una variable.

El tipo `undefined` corresponde a las variables que han sido definidas y todavía no se les ha asignado un valor:

```
var variable1;
typeof variable1;  // devuelve "undefined"
```

Se trata de un tipo similar a `undefined`, y de hecho en JavaScript se consideran iguales (`undefined == null`). El tipo `null` se suele utilizar para representar objetos que en ese momento no existen.

```
var nombreUsuario = null;
```

Se trata de una variable que sólo puede almacenar uno de los dos valores especiales definidos y que representan el valor *"verdadero"* y el valor *"falso"*.

```
var variable1 = true;
var variable2 = false;
```

Los valores `true` y `false` son valores especiales, de forma que no son palabras ni números ni ningún otro tipo de valor. Este tipo de variables son esenciales para crear cualquier aplicación, tal y como se verá más adelante.

Cuando es necesario convertir una variable numérica a una variable de tipo `boolean`, JavaScript aplica la siguiente conversión: el número `0` se convierte en `false` y cualquier otro número distinto de `0` se convierte en `true`.

Por este motivo, en ocasiones se asocia el número `0` con el valor `false` y el número `1` con el valor `true`. Sin embargo, es necesario insistir en que `true` y `false` son valores especiales que no se corresponden ni con números ni con ningún otro tipo de dato.

JavaScript es un lenguaje de programación *"no tipado"*, lo que significa que una misma variable puede guardar diferentes tipos de datos a lo largo de la ejecución de la aplicación. De esta forma, una variable se podría inicializar con un valor numérico, después podría almacenar una cadena de texto y podría acabar la ejecución del programa en forma de variable booleana.

No obstante, en ocasiones es necesario que una variable almacene un dato de un determinado tipo. Para asegurar que así sea, se puede convertir una variable de un tipo a otro, lo que se denomina *typecasting*:

Así, JavaScript incluye un método llamado `toString()` que permite convertir variables de cualquier tipo a variables de cadena de texto, tal y como se muestra en el siguiente ejemplo:

```
var variable1 = true;
variable1.toString();  // devuelve "true" como cadena de texto
```

```
var variable2 = 5;
variable2.toString();  // devuelve "5" como cadena de texto
```

JavaScript también incluye métodos para convertir los valores de las variables en valores numéricos. Los métodos definidos son `parseInt()` y `parseFloat()`, que convierten la variable que se le indica en un número entero o un número decimal respectivamente.

La conversión numérica de una cadena se realiza carácter a carácter empezando por el de la primera posición. Si ese carácter no es un número, la función devuelve el valor NaN. Si el primer carácter es un número, se continúa con los siguientes caracteres mientras estos sean números.

```
var variable1 = "hola";
parseInt(variable1);  // devuelve NaN
var variable2 = "34";
parseInt(variable2);  // devuelve 34
var variable3 = "34hola23";
parseInt(variable3);  // devuelve 34
var variable4 = "34.23";
parseInt(variable4);  // devuelve 34
```

En el caso de `parseFloat()`, el comportamiento es el mismo salvo que también se considera válido el carácter . que indica la parte decimal del número:

```
var variable1 = "hola";
parseFloat(variable1);  // devuelve NaN
var variable2 = "34";
parseFloat(variable2);  // devuelve 34.0
var variable3 = "34hola23";
parseFloat(variable3);  // devuelve 34.0
var variable4 = "34.23";
```

Aunque JavaScript no define el concepto de clase, los tipos de referencia se asemejan a las clases de otros lenguajes de programación. Los objetos en JavaScript se crean mediante la palabra reservada `new` y el nombre de la clase que se va a instanciar. De esta forma, para crear un objeto de tipo `String` se indica lo siguiente (los paréntesis solamente son obligatorios cuando se utilizan argumentos, aunque se recomienda incluirlos incluso cuando no se utilicen):

```
var variable1 = new String("hola mundo");
```

JavaScript define una clase para cada uno de los tipos de datos primitivos. De esta forma, existen objetos de tipo `Boolean` para las variables booleanas, `Number` para las variables numéricas y `String` para las variables de cadenas de texto. Las clases `Boolean`, `Number` y `String` almacenan los mismos valores de los tipos de datos primitivos y añaden propiedades y métodos para manipular sus valores.

```
var longitud = "hola mundo".length;
```

La propiedad `length` sólo está disponible en la clase `String`, por lo que en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto. Sin embargo, JavaScript convierte el tipo de dato primitivo al tipo de referencia `String`, obtiene el valor de la propiedad `length` y devuelve el resultado. Este proceso se realiza de forma automática y transparente para el programador.

En realidad, con una variable de tipo `String` no se pueden hacer muchas más cosas que con su correspondiente tipo de dato primitivo. Por este motivo, no existen muchas diferencias prácticas entre utilizar el tipo de referencia o el tipo primitivo, salvo en el caso del resultado del operador `typeof` y en el caso de la función `eval()`, como se verá más adelante.

La principal diferencia entre los tipos de datos es que los datos primitivos se manipulan por valor y los tipos de referencia se manipulan, como su propio nombre indica, por referencia. Los conceptos *"por valor"* y *"por referencia"* son iguales que en el resto de lenguajes de programación, aunque existen diferencias importantes (no existe por ejemplo el concepto de puntero).

Cuando un dato se manipula por valor, lo único que importa es el valor en sí. Cuando se asigna una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda. Cualquier modificación que se realice en la segunda variable es independiente de la primera variable.

De la misma forma, cuando se pasa una variable por valor a una función (como se explicará más adelante) sólo se pasa una copia del valor. Así, cualquier modificación que realice la función sobre el valor pasado no se refleja en el valor de la variable original.

En el siguiente ejemplo, una variable se asigna por valor a otra variable:

```
var variable1 = 3;
var variable2 = variable1;

variable2 = variable2 + 5;
// Ahora variable2 = 8 y variable1 sigue valiendo 3
```

La `variable1` se asigna por valor en la `variable1`. Aunque las dos variables almacenan en ese momento el mismo valor, son independientes y cualquier cambio en una de ellas no afecta a la otra. El motivo es que los tipos de datos primitivos siempre se asignan (y se pasan) por valor.

Sin embargo, en el siguiente ejemplo, se utilizan tipos de datos de referencia:

```
// variable1 = 25 diciembre de 2009
var variable1 = new Date(2009, 11, 25);
// variable2 = 25 diciembre de 2009
var variable2 = variable1;

// variable2 = 31 diciembre de 2010
variable2.setFullYear(2010, 11, 31);
// Ahora variable1 también es 31 diciembre de 2010
```

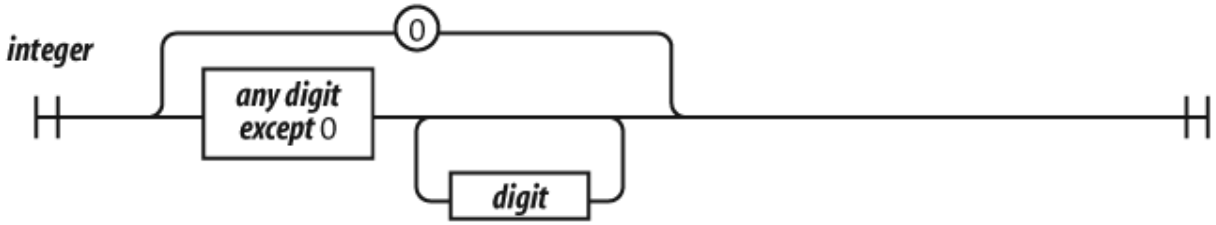
En el ejemplo anterior, se utiliza un tipo de dato de referencia que se verá más adelante, que se llama `Date` y que se utiliza para manejar fechas. Se crea una variable llamada `variable1` y se inicializa la fecha a 25 de diciembre de 2009. Al constructor del objeto `Date` se le pasa el año, el número del mes (siendo `0` = enero, `1` = febrero, ..., `11` = diciembre) y el día (al contrario que el mes, los días no empiezan en `0` sino en `1`). A continuación, se asigna el valor de la `variable1` a otra variable llamada `variable2`.

Como `Date` es un tipo de referencia, la asignación se realiza por referencia. Por lo tanto, las dos variables quedan "unidas" y hacen referencia al mismo objeto, al mismo dato de tipo `Date`. De esta forma, si se modifica el valor de `variable2` (y se cambia su fecha a 31 de diciembre de 2010) el valor de `variable1` se verá automáticamente modificado.

En JavaScript únicamente existe un tipo de número. Internamente, es representado como un dato de `64 bits` en coma flotante, al igual el tipo de dato `double` en Java. A diferencia de otros lenguajes de programación, no existe una diferencia entre un número entero y otro decimal, por lo que `1` y `1.0`son el mismo valor. Esto es significativo ya que evitamos los

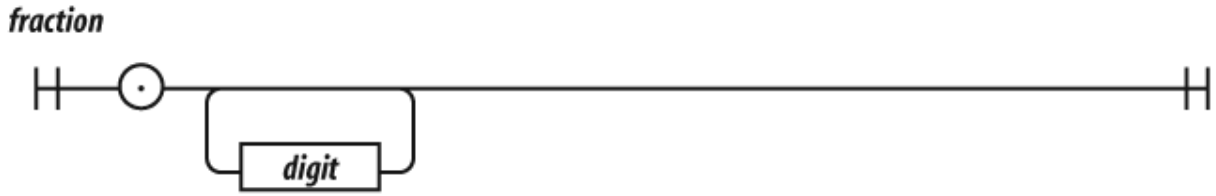
problemas desbordamiento en tipos de dato *pequeños*, al no existir la necesidad de conocer el tipo de dato.

Si el número es entero, se indica su valor directamente.



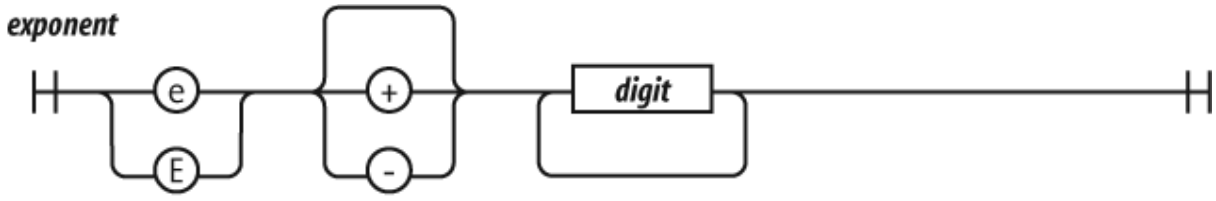

```
var variable1 = 10;
```

Si el número es decimal, se debe utilizar el punto (.) para separar la parte entera de la decimal.




```
var variable2 = 3.14159265;
```

Además del sistema numérico decimal, también se pueden indicar valores en el sistema octal (si se incluye un cero delante del número) y en sistema hexadecimal (si se incluye un cero y una x delante del número).




```
var variable1 = 10;
var variable_octal = 034;
var variable_hexadecimal = 0xA3;
```

JavaScript define tres valores especiales muy útiles cuando se trabaja con números. En primer lugar se definen los valores `Infinity` y `–Infinity`

para representar números demasiado grandes (positivos y negativos) y con los que JavaScript no puede trabajar.

```
var variable1 = 3, variable2 = 0;
alert(variable1/variable2);  // muestra "Infinity"
```

El otro valor especial definido por JavaScript es `NaN`, que es el acrónimo de *"Not a Number"*. De esta forma, si se realizan operaciones matemáticas con variables no numéricas, el resultado será de tipo `NaN`.

Para manejar los valores `NaN`, se utiliza la función relacionada `isNaN()`, que devuelve `true` si el parámetro que se le pasa no es un número:

```
var variable1 = 3;
var variable2 = "hola";
isNaN(variable1);  // false
isNaN(variable2);  // true
isNaN(variable1 + variable2);  // true
```

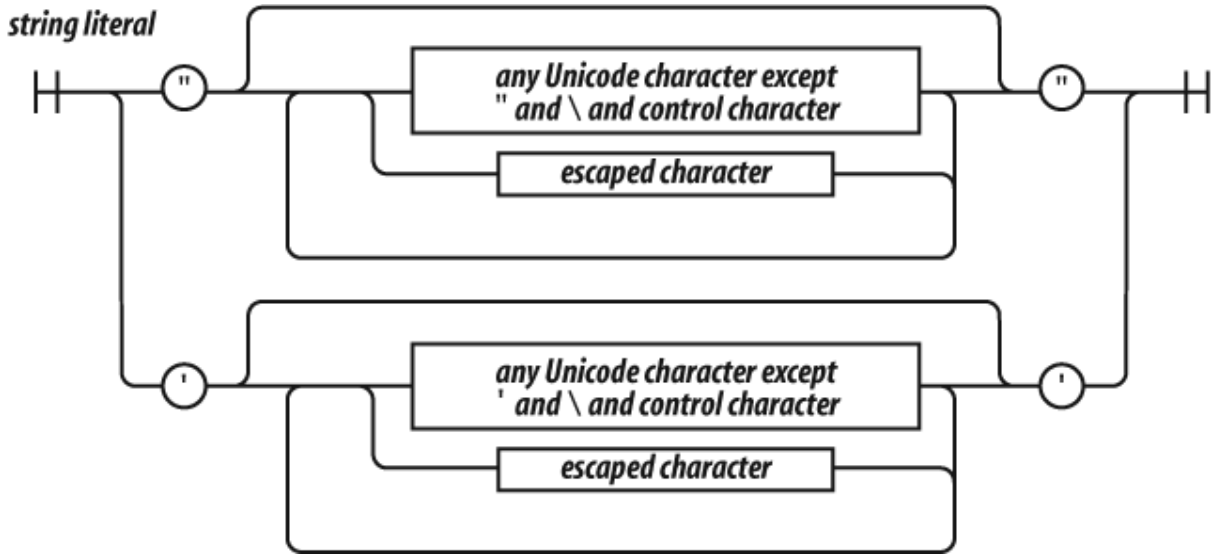
Por último, JavaScript define algunas constantes matemáticas que representan valores numéricos significativos:

`Math.E`	`2.718281828459045`	Constante de Euler, base de los logaritmos naturales y también llamado *número e*
`Math.LN2`	`0.6931471805599453`	Logaritmo natural de `2`
`Math.LN10`	`2.302585092994046`	Logaritmo natural de `10`
`Math.LOG2E`	`1.4426950408889634`	Logaritmo en base `2` de `Math.E`
`Math.LOG10E`	`0.4342944819032518`	Logaritmo en base `10` de `Math.E`
`Math.PI`	`3.141592653589793`	Pi, relación entre el radio de una circunferencia y su diámetro
`Math.SQRT1_2`	`0.7071067811865476`	Raíz cuadrada de `1/2`
`Math.SQRT2`	`1.4142135623730951`	Raíz cuadrada de `2`

De esta forma, para calcular el área de un círculo de radio r, se debe utilizar la constante que representa al número Pi:

```
var area = Math.PI * r * r;
```

Las variables de tipo cadena de texto permiten almacenar cuaquier sucesión de caracteres, por lo que se utilizan ampliamente en la mayoría de aplicaciones JavaScript. Cada carácter de la cadena se encuentra en una posición a la que se puede acceder individualmente, siendo el primer carácter el de la posición `0`.



El valor de las cadenas de texto se indica encerrado entre comillas simples o dobles:

```
var variable1 = "hola";
var variable2 = 'mundo';
var variable3 = "hola mundo, esta es una frase más larga";
```

Las cadenas de texto pueden almacenar cualquier carácter, aunque algunos no se pueden incluir directamente en la declaración de la variable. Si por ejemplo se incluye un ENTER para mostrar el resto de caracteres en la línea siguiente, se produce un error en la aplicación:

```
var variable = "hola mundo, esta es
                                una frase más larga";
```

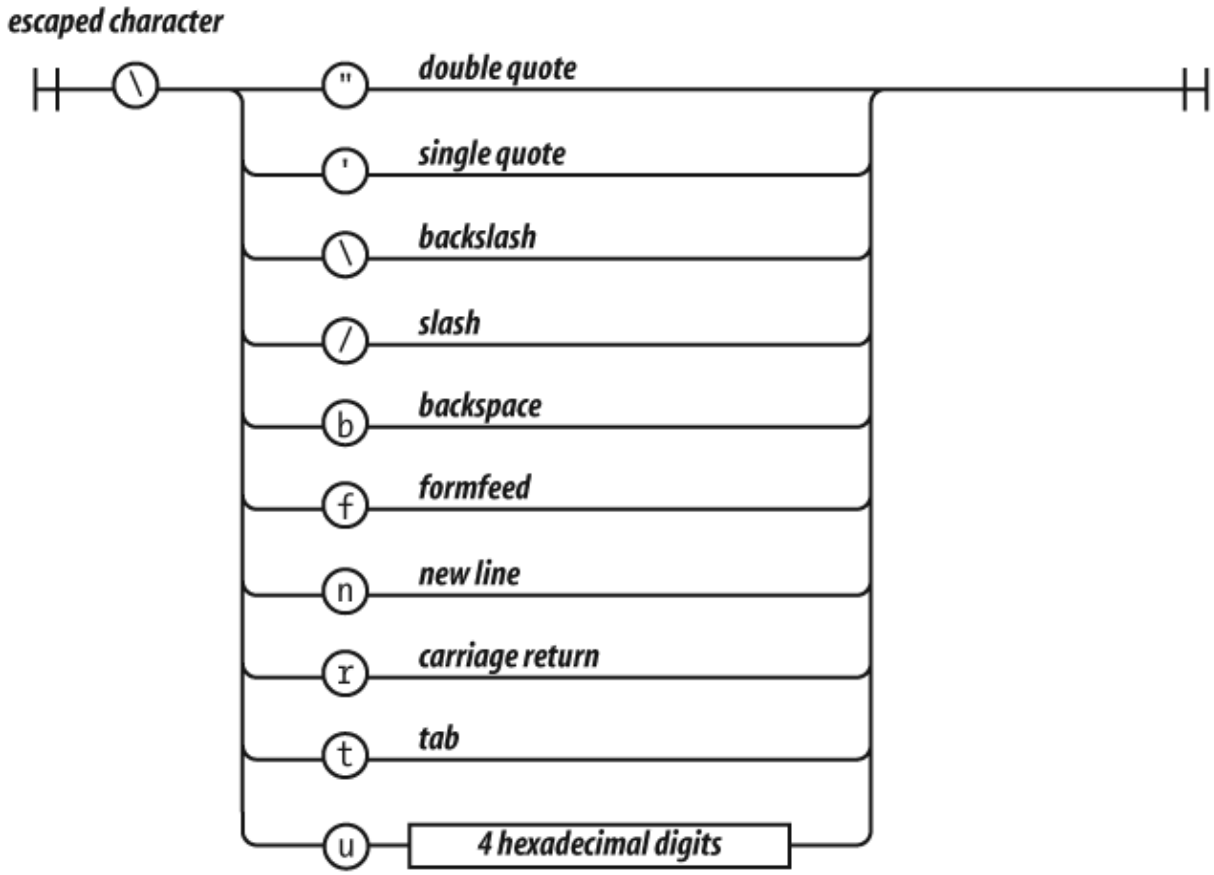
La variable anterior no está correctamente definida y se producirá un error en la aplicación. Por tanto, resulta evidente que algunos caracteres *especiales* no se pueden incluir directamente. De la misma forma, como las comillas (doble y simple) se utilizan para encerrar los contenidos, también se pueden producir errores:

```
var variable1 = "hola 'mundo'";
var variable2 = 'hola "mundo"';
var variable3 = "hola 'mundo', esta es una "frase" más larga";
```

Si el contenido de texto tiene en su interior alguna comilla simple, se encierran los contenidos con comillas dobles (como en el caso de la `variable1` anterior). Si el contenido de texto tiene en su interior alguna comilla doble, se encierran sus contenidos con comillas simples (como en el caso de la `variable2` anterior). Sin embargo, en el caso de la `variable3` su contenido tiene tanto comillas simples como comillas dobles, por lo que su declaración provocará un error.

Para resolver estos problemas, JavaScript define un mecanismo para incluir de forma sencilla caracteres especiales (`ENTER`, `Tabulador`) y problemáticos (comillas). Esta estrategia se denomina "mecanismo de escape", ya que se sustituyen los caracteres problemáticos por otros caracteres seguros que siempre empiezan con la barra `\`:

Una nueva línea	`\n`
Un tabulador	`\t`
Una comilla simple	`\'`
Una comilla doble	`\"`
Una barra inclinada	`\\`



Utilizando el mecanismo de escape, se pueden corregir los ejemplos anteriores:

```
var variable = "hola mundo, esta es \n una frase más larga";
var variable3 = "hola 'mundo', esta es una \"frase\" más larga";
```

Capítulo 4

Los operadores permiten manipular el valor de las variables, realizar operaciones matemáticas con sus valores y comparar diferentes variables. De esta forma, los operadores permiten a los programas realizar cálculos complejos y tomar decisiones lógicas en función de comparaciones y otros tipos de condiciones.

El operador de asignación es el más utilizado y el más sencillo. Este operador se utiliza para guardar un valor específico en una variable. El símbolo utilizado es = (no confundir con el operador == que se verá más adelante):

```
var numero1 = 3;
```

A la izquierda del operador, siempre debe indicarse el nombre de una variable. A la derecha del operador, se pueden indicar variables, valores, condiciones lógicas, etc:

```
var numero1 = 3;
var numero2 = 4;

/* Error, la asignación siempre se realiza a una variable,
    por lo que en la izquierda no se puede indicar un número */
5 = numero1;

// Ahora, la variable numero1 vale 5
numero1 = 5;
```

```
// Ahora, la variable numero1 vale 4
numero1 = numero2;
```

JavaScript permite realizar manipulaciones matemáticas sobre el valor de las variables numéricas. Los operadores definidos son: suma (+), resta (-), multiplicación (*), división (/) y módulo (%). Ejemplo:

```
var numero1 = 10;
var numero2 = 5;

resultado = numero1 / numero2;  // resultado = 2
resultado = 3 + numero1;        // resultado = 13
resultado = numero2 – 4;        // resultado = 1
resultado = numero1 * numero 2; // resultado = 50
resultado = numero1 % numero2;  // resultado = 0

numero1 = 9;
numero2 = 5;
resultado = numero1 % numero2;  // resultado = 4
```

Los operadores matemáticos también se pueden combinar con el operador de asignación para abreviar su notación:

```
var numero1 = 5;
numero1 += 3;   // numero1 = numero1 + 3 = 8
numero1 -= 1;   // numero1 = numero1 - 1 = 4
numero1 *= 2;   // numero1 = numero1 * 2 = 10
numero1 /= 5;   // numero1 = numero1 / 5 = 1
numero1 %= 4;   // numero1 = numero1 % 4 = 1
```

Existen dos operadores especiales que solamente son válidos para las variables numéricas y se utilizan para incrementar o decrementar en una unidad el valor de una variable.

Ejemplo:

```
var numero = 5;
++numero;
alert(numero);  // numero = 6
```

El operador de incremento se indica mediante el prefijo ++ en el nombre de la variable. El resultado es que el valor de esa variable se incrementa en una unidad. Por tanto, el anterior ejemplo es equivalente a:

```
var numero = 5;
numero = numero + 1;
alert(numero);  // numero = 6
```

De forma equivalente, el operador decremento (indicado como un prefijo -- en el nombre de la variable) se utiliza para decrementar el valor de la variable:

```
var numero = 5;
--numero;
alert(numero);  // numero = 4
```

El anterior ejemplo es equivalente a:

```
var numero = 5;
numero = numero - 1;
alert(numero);  // numero = 4
```

Los operadores de incremento y decremento no solamente se pueden indicar como prefijo del nombre de la variable, sino que también es posible utilizarlos como sufijo. En este caso, su comportamiento es similar pero muy diferente. En el siguiente ejemplo:

```
var numero = 5;
numero++;
alert(numero);  // numero = 6
```

El resultado de ejecutar el script anterior es el mismo que cuando se utiliza el operador `++numero`, por lo que puede parecer que es equivalente indicar el operador ++ delante o detrás del identificador de la variable. Sin embargo, el siguiente ejemplo muestra sus diferencias:

```
var numero1 = 5;
var numero2 = 2;
numero3 = numero1++ + numero2;
// numero3 = 7, numero1 = 6

var numero1 = 5;
var numero2 = 2;
```

```
numero3 = ++numero1 + numero2;
// numero3 = 8, numero1 = 6
```

Si el operador ++ se indica como prefijo del identificador de la variable, su valor se incrementa **antes** de realizar cualquier otra operación. Si el operador ++ se indica como sufijo del identificador de la variable, su valor se incrementa **después** de ejecutar la sentencia en la que aparece.

Por tanto, en la instrucción `numero3 = numero1++ + numero2;`, el valor de `numero1` se incrementa después de realizar la operación (primero se suma y `numero3` vale `7`, después se incrementa el valor de `numero1` y vale `6`). Sin embargo, en la instrucción `numero3 = ++numero1 + numero2;`, en primer lugar se incrementa el valor de `numero1` y después se realiza la suma (primero se incrementa `numero1` y vale `6`, después se realiza la suma y `numero3` vale `8).

Los operadores lógicos son imprescindibles para realizar aplicaciones complejas, ya que se utilizan para tomar decisiones sobre las instrucciones que debería ejecutar el programa en función de ciertas condiciones. El resultado de cualquier operación que utilice operadores lógicos siempre es un valor lógico o `booleano`.

Uno de los operadores lógicos más utilizados es el de la negación. Se utiliza para obtener el valor contrario al valor de la variable:

```
var visible = true;
alert(!visible);  // Muestra "false" y no "true"
```

La negación lógica se obtiene prefijando el símbolo `!` al identificador de la variable. El funcionamiento de este operador se resume en la siguiente tabla:

`true`	`false`
`false`	`true`

Si la variable original es de tipo *booleano*, es muy sencillo obtener su negación. Sin embargo, ¿qué sucede cuando la variable es un número o

una cadena de texto? Para obtener la negación en este tipo de variables, se realiza en primer lugar su conversión a un valor booleano:

- Si la variable contiene un número, se transforma en `false` si vale `0` y en `true` para cualquier otro número (positivo o negativo, decimal o entero).
- Si la variable contiene una cadena de texto, se transforma en `false` si la cadena es vacía (`""`) y en `true` en cualquier otro caso.

```
var cantidad = 0;
vacio = !cantidad;  // vacio = true

cantidad = 2;
vacio = !cantidad;  // vacio = false

var mensaje = "";
mensajeVacio = !mensaje;  // mensajeVacio = true

mensaje = "Bienvenido";
mensajeVacio = !mensaje;  // mensajeVacio = false
```

La operación lógica `AND` obtiene su resultado combinando dos valores booleanos. El operador se indica mediante el símbolo `&&` y su resultado solamente es `true` si los dos operandos son `true`:

true	true	true
true	false	false
false	true	false
false	false	false

```
var valor1 = true;
var valor2 = false;
resultado = valor1 && valor2; // resultado = false

valor1 = true;
valor2 = true;
resultado = valor1 && valor2; // resultado = true
```

La operación lógica OR también combina dos valores booleanos. El operador se indica mediante el símbolo || y su resultado es true si alguno de los dos operandos es true:

true	true	true
true	false	false
false	true	true
false	false	false

```
var valor1 = true;
var valor2 = false;
resultado = valor1 || valor2; // resultado = true

valor1 = false;
valor2 = false;
resultado = valor1 || valor2; // resultado = false
```

Los operadores relacionales definidos por JavaScript son idénticos a los que definen las matemáticas: mayor que (>), menor que (<), mayor o igual (>=), menor o igual (<=), igual que (==) y distinto de (!=).

Los operadores que relacionan variables son imprescindibles para realizar cualquier aplicación compleja. El resultado de todos estos operadores siempre es un valor booleano:

```
var numero1 = 3;
var numero2 = 5;
resultado = numero1 > numero2;  // resultado = false
resultado = numero1 < numero2;  // resultado = true

numero1 = 5;
numero2 = 5;
resultado = numero1 >= numero2; // resultado = true
resultado = numero1 <= numero2; // resultado = true
resultado = numero1 == numero2; // resultado = true
resultado = numero1 != numero2; // resultado = false
```

Se debe tener especial cuidado con el operador de igualdad (==), ya que es el origen de la mayoría de errores de programación, incluso para los usuarios que ya tienen cierta experiencia desarrollando scripts. El operador == se utiliza para comparar el valor de dos variables, por lo que es muy diferente del operador =, que se utiliza para asignar un valor a una variable:

```
// El operador "=" asigna valores
var numero1 = 5;
resultado = numero1 = 3;  // numero1 = 3 y resultado = 3

// El operador "==" compara variables
var numero1 = 5;
resultado = numero1 == 3; // numero1 = 5 y resultado = false
```

Los operadores relacionales también se pueden utilizar con variables de tipo cadena de texto:

```
var texto1 = "hola";
var texto2 = "hola";
var texto3 = "adios";

resultado = texto1 == texto3; // resultado = false
resultado = texto1 != texto2; // resultado = false
resultado = texto3 >= texto2; // resultado = false
```

Cuando se utilizan cadenas de texto, los operadores "mayor que" (>) y "menor que" (<) siguen un razonamiento no intuitivo: se compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia entre las dos cadenas de texto. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (`a` es menor que `b`, `b` es menor que `c`, `A` es menor que `a`, etc.)

El operador `typeof` se emplea para determinar el tipo de dato que almacena una variable. Su uso es muy sencillo, ya que sólo es necesario indicar el nombre de la variable cuyo tipo se quiere averiguar:

```
var myFunction = function() {
  console.log('hola');
};
```

```
var myObject = {
    foo : 'bar'
};

var myArray = [ 'a', 'b', 'c' ];

var myString = 'hola';

var myNumber = 3;

typeof myFunction;   // devuelve 'function'
typeof myObject;     // devuelve 'object'
typeof myArray;      // devuelve 'object' -- tenga cuidado
typeof myString;     // devuelve 'string'
typeof myNumber;     // devuelve 'number'

typeof null;         // devuelve 'object' -- tenga cuidado

if (myArray.push && myArray.slice && myArray.join) {
    // probablemente sea un vector
    // (este estilo es llamado, en inglés, "duck typing")
}

if (Object.prototype.toString.call(myArray) === '[object Array]') {
    // definitivamente es un vector;
    // esta es considerada la forma más robusta
    // de determinar si un valor es un vector.
}
```

Los posibles valores de retorno del operador son: `undefined`, `boolean`, `number`, `string` para cada uno de los tipos primitivos y `object` para los valores de referencia y también para los valores de tipo `null`.

El operador `typeof` no distingue entre las variables declaradas pero no inicializadas y las variables que ni siquiera han sido declaradas:

```
var variable1;

// devuelve "undefined", aunque la variable1 ha sido declarada
typeof variable1;
// devuelve "undefined", la variable2 no ha sido declarada
typeof variable2;
```

El operador `typeof` no es suficiente para trabajar con tipos de referencia, ya que devuelve el valor `object` para cualquier objeto independientemente de su tipo. Por este motivo, JavaScript define el operador `instanceof` para determinar la clase concreta de un objeto.

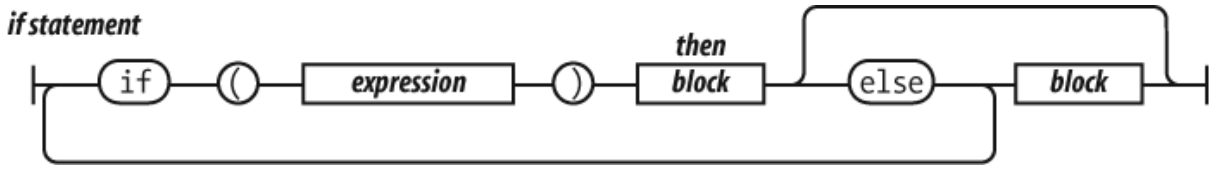
```
var variable1 = new String("hola mundo");
typeof variable1;              // devuelve "object"
variable1 instanceof String;   // devuelve true
```

El operador `instanceof` sólo devuelve como valor `true` o `false`. De esta forma, `instanceof` no devuelve directamente la clase de la que ha instanciado la variable, sino que se debe comprobar cada posible tipo de clase individualmente.

Esta página se ha dejado vacía a propósito

Capítulo 5

La estructura más utilizada en JavaScript y en la mayoría de lenguajes de programación es la estructura `if`. Se emplea para tomar decisiones en función de una condición. Su definición formal es:




```
if(condicion) {
  ...
}
```

Si la condición se cumple (es decir, si su valor es `true`) se ejecutan todas las instrucciones que se encuentran dentro del bloque `{...}`. Si la condición no se cumple (es decir, si su valor es `false`) no se ejecuta ninguna instrucción contenida en `{...}` y el programa continúa ejecutando el resto de instrucciones del script.

Ejemplo:

```
var mostrarMensaje = true;

if(mostrarMensaje) {
  alert("Hola Mundo");
}
```

En el ejemplo anterior, el mensaje sí que se muestra al usuario ya que la variable `mostrarMensaje` tiene un valor de `true` y por tanto, el programa entra dentro del bloque de instrucciones del `if`.

El ejemplo se podría reescribir también como:

```
var mostrarMensaje = true;

if(mostrarMensaje == true) {
  alert("Hola Mundo");
}
```

En este caso, la condición es una comparación entre el valor de la variable `mostrarMensaje` y el valor `true`. Como los dos valores coinciden, la igualdad se cumple y por tanto la condición es cierta, su valor es `true` y se ejecutan las instrucciones contenidas en ese bloque del `if`.

La comparación del ejemplo anterior suele ser el origen de muchos errores de programación, al confundir los operadores `==` y `=`. Las comparaciones siempre se realizan con el operador `==`, ya que el operador `=` solamente asigna valores:

```
var mostrarMensaje = true;

// Se comparan los dos valores
if(mostrarMensaje == false) {
  ...
}

// Error - Se asigna el valor "false" a la variable
if(mostrarMensaje = false) {
  ...
}
```

La condición que controla el `if()` puede combinar los diferentes operadores lógicos y relacionales mostrados anteriormente:

```
var mostrado = false;

if(!mostrado) {
  alert("Es la primera vez que se muestra el mensaje");
}
```

Los operadores `AND` y `OR` permiten encadenar varias condiciones simples para construir condiciones complejas:

```
var mostrado = false;
var usuarioPermiteMensajes = true;

if(!mostrado && usuarioPermiteMensajes) {
  alert("Es la primera vez que se muestra el mensaje");
}
```

La condición anterior está formada por una operación `AND` sobre dos variables. A su vez, a la primera variable se le aplica el operador de negación antes de realizar la operación `AND`. De esta forma, como el valor de `mostrado` es `false`, el valor `!mostrado` sería `true`. Como la variable `usuarioPermiteMensajes` vale `true`, el resultado de `!mostrado && usuarioPermiteMensajes` sería igual a `true && true`, por lo que el resultado final de la condición del `if()` sería `true` y por tanto, se ejecutan las instrucciones que se encuentran dentro del bloque del `if()`.

En ocasiones, las decisiones que se deben realizar no son del tipo *"si se cumple la condición, hazlo; si no se cumple, no hagas nada"*. Normalmente las condiciones suelen ser del tipo *"si se cumple esta condición, hazlo; si no se cumple, haz esto otro"*.

Para este segundo tipo de decisiones, existe una variante de la estructura `if` llamada `if...else`. Su definición formal es la siguiente:

```
if(condicion) {
  ...
}
else {
  ...
}
```

Si la condición se cumple (es decir, si su valor es `true`) se ejecutan todas las instrucciones que se encuentran dentro del `if()`. Si la condición no se cumple (es decir, si su valor es `false`) se ejecutan todas las instrucciones contenidas en `else { }`. Ejemplo:

```
var edad = 18;

if(edad >= 18) {
  alert("Eres mayor de edad");
```

```
}
else {
  alert("Todavía eres menor de edad");
}
```

Si el valor de la variable `edad` es mayor o igual que el valor numérico `18`, la condición del `if()` se cumple y por tanto, se ejecutan sus instrucciones y se muestra el mensaje `"Eres mayor de edad"`. Sin embargo, cuando el valor de la variable `edad` no es igual o mayor que `18`,la condición del `if()` no se cumple, por lo que automáticamente se ejecutan todas las instrucciones del bloque `else { }`. En este caso, se mostraría el mensaje `"Todavía eres menor de edad"`.

El siguiente ejemplo compara variables de tipo cadena de texto:

```
var nombre = "";

if(nombre == "") {
  alert("Aún no nos has dicho tu nombre");
}
else {
  alert("Hemos guardado tu nombre");
}
```

La condición del `if()` anterior se construye mediante el operador `==`, que es el que se emplea para comparar dos valores (no confundir con el operador `=` que se utiliza para asignar valores). En el ejemplo anterior, si la cadena de texto almacenada en la variable `nombre` es vacía (es decir, es igual a `""`) se muestra el mensaje definido en el `if()`. En otro caso, se muestra el mensaje definido en el bloque `else { }`.

La estructura `if...else` se puede encadenar para realizar varias comprobaciones seguidas:

```
if(edad < 12) {
  alert("Todavía eres muy pequeño");
}
else if(edad < 19) {
  alert("Eres un adolescente");
}
else if(edad < 35) {
  alert("Aun sigues siendo joven");
}
else {
```

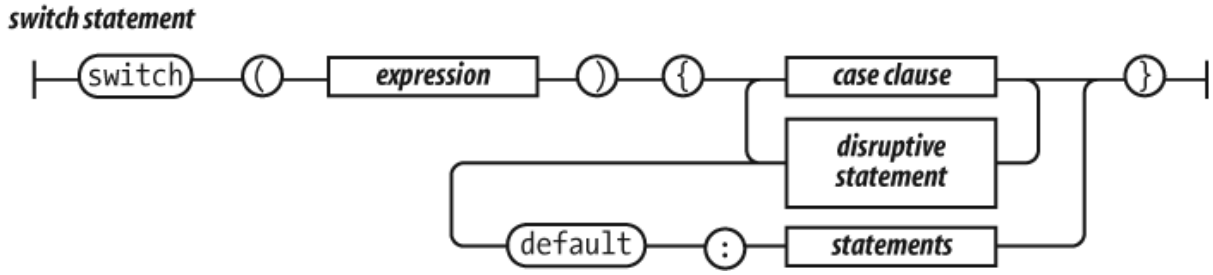
```
  alert("Piensa en cuidarte un poco más");
}
```

No es obligatorio que la combinación de estructuras `if...else` acabe con la instrucción `else`, ya que puede terminar con una instrucción de tipo `else if()`.

La estructura `switch` es muy útil cuando la condición que evaluamos puede tomar muchos valores. Si utilizasemos una sentencia `if...else`, tendríamos que repetir la condición para los distintos valores.

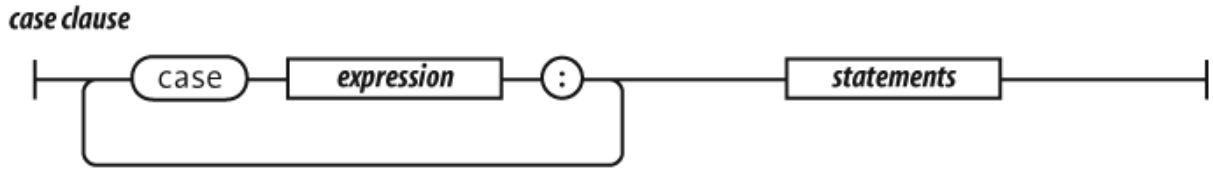
```
if(dia == 1) {
  alert("Hoy es lunes.");
}
else if(dia == 2) {
  alert("Hoy es martes.");
}
else if(dia == 3) {
  alert("Hoy es miércoles.");
}
else if(dia == 4) {
  alert("Hoy es jueves.");
}
else if(dia == 5) {
  alert("Hoy es viernes.");
}
else if(dia == 6) {
  alert("Hoy es sábado.");
}
else if(dia == 0) {
  alert("Hoy es domingo.");
}
```

En este caso es más conveniente utilizar una estructura de control de tipo `switch`, ya que permite ahorrarnos trabajo y producir un código más limpio. Su definición formal es la siguiente:

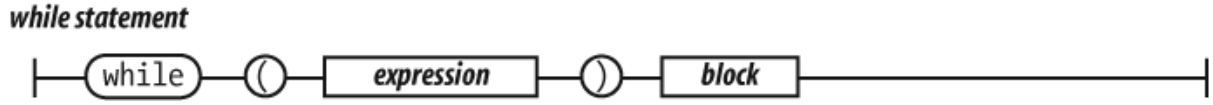



```
switch(dia) {
  case 1: alert("Hoy es lunes."); break;
  case 2: alert("Hoy es martes."); break;
  case 3: alert("Hoy es miércoles."); break;
  case 4: alert("Hoy es jueves."); break;
  case 5: alert("Hoy es viernes."); break;
  case 6: alert("Hoy es sábado."); break;
  case 0: alert("Hoy es domingo."); break;
}
```

La cláusula `case` no tiene por qué ser una constante, sino que puede ser una expresión al igual que en la estructura `if`. El comportamiento por defecto de la estructura `switch`es seguir evaluando el resto de cláusulas, aún cuando una de ellas haya cumplido la condición. Para evitar ese comportamiento, es necesario utilizar la sentencia `break` en las cláusulas que deseemos.



La estructura `while` ejecuta un simple bucle, mientras se cumpla la condición. Su definición formal es la siguiente:



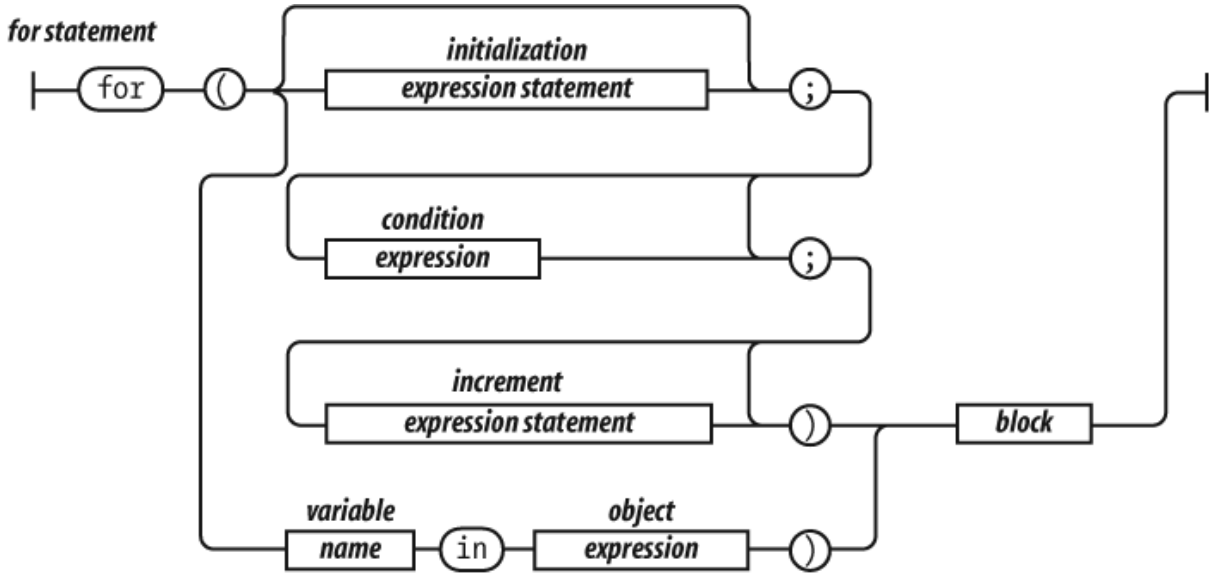

```
var veces = 0;

while(veces < 7) {
  alert("Mensaje " + veces);
  veces++;
}
```

La idea del funcionamiento de un bucle `while` es la siguiente: "mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del `while`. Es importante modificar los valores de las variables incluidas dentro de la condición, ya que otra manera, el bucle se repetiría de manera indefinida, perjudicando la ejecución de la página y bloqueando la ejecución del resto del script.

```
var veces = 0;

while(veces < 7) {
  alert("Mensaje " + veces);
  veces = 0;
}
```

En este ejemplo, se mostraría de manera infinita una alerta con el texto "Mensaje 0".

La estructura `for` permite realizar bucles de una forma muy sencilla. Su definición formal es la siguiente:




```
for(inicializacion; condicion; actualizacion) {
  ...
}
```

La idea del funcionamiento de un bucle `for` es la siguiente: *"mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del for. Además, después de cada repetición, actualiza el valor de las variables que se utilizan en la condición"*.

- La "inicialización" es la zona en la que se establece los valores iniciales de las variables que controlan la repetición.
- La "condición" es el único elemento que decide si continua o se detiene la repetición.
- La "actualización" es el nuevo valor que se asigna después de cada repetición a las variables que controlan la repetición. var mensaje = "Hola, estoy dentro de un bucle";

```
for(var i = 0; i < 5; i++) {
  alert(mensaje);
}
```

La parte de la inicialización del bucle consiste en:

```
var i = 0;
```

Por tanto, en primer lugar se crea la variable `i` y se le asigna el valor de `0`. Esta zona de inicialización solamente se tiene en consideración justo antes de comenzar a ejecutar el bucle. Las siguientes repeticiones no tienen en cuenta esta parte de inicialización.

La zona de condición del bucle es:

```
i < 5
```

Los bucles se siguen ejecutando mientras se cumplan las condiciones y se dejan de ejecutar justo después de comprobar que la condición no se cumple. En este caso, mientras la variable `i` valga menos de `5` el bucle se ejecuta indefinidamente.

Como la variable `i` se ha inicializado a un valor de `0` y la condición para salir del bucle es que `i` sea menor que `5`, si no se modifica el valor de `i` de alguna forma, el bucle se repetiría indefinidamente.

Por ese motivo, es imprescindible indicar la zona de actualización, en la que se modifica el valor de las variables que controlan el bucle:

```
i++
```

En este caso, el valor de la variable `i` se incrementa en una unidad después de cada repetición. La zona de actualización se ejecuta después de la ejecución de las instrucciones que incluye el `for`.

Así, durante la ejecución de la quinta repetición el valor de `i` será `4`. Después de la quinta ejecución, se actualiza el valor de `i`, que ahora valdrá `5`. Como la condición es que `i` sea menor que `5`, la condición ya no se cumple y las instrucciones del `for` no se ejecutan una sexta vez.

Normalmente, la variable que controla los bucles `for` se llama `i`, ya que recuerda a la palabra índice y su nombre tan corto ahorra mucho tiempo y espacio.

El ejemplo anterior que mostraba los días de la semana contenidos en un array se puede rehacer de forma más sencilla utilizando la estructura `for`:

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

for(var i=0; i<7; i++) {
  alert(dias[i]);
}
```

Una estructura de control derivada de `for` es la estructura `for...in`. Su definición exacta implica el uso de objetos, permitiendo recorrer las propiedades de un objeto. En cada iteración, un nuevo nombre de propiedad del objeto es asignada a la variable:

```
for(propiedad in object) {
  if (object.hasOwnProperty(propiedad)) {
    ...
  }
}
```

Suele ser conveniente comprobar que la propiedad pertenece efectivamente al objeto, a través de `object.hasOwnProperty(propiedad)`. De la misma manera que podemos recorrer las propiedades de un objeto, es posible adaptar este comportamiento a los arrays:

```
for(indice in array) {
  ...
}
```

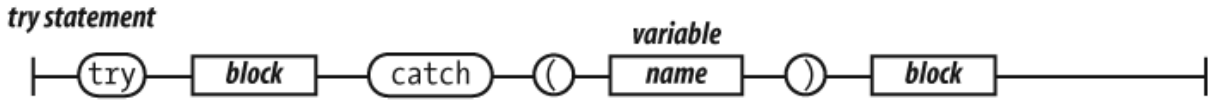
Si se quieren recorrer todos los elementos que forman un array, la estructura `for...in` es la forma más eficiente de hacerlo, como se muestra en el siguiente ejemplo:

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

for(i in dias) {
  alert(dias[i]);
}
```

La variable que se indica como indice es la que se puede utilizar dentro del bucle `for...in` para acceder a los elementos del array. De esta forma, en la primera repetición del bucle la variable `i` vale `0` y en la última vale `6`.

Esta estructura de control es la más adecuada para recorrer arrays (y objetos), ya que evita tener que indicar la inicialización y las condiciones del bucle `for` simple y funciona correctamente cualquiera que sea la longitud del array. De hecho, sigue funcionando igual aunque varíe el número de elementos del array.

La estructura `try` consiste en un bloque de código que se ejecuta de manera normal, y captura cualquier excepción que se pueda producir en ese bloque de sentencias. Su definición formal es la siguiente:




```
try {
    funcion_que_no_existe();
} catch(ex) {
    alert("Error detectado: " + ex.description);
}
```

En este ejemplo, llamamos a una función que no está definida, y por lo tanto provoca una excepción en JavaScript. Este error es *capturado* por la cláusula `catch`, que contiene una serie de sentencias que indican que acciones realizar con esa excepción que acaba de producirse. Si no se produce ninguna excepción en el bloque `try`, no se ejecuta el bloque dentro de `catch`.

La cláusula `finally` contiene las sentencias a ejecutar después de los bloques `try` y `catch`. Las sentencias incluidas en este bloque se ejecutan siempre, se haya producido una excepción o no. Un ejemplo clásico de utilización de la cláusula `finally`, es la de liberar recursos que el script ha solicitado.

```
abrirFichero()
try {
   escribirFichero(datos);
} catch(ex) {
    // Tratar la excepción
} finally {
   cerrarFichero(); // siempre se cierra el recurso
}
```

Ejercicio 5

Ver enunciado (#ej05)

Ejercicio 6

Ver enunciado (#ej06)

Ejercicio 7

Ver enunciado (#ej07)

Esta página se ha dejado vacía a propósito

Capítulo 6

JavaScript incorpora una serie de herramientas y utilidades (llamadas funciones y propiedades, como se verá más adelante) para el manejo de las variables. De esta forma, muchas de las operaciones básicas con las variables, se pueden realizar directamente con las utilidades que ofrece JavaScript.

A continuación se muestran algunas de las funciones más útiles para el manejo de cadenas de texto:

`length`, calcula la longitud de una cadena de texto (el número de caracteres que la forman)

```
var mensaje = "Hola Mundo";
var numeroLetras = mensaje.length; // numeroLetras = 10
```

+, se emplea para concatenar varias cadenas de texto

```
var mensaje1 = "Hola";
var mensaje2 = " Mundo";
var mensaje = mensaje1 + mensaje2; // mensaje = "Hola Mundo"
```

Además del operador +, también se puede utilizar la función `concat()`

```
var mensaje1 = "Hola";
var mensaje2 = mensaje1.concat(" Mundo"); // mensaje2 = "Hola Mundo"
```

Las cadenas de texto también se pueden unir con variables numéricas:

```
var variable1 = "Hola ";
var variable2 = 3;
var mensaje = variable1 + variable2; // mensaje = "Hola 3"
```

Cuando se unen varias cadenas de texto es habitual olvidar añadir un espacio de separación entre las palabras:

```
var mensaje1 = "Hola";
var mensaje2 = "Mundo";
var mensaje = mensaje1 + mensaje2; // mensaje = "HolaMundo"
```

Los espacios en blanco se pueden añadir al final o al principio de las cadenas y también se pueden indicar forma explícita:

```
var mensaje1 = "Hola";
var mensaje2 = "Mundo";
var mensaje = mensaje1 + " " + mensaje2; // mensaje = "Hola Mundo"
```

`toUpperCase()`, transforma todos los caracteres de la cadena a sus correspondientes caracteres en mayúsculas:

```
var mensaje1 = "Hola";
var mensaje2 = mensaje1.toUpperCase(); // mensaje2 = "HOLA"
```

`toLowerCase()`, transforma todos los caracteres de la cadena a sus correspondientes caracteres en minúsculas:

```
var mensaje1 = "HolA";
var mensaje2 = mensaje1.toLowerCase(); // mensaje2 = "hola"
```

`charAt(posicion)`, obtiene el carácter que se encuentra en la posición indicada:

```
var mensaje = "Hola";
var letra = mensaje.charAt(0); // letra = H
letra = mensaje.charAt(2);     // letra = l
```

`indexOf(caracter)`, calcula la posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si el carácter se incluye varias veces dentro de la cadena de texto, se devuelve su primera posición empe-

zando a buscar desde la izquierda. Si la cadena no contiene el carácter, la función devuelve el valor `-1`:

```
var mensaje = "Hola";
var posicion = mensaje.indexOf('a'); // posicion = 3
posicion = mensaje.indexOf('b');     // posicion = -1
```

Su función análoga es lastIndexOf():

`lastIndexOf(caracter)`, calcula la última posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si la cadena no contiene el carácter, la función devuelve el valor `-1`:

```
var mensaje = "Hola";
var posicion = mensaje.lastIndexOf('a'); // posicion = 3
posicion = mensaje.lastIndexOf('b');     // posicion = -1
```

La función `lastIndexOf()` comienza su búsqueda desde el final de la cadena hacia el principio, aunque la posición devuelta es la correcta empezando a contar desde el principio de la palabra.

`substring(inicio, final)`, extrae una porción de una cadena de texto. El segundo parámetro es opcional. Si sólo se indica el parámetro `inicio`, la función devuelve la parte de la cadena original correspondiente desde esa posición hasta el final:

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(2); // porcion = "la Mundo"
porcion = mensaje.substring(5);     // porcion = "Mundo"
porcion = mensaje.substring(7);     // porcion = "ndo"
```

Si se indica un `inicio` negativo, se devuelve la misma cadena original:

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(-2); // porcion = "Hola Mundo"
```

Cuando se indica el inicio y el final, se devuelve la parte de la cadena original comprendida entre la posición inicial y la inmediatamente anterior a la posición final (es decir, la posición `inicio` está incluida y la posición `final` no):

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(1, 8); // porcion = "ola Mun"
porcion = mensaje.substring(3, 4);     // porcion = "a"
```

Si se indica un `final` más pequeño que el `inicio`, JavaScript los considera de forma inversa, ya que automáticamente asigna el valor más pequeño al `inicio` y el más grande al `final`:

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(5, 0); // porcion = "Hola "
porcion = mensaje.substring(0, 5);     // porcion = "Hola "
```

`split(separador)`, convierte una cadena de texto en un array de cadenas de texto. La función parte la cadena de texto determinando sus trozos a partir del carácter `separador` indicado:

```
var mensaje = "Hola Mundo, soy una cadena de texto!";
var palabras = mensaje.split(" ");
// palabras = ["Hola", "Mundo,", "soy", "una", "cadena", "de", "texto!"];
```

Con esta función se pueden extraer fácilmente las letras que forman una palabra:

```
var palabra = "Hola";
var letras = palabra.split(""); // letras = ["H", "o", "l", "a"]
```

A continuación se muestran algunas de las funciones más útiles para el manejo de arrays:

`length`, calcula el número de elementos de un array

```
var vocales = ["a", "e", "i", "o", "u"];
var numeroVocales = vocales.length; // numeroVocales = 5
```

`concat()`, se emplea para concatenar los elementos de varios arrays

```
var array1 = [1, 2, 3];
array2 = array1.concat(4, 5, 6);   // array2 = [1, 2, 3, 4, 5, 6]
array3 = array1.concat([4, 5, 6]); // array3 = [1, 2, 3, 4, 5, 6]
```

`join(separador)`, es la función contraria a `split()`. Une todos los elementos de un array para formar una cadena de texto. Para unir los elementos se utiliza el carácter `separador` indicado

```
var array = ["hola", "mundo"];
var mensaje = array.join(""); // mensaje = "holamundo"
mensaje = array.join(" ");    // mensaje = "hola mundo"
```

`pop()`, elimina el último elemento del array y lo devuelve. El array original se modifica y su longitud disminuye en 1 elemento.

```
var array = [1, 2, 3];
var ultimo = array.pop();
// ahora array = [1, 2], ultimo = 3
```

`push()`, añade un elemento al final del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];
array.push(4);
// ahora array = [1, 2, 3, 4]
```

`shift()`, elimina el primer elemento del array y lo devuelve. El array original se ve modificado y su longitud disminuida en 1 elemento.

```
var array = [1, 2, 3];
var primero = array.shift();
// ahora array = [2, 3], primero = 1
```

`unshift()`, añade un elemento al principio del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];
array.unshift(0);
// ahora array = [0, 1, 2, 3]
```

`reverse()`, modifica un array colocando sus elementos en el orden inverso a su posición original:

```
var array = [1, 2, 3];
array.reverse();
// ahora array = [3, 2, 1]
```

A continuación se muestran algunas de las funciones y propiedades más útiles para el manejo de números.

`NaN`, (del inglés, "Not a Number") JavaScript emplea el valor `NaN` para indicar un valor numérico no definido (por ejemplo, la división `0/0`).

```
var numero1 = 0;
var numero2 = 0;
alert(numero1/numero2); // se muestra el valor NaN
```

`isNaN()`, permite proteger a la aplicación de posibles valores numéricos no definidos

```
var numero1 = 0;
var numero2 = 0;
if(isNaN(numero1/numero2)) {
  alert("La división no está definida para los números indicados");
}
else {
  alert("La división es igual a => " + numero1/numero2);
}
```

`Infinity`, hace referencia a un valor numérico infinito y positivo (también existe el valor –`Infinity` para los infinitos negativos)

```
var numero1 = 10;
var numero2 = 0;
alert(numero1/numero2); // se muestra el valor Infinity
```

`toFixed(digitos)`, devuelve el número original con tantos decimales como los indicados por el parámetro `digitos` y realiza los redondeos necesarios. Se trata de una función muy útil por ejemplo para mostrar precios.

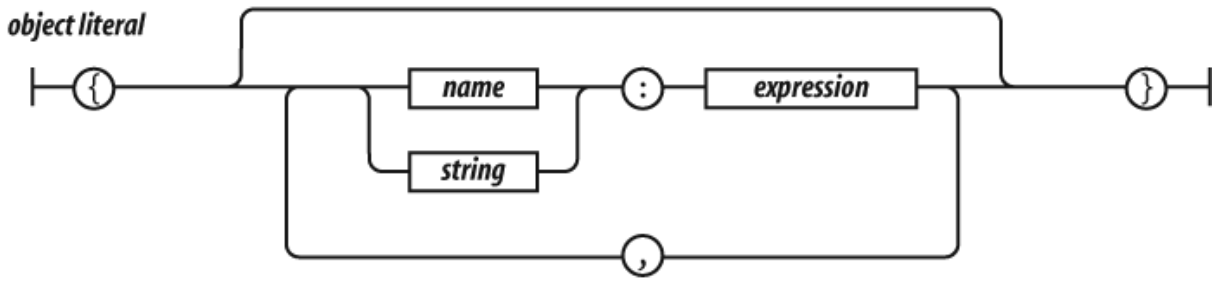
```
var numero1 = 4564.34567;
numero1.toFixed(2); // 4564.35
numero1.toFixed(6); // 4564.345670
numero1.toFixed(); // 4564
```

Capítulo 7

Los tipos primitivos en JavaScript son `undefined`, `null`, `boolean`, `number` y `string`. Aunque pueda resultar extraño, el resto de elementos en JavaScript son objetos, tanto las funciones, arrays, expresiones regulares como los propios objetos. Un objeto en JavaScript es un contenedor de propiedades, donde una propiedad tiene un nombre y un valor. El nombre de una propiedad puede ser una cadena de caracteres, incluso una vacía. El valor de la propiedad puede ser cualquier valor que podamos utilizar en JavaScript, excepto `undefined`.

Los objetos en JavaScript no dependen de una clase, como lo hacen en otros lenguajes de programación. No hay ninguna restricción en el nombre de las nuevas propiedades ni en el de sus valores, simplemente organizan datos. Por supuesto, los objetos pueden contener otros objetos.

JavaScript ofrecen una manera muy cómoda para crear y representar objetos. Una representación de un objeto consiste en una pareja de llaves (`{ }`) que contienen cero o más pares de clave/valor, de la siguiente manera:



```
var empty_object = {};

var obj = {
    "first-name": "Arkaitz",
    "last-name": "Garro"
};
```

Como hemos dicho, el nombre de una propiedad puede ser un `string`, incluso uno vacío. Las *comillas dobles* alrededor del nombre de la propiedad son opcionales, si el nombre es un nombre legal de JavaScript, y no es una palabra reservada. Así, las *comillas dobles* son necesarias para el nombre "first-name", pero no son necesarias para first_name. Los valores de las propiedades pueden ser cualquier expresión, incluso la definición de otro objeto:

```
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

Los valores de un objeto pueden ser accedidos indicando el nombre de la propiedad dentro de unos corchetes (`[ ]`). Si el nombre de la propiedad es un nombre legal de JavaScript, y no es una palabra reservada, se puede utilizar la notación `.`. Es preferible utilizar la notación `.`, ya que es más corta y comunmente utilizada como acceso a métodos y propiedades en lenguajes orientados a objetos:

```
stooge["first-name"]    // "Arkaitz"
flight.departure.IATA   // "SYD"
```

El valor `undefined` es devuelto si se intenta acceder a una propiedad que no existe:

```
stooge["middle-name"]   // undefined
flight.status           // undefined
stooge["FIRST-NAME"]    // undefined
```

El operador `||` puede ser utilizado para obtener valores por defecto:

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

Intentar acceder a los valores de una propiedad no definida, lanzará una excepción de tipo `TypeError`. Esto puede evitarse utilizando el operador `&&`, para asegurarnos que el valor existe y es accesible:

```
flight.equipment                              // undefined
flight.equipment.model                        // undefined
flight.equipment && flight.equipment.model    // undefined
```

El valor de una objeto puede actualizarse a través de una asignación. Si el nombre de la propiedad existe en el objeto, su valor es reemplazado:

```
stooge['first-name'] = 'Arkaitz';
```

Si la propiedad no existe en el objeto, esta nueva propiedad es añadida al objeto:

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = { model: 'Boeing 777' }
flight.status = 'overdue';
```

Los objetos siempre son accedidos como referencias, nunca se copia su valor cuando los asignamos a otros objetos, o los pasamos como parámetros en funciones:

```
var x = stooge;
x.nickname = 'Curly';

// nick contiene 'Curly' porque x y stooge
// referencian al mismo objeto
var nick = stooge.nickname;

// a, b, y c hacen referencia a
```

```
// diferentes objetos vacíos
var a = {}, b = {}, c = {};

// a, b, y c hacen referencia al
// mismo objeto vacío
a = b = c = {};

var obj { value = 5 };
alert(obj.value); // o.value = 5

function change(obj)
{
    obj.value = 6;
}

change(obj);
alert(obj.value); // o.value = 6
```

Todos los objetos de JavaScript enlazan con un objeto prototipo del que heredan todas sus propiedades. Los objetos creado a través de literales, están enlazados con `Object.prototype`, un objeto estándar incluido en JavaScript.

Cuando creamos un objeto nuevo, tenemos la posibilidad de seleccionar cuál será su prototipo. El mecanismo que JavaScript proporciona para hacer esto es desordenado y complejo, pero se puede simplificar de manera significativa. Vamos a añadir un método de creación a nuestro objeto. El método `create` crea un nuevo objeto que utiliza un objeto antiguo como su prototipo.

```
// Shape - superclass
function Shape() {
    this.x = 0;
    this.y = 0;
}

Shape.prototype.move = function(x, y) {
    this.x += x;
    this.y += y;
    console.info("Shape moved.");
};

// Rectangle - subclass
```

```
function Rectangle() {
    Shape.call(this); //call super constructor.
}

Rectangle.prototype = Object.create(Shape.prototype);

var rect = new Rectangle();

rect instanceof Rectangle   // true.
rect instanceof Shape       // true.

rect.move(); // Outputs, "Shape moved."
```

Para los navegadores que no soportan la función `create`, podemos extender el objeto de JavaSript `Object` para incluir esta funcionalidad:

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
var another_stooge = Object.create(stooge);
```

El prototipo enlazado no se ve afectado por las modificaciones. Si realizamos cambios en un objeto, el objeto prototipo no se ve afectado.

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

El enlace de los prototipos es únicamente utilizado cuando accedemos a los datos. Si intentamos acceder al valor de una propiedad, y esa propiedad no existe en el objeto, entonces JavaScript va a intentar obtener ese valor del prototipo del objeto. Y si ese objeto tampoco dispone de la propiedad, lo intentará obtener de sucesivos prototiposm, hasta que finalmente se encuentre con `Object.prototype`. Si la propidad no existe en ninguno de los prototipos, entonces el valor devuelto es `undefined`.

La relación de prototipos es dinámica. Si nosotros añadimos una nueva propiedad a un prototipo, entonces esta propiedad estará inmediatamente accesible para el resto de prototipos que estén basados en ese prototipo:

```
stooge.profession = 'actor';
another_stooge.profession       // 'actor'
```

La sentencia `for...in` puede iterar sobre todos los nombres de propiedades de un objeto. Esta iteración incluirá todas las propiedades, funciones y propiedades definidas en los prototipos, en las que no podemos estar interesados. La mejor manera de filtrar estas pripiedades es a través de la función `hasOwnProperty` y el operador `typeof`:

```
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

No hay ningún tipo de garantía en el orden en el que se van a mostra las priopiedades, por lo que si esto es importante, tendremos que controlarlo de alguna manera. Para ello, la mejor manera es olvidarnos de la sentencia `for...in` y acceder directamente a las propiedades concretas, en el orden que definamos:

```
var i;
var properties = [
    'first-name',
    'middle-name',
    'last-name',
    'profession'
];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' +
        another_stooge[properties[i]]);
}
```

El operado `delete` puede ser utilizado para eliminar la propiedad de un objeto. Este operador eliminará la propiedad de un objeto, si la tuviera, pero no afectará al resto de propiedades de los prototipos.

```
another_stooge.nickname // 'Moe'

// Remove nickname from another_stooge, revealing
```

```
// the nickname of the prototype.
delete another_stooge.nickname;

another_stooge.nickname // 'Curly'
```

JavaScript permite crear variables globales de una manera muy sencilla, Desafortunadamente, las variables globales perjudican directamente la calidad de los programas, por lo que deben ser evitadas. Una manera de minimizar la utilización de variables globales, es crear una única variable global para toda la aplicación, que incluya el resto de variables:

```
var MYAPP = {};
```

En este momento, esta variable se convierte en la contenedora de todas las variables de la aplicación:

```
MYAPP.stooge = {
    "first-name": "Joe",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

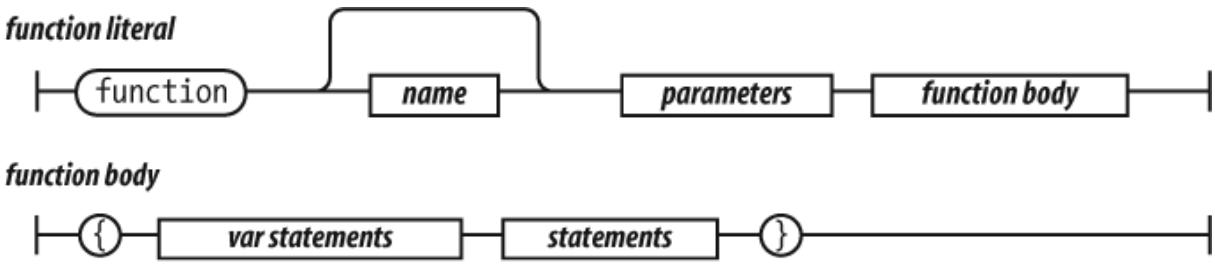
Reduciendo el número de variables globales a uno, se reduce de manera significativa la posibilidad de colisiones con otras aplicaciones, widgtes o librerías. Además, la aplicación puede leerse y entenderse de manera más sencilla.

Esta página se ha dejado vacía a propósito

Capítulo 8

Las funciones son la base de la modularidad en JavaScript. Son utilizadas para reutilizar código, ocultar información y abstracción. Por norma general, las funciones son utilizadas para especificar el comportamiento de los objetos, aunque pueden definirse funciones al margen de los objetos.

Las funciones en JavaScript se definen mediante la palabra reservada `function`, seguida del nombre de la función. Su definición formal es la siguiente:




```
function nombre_funcion() {
  ...
}
```

El nombre de la función se utiliza para *llamar* a esa función cuando sea necesario. El concepto es el mismo que con las variables, a las que se les asigna un nombre único para poder utilizarlas dentro del código. Después del nombre de la función, se incluyen dos paréntesis donde indicaremos los parámetros de la función. Por último, los símbolos `{` y `}` se utilizan para encerrar todas las instrucciones que pertenecen a la función.

Las funciones en Javascript son objetos. Las funciones son objetos enlazados con `Function.prototype` (que a su vez enlaza con `Object.prototype`), que incluyen dos propiedades ocultas: el contexto de la función y el código que implementa su comportamiento.

Toda función en JavaScript es creada con una propiedad `protoype`. Su valor es un objeto con una propiedad *constructor* cuyo valor es la propia función. Esto es diferente al enlace oculto a `Function.prototype`. Veremos el significado de esta complicada construcción más adelante. Como las funciones son objetos, pueden ser utilizadas como cualquier otro valor. Pueden ser almacenadas en variables, objetos o arrays. Pueden ser pasadas como argumentos a otras funciones, y pueden ser retornadas por otras funciones. Además, al ser objetos, también pueden tener métodos. Lo que las hace realmente especiales es que pueden ser llamadas.

Las funciones más sencillas no necesitan ninguna información para producir sus resultados, aunque lo normal es que necesiten de datos para producir resultados. Las variables que necesitan las funciones se llaman *argumentos*. Antes de que pueda utilizarlos, la función debe indicar cuántos argumentos necesita y cuál es el nombre de cada argumento. Además, al llamar a la función, se deben incluir los valores (o expresiones) que se le van a pasar a la función. Los argumentos se indican dentro de los paréntesis que van detrás del nombre de la función y se separan con una coma (,).

```
var s = function suma_y_muestra(n1, n2) { ... }
```

A continuación, para utilizar el valor de los argumentos dentro de la función, se debe emplear el mismo nombre con el que se definieron los argumentos:

```
var s = function suma_y_muestra(n1, n2) {
  var resultado = n1 + n2;
  alert("El resultado es " + resultado);
}
```

Dentro de la función, el valor de la variable `n1` será igual al primer valor que se le pase a la función y el valor de la variable `n2` será igual al segundo valor que se le pasa.

Las funciones no solamente puede recibir variables y datos, sino que también pueden devolver los valores que han calculado. Para devolver valores dentro de una función, se utiliza la palabra reservada `return`. Aunque las funciones pueden devolver valores de cualquier tipo, solamente pueden devolver un valor cada vez que se ejecutan.

```
var c = function (precio) {
  var impuestos = 1.21;
  var gastosEnvio = 10;
  var precioTotal = ( precio * impuestos ) + gastosEnvio;

  return precioTotal;
}
```

Para que la función devuelva un valor, solamente es necesario escribir la palabra reservada `return` junto con el nombre de la variable que se quiere devolver. En el ejemplo anterior, la ejecución de la función llega a la instrucción `return precioTotal;` y en ese momento, devuelve el valor que contenga la variable `precioTotal`.

Como la función devuelve un valor, en el punto en el que se realiza la llamada, debe indicarse el nombre de una variable en el que se guarda el valor devuelto:

```
var precioTotal = c(23.34);
```

Si no se indica el nombre de ninguna variable, JavaScript no muestra ningún error y el valor devuelto por la función simplemente se pierde y por tanto, no se utilizará en el resto del programa. Si la función llega a una instrucción de tipo `return`, se devuelve el valor indicado y finaliza la ejecución de la función. Por tanto, todas las instrucciones que se incluyen después de un `return` se ignoran y por ese motivo la instrucción `return` suele ser la última de la mayoría de funciones.

Llamar a una función suspende la ejecución de la función actual, pasando el control y los parámetros a la nueva función. Además de los argumentos declarados, todas las funciones reciben dos argumentos extra: `this` y `arguments`. El parámetro `this` es muy importante la programación orientada a objetos, y su valor viene determinado por el patrón de llamada utilizado. Existen cuatro patrones de llamada en JavaScript: el patrón

de llamada `method`, el patrón de llamada `function`, el patrón de llamada `constructor` y el patrón de llamada `apply`.

El parámetro extra disponible en las funciones es el array `arguments`. Da a la función acceso a todas los argumentos pasados en la llamada, incluidos los argumentos extra que no coinciden con los parámetros definidos en la función. Esto hace posible escribir funciones que toman un número indefinido de parámetros.

```
var sum = function() {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};

document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

Debido a un problema de diseño, `arguments` no es realmente un array, sino un objeto que se comporta como un array. Dispone de la propiedad `length`, pero no incluye el resto de métodos de los arrays.

Un patrón común en JavaScript son las funciones anónimas autoejecutables. Este patrón consiste en crear una expresión de función e inmediatamente ejecutarla.

```
(function(){
  var foo = 'Hola mundo';
})();

console.log(foo);   // indefinido (undefined)
```

En JavaScript, las funciones pueden ser asignadas a variables o pasadas a otras funciones como argumentos. En *frameworks* como jQuery por ejemplo, pasar funciones como argumentos es una práctica muy común.

```
var myFn = function(fn) {
  var result = fn();
  console.log(result);
```

```
};

myFn(function() { return 'hola mundo'; });    // muestra en la consola
'hola mundo'

var myOtherFn = function() {
    return 'hola mundo';
};

myFn(myOtherFn);    // muestra en la consola 'hola mundo'
```

El alcance en un lenguaje de programación controla la visibilidad y el ciclo de vida de las variables y los parámetros. Por ejemplo:

```
var foo = function () {
    var a = 3, b = 5;
    var bar = function () {
        var b = 7, c = 11;
        // En este punto, a es 3, b es 7, y c es 11

        a += b + c;
        // En este punto, a es 21, b es 7, y c es 11
    };

    // En este punto, a es 3, b es 5, y c es undefined

    bar();

    // En este punto, a es 21, b es 5
};
```

La mayoría de los lenguajes, con sintaxis de C, tienen un alcance de bloque. Todas las variables definidas en un bloque (sentencias definidas entre llaves) no son visibles fuera de ese bloque. Desafortunadamente, JavaScript no tiene esa visibilidad de bloque, a pesar de que su sintaxis así pueda sugerirlo, y esto puede ser una fuente de problemas. JavaScript tiene un alcance de función: esto quiere decir que los parámetros y variables definidos dentro de una función no son visibles fuera de esa función, y que una variable definida en cualquier lugar de la función, es visible desde cualquier lugar dentro de esa función.

La buena noticia sobre la visibilidad, es que las funciones internas pueden tener acceso a los parámetros y variables de las funciones donde han sido definidas (con la excepción de `this` y `arguments`). Antes hemos definido un objeto que tenía una propieda `value` y un método `increment`. Supongamos que queremos proteger ese valor cambios no autorizados. En lugar de inicializar ese objeto como un literal, vamos a inicializarlo llamando a una función que devuelva un objeto literal. Vamos a definir una variable `value` y un objeto de retorno con los métodos `increment` y `getValue`:

```
var myObject = (function () {
    var value = 0;

    return {
        increment: function (inc) {
            value += typeof inc === 'number' ? inc : 1;
        },
        getValue: function () {
            return value;
        }
    };
}());
```

En este caso, tanto el método `increment` como `getValue` tienen acceso a la variable `value`, pero debido a su alcance de función, el resto de la aplicación no tiene acceso a su valor.

Las funciones pueden facilitar el trabajo con métodos asíncronos. Supongamos el siguiente caso en el que hacemos una petición al servidor:

```
request = prepare_the_request();
response = send_request_synchronously(request);
display(response);
```

El problema aquí, es que esperamos la respuesta del servidor, por lo que bloqueamos la ejecución del script hasta obtener una respuesta. Una estrategia mucho mejor es realizar una llamada asíncrona, proporcionando una función de respuesta (`callback`) que se ejecutará cuando la respuesta esté disponible:

```
request = prepare_the_request();
send_request_asynchronously(request, function (response) {
    display(response);
});
```

Algunos métodos no devuelven ningún tipo de dato, ya simplemente pueden modificar el estado de algún tipo de dato. Si hacemos que estos métodos devuelvan `this` en lugar de `undefined`, podemos encadenar la ejecución de esos métodos en cascada. Las funciones en cascada pueden permitirnos escribir sentencias de este tipo:

```
getElement('myBoxDiv')
    .move(350, 150)
    .width(100)
    .height(100)
    .color('red')
    .border('10px outset')
    .padding('4px')
    .appendText("Please stand by")
    .on('mousedown', function (m) {
        this.startDrag(m, this.getNinth(m));
    })
    .on('mousemove', 'drag')
    .on('mouseup', 'stopDrag')
    .tip('This box is resizeable');
```

Ejercicio 8

Ver enunciado (#ej08)

Ejercicio 9

Ver enunciado (#ej09)

Ejercicio 10

Ver enunciado (#ej010)

Esta página se ha dejado vacía a propósito

Capítulo 9

JavaScript en un lenguaje orientado a objetos basado en prototipos, en lugar de estar basado en clases. Debido a esta básica diferencia, es menos evidente entender cómo JavaScript nos permite crear herencia entre objetos, y heredar las propiedades y sus valores.

Los lenguajes orientados a objetos basados en clases como Java o C++, se basan en el concepto de dos entidades distintas: la clase y las instancias.

- Una *clase* define todas las propiedades que caracteriza a una serie de objetos. La *clase* es algo abstracto, no como las instancias de los objetos que describe. Por ejemplo, una clase `Empleado`, puede representar un conjunto concreto de empleados.
- Una *instancia*, en cambio, es una representación concreta de esa *clase*. Por ejemplo, *Victoria* puede ser una instancia concreta de la *clase* `Empleado`, es decir, representa de manera concreta a un empleado. Una instancia tiene exactamente las mismas propiedades que la clase padre (Ni más, ni menos).

Un lenguaje basado en prototipos, como JavaScript, no hace esta distinción: simplemente maneja objetos. Este tipo de lenguajes tiene la noción de objetos *prototipo*, objetos usados como platilla para obtener las propiedades iniciales de un objeto. Cualquier objeto puede especificar su propias propieades, tanto en el momento que los creamos como en

tiempo de ejecución. Además, cualquier objeto puede asociarse como prototipo a otro objeto, permitiendo compartir todas sus propiedades.

Un lenguaje basado en clases, definimos la clase de manera independiente. En esta definición, especificamos los constructores, que son utilizamos para crear las instancias de las clases. Un método constructor, puede especificar los valores iniciales de una instancia de una clase, y realizar las operaciones necesarias a la hora de crear el objeto. Utilizamos el operador `new`, conjuntamente con el nombre del constructor, para crear nuevas instancias.

JavaScript sigue un modelo similar, pero no separa la definición de las propiedades del constructor. En este caso, definimos una función *constructora* para crear los objetos con un conjunto inicial de propiedades y valores. Cualquier función de JavaScript puede ser utilizada como constructor. Utilizamos el operador `new`, conjuntamente con el nombre de la función *constructora*, para crear nuevas instancias.

En un lenguaje basado en clases, es posible crear estructura de clases a través de su definición. En esta definición, podemos especificar que la nueva clase es una subclase de una clase que ya existe. Esta subclase, hereda todas las propiedades de la superclase, y además puede añadir o modificar las propiedades heredadas.

JavaScript implementa una herencia que nos permite asociar un objeto prototipo con una función constructora. De esta manera, el nuevo objeto hereda todas las propiedades del objeto prototipo.

La siguiente tabla muestra un pequeño resumen de las diferencias entre un lenguaje basado en clases, como Java, y un lenguaje basado en prototipos, como JavaScript.

Clase e instancia son dos entidades diferentes	Todos los objetos son instancias
Las clases se definen de manera explícita, y se	Las clases se definen y crean con las funciones constructoras.

instancias a través de su método constructor.	
Un objeto se instancia con el operador `new`.	Un objeto se instancia con el operador `new`.
La estructura de clases se crea utilizando la definición de clases.	La estructura de clases se crea asignando un objeto como prototipo.
La herencia de propiedades se realiza a través de la cadena de clases.	La herencia de propiedades se realiza a través de la cadena de prototipos.
La definición de clases especifica todas las propiedades de una instancia de una clase. No se pueden añadir propiedades en tiempo de ejecución.	La función constructora o el prototipo especifican unas propiedades iniciales. Se pueden añadir o eliminar estas propiedades en tiempo de ejecución, en un objeto concreto o a un conjunto de objetos.

Veamos como se implementa esta herencia en JavaScript, a través de un simple ejemplo. Queremos implementar la siguiente estructura:

- Un `Empleado` se define con las propiedades `nombre` (cuyo valor por defecto es una cadena vacía), y un `departamento` (cuyo valor por defecto es "General").
- Un `Director` está basado en `Empleado`. Añade la propiedad `informes` (cuyo valor por defecto es un array vacío).
- Un `Trabajador` está basado también en `Empleado`. Añade la propiedad `proyectos` (cuyo valor por defecto es un array vacío).
- Un `Ingeniero` está basado en `Trabajador`. Añade la propiedad `maquina` (cuyo valor por defecto es una cadena vacía) y sobreescribe la propiedad `departamento` con el valor "Ingeniería".

Codificación en JavaScript:

```
function Empleado (nombre, departamento) {
  this.nombre = nombre || "";
  this.departamento = departamento || "General";
}

function Director (nombre, departamento, informes) {
  this.base = Empleado;
  this.base(nombre, departamento);
  this.informes = informes || [];
}
Director.prototype = new Empleado;

function Obrero (nombre, departamento, proyectos) {
  this.base = Empleado;
  this.base(nombre, departamento);
  this.proyectos = proyectos || [];
}
Obrero.prototype = new Empleado;

function Ingeniero (nombre, proyectos, maquina) {
  this.base = Obrero;
  this.base(nombre, "Ingeniería", proyectos);
  this.maquina = maquina || "";
}
Ingeniero.prototype = new Obrero;
```

Supongamos que queremos crear un nuevo `Ingeniero` de la siguiente manera:

```
var arkaitz = new Ingeniero("Garro, Arkaitz",
                            ["xhtml", "javascript", "html5"],
                            "Chrome");
```

JavaScript sigue el siguiente proceso:

1. El operador `new` creará un objeto genérico y asignará a la propiedad `__proto__` el valor de `Ingeniero.prototype`.
2. El operador `new` pasará el nuevo objeto creado al constructor de `Ingeniero`, como valor de la palabra reservada `this`.
3. El constructor crea una nueva propiedad llamada `base` para este objeto, y le asigna el valor del constructor `Obrero`. Esto convierte el constructor de `Obrero`en un método del objeto `Ingeniero`. El nombre

`base`no hace referencia a ninguna palabra reservada, es simplemente para hacer referencia al padre.

4. El constructor llama al método `base`, pasando como argumentos dos de sus argumentos (`"Garro, Arkaitz"` y `["xhtml", "javascript", "html5"]`), además de la cadena de caracteres "Ingeniería". Indicar este parámetro fijo, hace que todos los objetos creados de tipo `Ingeniero`, tengan el mismo valor para la propiedad `departamento`, sobreescribiendo la el valor original de `Empleado`.
5. Al llamar al método `base`, JavaScript asocia la palabra reservada `this` al objeto creado en el paso 1. En consecuencia, la función `Obrero` pasa los valores `"Garro, Arkaitz"` y `["xhtml", "javascript", "html5"]` al constructor de `Empleado`. Cuando se completa este paso, la función `Obrero` asigna el valor de los projectos a su propiedad.
6. Una vez finalizado el método `base`, el constructor de `Ingeniero` asigna el valor `Chrome` a la propiedad `maquina`.
7. Una vez finalizado el constructor, JavaScript asigna el nuevo objeto a la variable `arkaitz`.

La búsqueda de propiedades en JavaScript comienza en las propias propiedades del objeto, y si este nombre de propiedad no se encuentra, consulta las propiedades del objeto especial `__proto__`. Este proceso se realiza de manera recursiva.

La propiedad especial `__proto__` se define cuando un objeto es construido: su valor corresponde con la propiedad `prototype` del constructor. Así, la expresión `new Foo()` crea un objeto con la propiedad `__proto__ == Foo.prototype`. En consecuencia, los cambios producidos en `Foo.prototype` alteran la búsqueda de propiedades de todos los objetos creados con `new Foo()`.

Todo objeto tiene una propiedad `__proto__`, así como una propiedad `prototype`. Por lo tanto, los objetos pueden relacionarse a través de esta propiedad. Un ejemplo:

```
var arkaitz = new Ingeniero("Garro, Arkaitz",
                            ["xhtml", "javascript", "html5"],
                            "Chrome");
```

En este objeto, todas las siguientes sentencias se cumplen:

```
arkaitz.__proto__ == Ingeniero.prototype;
arkaitz.__proto__.__proto__ == Obrero.prototype;
arkaitz.__proto__.__proto__.__proto__ == Empleado.prototype;
arkaitz.__proto__.__proto__.__proto__.__proto__ == Object.prototype;
arkaitz.__proto__.__proto__.__proto__.__proto__.__proto__ == null;
```

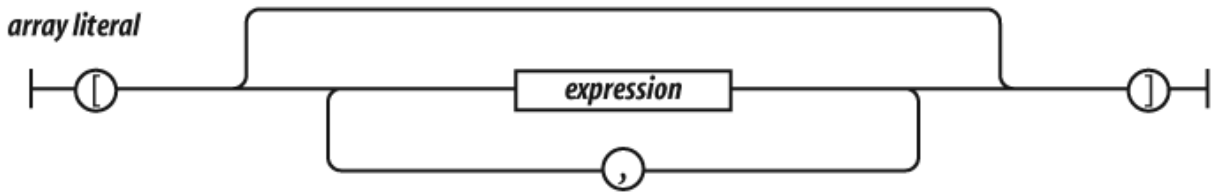
De esta manera, podemos counstruir una función apara saber si un objeto es una variable es una instancia de una objeto.

```
function instanceOf(object, constructor) {
   while (object != null) {
      if (object == constructor.prototype)
         return true;
      if (typeof object == 'xml') {
        return constructor.prototype == XML.prototype;
      }
      object = object.__proto__;
   }
   return false;
}

instanceOf (arkaitz, Engineer)      // true
instanceOf (arkaitz, WorkerBee)     // true
instanceOf (arkaitz, Employee)      // true
instanceOf (arkaitz, Object)        // true

instanceOf (arkaitz, SalesPerson)   // false
```

Capítulo 10

Un array es una asignación lineal de memoria donde los elementos son accedidos a través de índices numéricos, siendo además una estructura de datos muy rápida. Desafortunadamente, JavaScript no utiliza este tipo de arrays. En su lugar, JavaScript ofrece un objeto que dispone de características que le hacen parecer un array. Internamente, convierte los índices del array en `strings` que son utilizados como nombres de propiedades, haciéndolo sensiblemente más lento que un array.

JavaScript ofrecen una manera muy cómoda para crear y representar arrays. Una representación de un array consiste en una pareja de corchetes (`[ ]`) que contienen cero o más expresiones. El primer valor recibe la propiedad de nombre '0', la segunda la propiedad de nombre '1', y así sucesivamente. Se define de la siguiente manera:



Algunos ejemplos de declaración de arrays:

```
var empty = [];

var numbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'eight', 'nine'
```

```
];

empty[1]        // undefined
numbers[1]      // 'one'

empty.length    // 0
numbers.length  // 10
```

Si representasemos el array como un objeto

```
var numbers_object = {
    '0': 'zero', '1': 'one',    '2': 'two',
    '3': 'three', '4': 'four', '5': 'five',
    '6': 'six', '7': 'seven', '8': 'eight',
    '9': 'nine'
];
```

se produce un efecto similar. Ambas representaciones contienen 10 propiedades, y ests propiedades tienen exactamente los mismos nombres y valores. La diferencia radica en que `numbers` hereda de `Array.prototype`, mientras que `numbers_object` lo hace de `Object.prototype`, por lo que `numbers` hereda una serie de métodos que *convierten* a `numbers` en un array.

En la mayoría de los lenguajes de programación, se requiere que todos los elementos de un array sean del mismo tipo, pero en JavaScript eso no ocurre. Un array puede contener una mezcla valores:

```
var misc = [
    'string', 98.6, true, false, null, undefined,
    ['nested', 'array'], {object: true}, NaN,
    Infinity
];
misc.length // 10
```

Todo array tiene una propiedad `lenth`. A diferencia de otros lenguajes, la longitud del array no es fija, y podemos añadir elementos de manera dinámica. Esto hace que la propiedad `length` varíe, y tenga en cuenta los nuevos elementos. La propiedad `length` hace referencia al mayor índice presente en el array, más uno. Esto es:

```
var myArray = [];
myArray.length              // 0
```

```
myArray[1000000] = true;
myArray.length              // 1000001

// myArray contiene un elemento!
```

La propiedad `length` puede indicarse de manera explícita. Aumentando su valor, no vamos a reservar más espacio para el array, pero si disminuimos su valor, haciendo que sea menor que el número de elementos del array, eliminará los elementos cuyo índice sea mayor que el nuevo `length`:

```
numbers.length = 3; // numbers es ['zero', 'one', 'two']
```

Como los arrays de JavaScript son realmente objetos, podemos utilizar el operador `delete` para eliminar elementos de un array:

```
delete numbers[2];
// numbers es ['zero', 'one', undefined, 'shi', 'go']
```

Desafortunadamente, esto deja un espacio en el array. Esto es porque los elementos a la derecha del elemento eliminado conservan sus nombres. Para este caso, JavaScript incorpora una función `splice`, que permite eliminar y reemplazar elementos de un array. El primer argumento indica el por qué elemento comenzar a reemplazar, y el segundo argumento el número de elementos a eliminar.

```
numbers.splice(2, 1);
// numbers es ['zero', 'one', 'shi', 'go']
```

Unfortunately, that leaves a hole in the array. This is because the elements to the right of the deleted element retain their original names. What you usually want is to decrement the names of each of the elements to the right.

Esta página se ha dejado vacía a propósito

Capítulo 11

Las expresiones, son un lenguaje utilizado para describir patrones en cadenas de caracteres. Forman un pequeño y separado lenguaje, que está incluido en JavaScript (y en la gran mayoría de lenguajes de programación). No es un lenguaje fácil de leer, pero es una herramienta muy poderosa que simplifica mucho tareas de procesado de cadenas de caracteres.

Las expresiones regulares son utilizadas para buscar, reemplazar y extraer información de las cadenas de caracteres. Al igual que la gran mayoría de elementos en JavaScript, las expresiones regulares también son objetos. Los métodos que funcionan con expresiones regulares en JavaScript son las siguientes: `regexp.exec`, `regexp.test`, `string.match`, `string.replace`, `string.search`, y `string.split`. Las expresiones regulares tienen un rendimiento sensiblemente superior a las operaciones equivalentes sobre *strings*.

Al igual que las cadenas de caracteres se escriben entre comillas dobles (″), los patrones de expresiones regulares se escriben entre barras (/).

```
var slash = /\//;
alert("AC/DC".search(slash));
```

Este método de búsqueda se asemeja a `indexOf`, pero utiliza una expresión regular para la búsqueda en lugar de un *string*. Los patrones especificados por expresiones regulares, pueden hacer cosas que las cade-

nas de caracteres no pueden hacer, como que alguno de sus elementos coincida con más de un carácter.

Un ejemplo complejo de expresión regular puede ser el siguiente, un patrón para analizar URLs:

```
var parse_url =
/^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:

var url = "http://www.ora.com:80/goodparts?q#fragment";
```

Utilicemos el método `exec`, que procesa la cadena de caracteres en función del patrón proporcionado, devolviendo un *array* que contiene las partes extraidas:

```
var result = parse_url.exec(url);

var names = ['url', 'scheme', 'slash', 'host', 'port',
     'path', 'query', 'hash'];

var i;
for (i = 0; i < names.length; i += 1) {
    document.writeln(names[i] + ':\n', result[i]);
}
```

Donde el resultado es el siguiente:

```
url:    http://www.ora.com:80/goodparts?q#fragment
scheme: http
slash:  //
host:   www.ora.com
port:   80
path:   goodparts
query:  q
hash:   fragment
```

La principal manera de construir expresiones regulares, es utilizando los literales, de la siguiente manera:

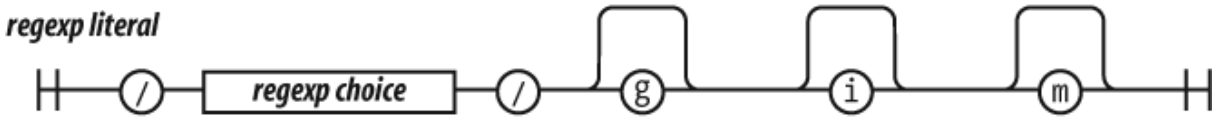


Figura 11.1 Construcción de una expresión literal

Podemos indicar tres parámetros al final de la expresión regular, que modifican ligeramente su comportamiento.

```
var my_regexp = /\s*/g;
```

Su significado es el siguiente:

`g`	Global (hace coincidir el patrón todas las veces posibles, aunque puede variar según el método).
`i`	Insensitive (ignora las mayúsculas y minúsculas).
`m`	Multiline (los caracteres `^` y `$` pueden coincidir con caracteres multilínea).

El punto se interpreta como *cualquier carácter*, es decir, busca cualquier carácter sin incluir los saltos de línea.

```
var point = /g.ant/;
var story = "We noticed the *gi\nant sloth*, hanging from a giant
branch.";
alert(story.search(point));     // 17
```

Se utiliza para "marcar" el siguiente carácter de la expresión de búsqueda, de forma que este adquiera un significado especial o deje de tenerlo. Es decir, la barra invertida no se utiliza nunca por sí sola, sino en combinación con otros caracteres. Al utilizarlo por ejemplo en combinación con el punto "." este deja de tener su significado normal y se comporta como un carácter literal.

De la misma forma, cuando se coloca la barra invertida seguida de cualquiera de los caracteres especiales listados a continuación, estos dejan de tener su significado especial y se convierten en caracteres de búsqueda literal.

- `\t`: representa un tabulador.
- `\r`: representa el "retorno de carro".

- `\n`: representa la "nueva línea".
- `\e`: representa la tecla "Esc" o "Escape".
- `\f`: se utiliza para representar caracteres ASCII o ANSI si se conoce su código. Por ejemplo el símbolo © es representado mediante "\fA9".
- `\x`: se utiliza para representar caracteres Unicode si se conoce su código. Por ejemplo, "\u02" representa el símbolo de centavos.
- `\u`: se utiliza para representar caracteres Unicode si se conoce su código. Por ejemplo, "\u00A2" representa el símbolo de centavos.
- `\d`: representa un dígito del 0 al 9.
- `\w`: representa cualquier carácter alfanumérico (incluidos los guiones bajos _).
- `\s`: representa un espacio en blanco (espacios en blanco, tabuladores y nuevas líneas).
- `\b`: marca el inicio y el final de una palabra.

Para los caracteres `d`, `w` y `s`, se puede utlizar su variante en mayúsculas para indicar justamente su significado opuesto.

- `\D`: representa cualquier carácter que no sea un dígito del 0 al 9.
- `\W`: representa cualquier carácter no alfanumérico.
- `\S`: representa cualquier carácter que no sea un espacio en blanco.

```
var digitSurroundedBySpace = /\s\d\s/;
alert("1a 2 3d".search(digitSurroundedBySpace));    // 2
```

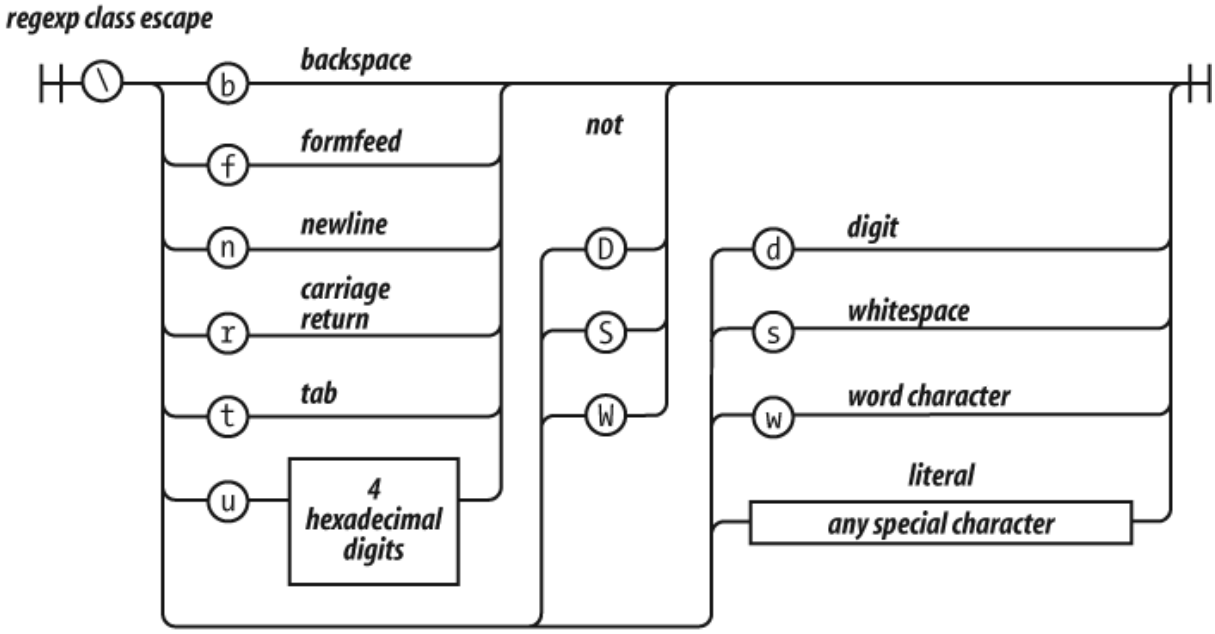


Figura 11.2 Caracteres de escape

La función de los corchetes es representar "clases de caracteres", es decir, agrupar caracteres en grupos o clases. Dentro de los corchetes es posible utilizar el guion "-" para especificar rangos de caracteres. Hay que tener en cuenta que dentro de los corchetes, los metacaracteres ($, *, +, ?) pierden su significado y se convierten en literales cuando se encuentran dentro de estos corchetes.

```
var asteriskOrBrace = /[{*]/;
var story = "We noticed the *giant sloth*, hanging from a giant branch.";
alert(story.search(asteriskOrBrace));       // 15
```

Cuando al inicio de este grupo de caracteres se encuentra el carácter ^, permite encontrar cualquier carácter que NO se encuentre dentro del grupo indicado.

```
var notABC = /[^ABC]/;
alert("ABCBACCBBADABC".search(notABC));     // 10
```

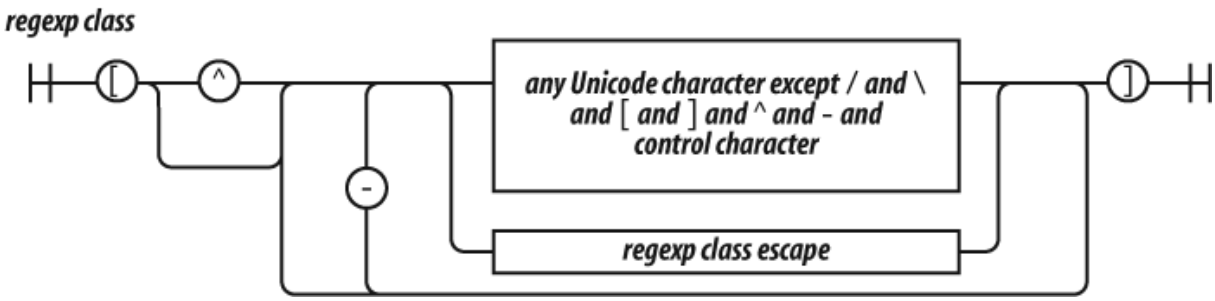


Figura 11.3 Caracteres de clase

Sirve para indicar una de varias opciones.

```
var cardinalPoints = /north|south|east|west/i;
alert(cardinalPoints.test("At north"));                 // true
alert(cardinalPoints.test("I'm from Southampton"));     // true
```

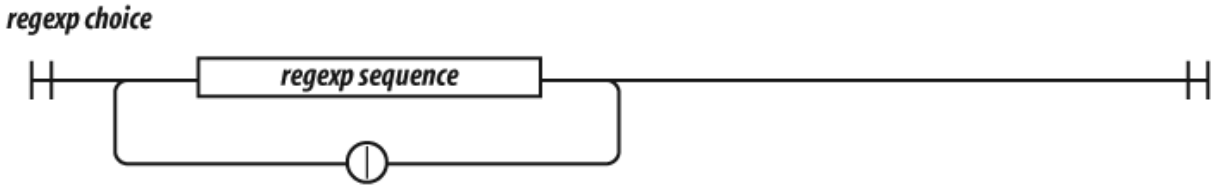


Figura 11.4 Carácter para indicar posibles opciones

Representa el final de la cadena de caracteres o el final de la línea, si se utiliza el modo multi-línea. No representa un carácter en especial sino una posición. Si se utiliza la expresión regular ".$", el motor encontrará todos los lugares donde un punto finalice la línea, lo que es útil para avanzar entre párrafos.

```
alert(/a+/.test("blah"));        // true
alert(/a+$/.test("blah"));       // false
alert(/a+$/.test("blahaa"));     // true
```

Éste carácter tiene una doble funcionalidad, en función de si utiliza individualmente y si se utiliza en conjunto con otros caracteres especiales. Su funcionalidad como carácter individual: el carácter `^` representa el inicio de la cadena (de la misma forma que el signo de dólar `$` representa el final de la cadena).

```
alert(/a+/.test("blah"));        // true
alert(/^a+/.test("blah"));       // false
alert(/^a+/.test("aaaablah"));     // true
```

Cuando se utiliza en conjunto con los corchetes, permite encontrar cualquier carácter que NO se encuentre dentro del grupo indicado.

```
var notABC = /[^ABC]/;
alert("ABCBACCBBADABC".search(notABC));     // 10
```

La utilización en conjunto de los caracteres especiales `^` y `$` permite realizar validaciones de forma sencilla. Por ejemplo `^\d$` permite asegurar

que la cadena a verificar representa un único dígito, mientras que `^\d\d/\d\d/\d\d\d\d$` permite validar una fecha en formato corto.

De forma similar que los corchetes, los paréntesis sirven para agrupar caracteres, sin embargo existen varias diferencias fundamentales entre los grupos establecidos por medio de corchetes y los grupos establecidos por paréntesis:

- Los caracteres especiales conservan su significado dentro de los paréntesis.
- Los grupos establecidos con paréntesis establecen una "etiqueta" o "punto de referencia" para el motor de búsqueda, que puede ser utilizada posteriormente.
- Utilizados en conjunto con la barra `|` permiten hacer búsquedas opcionales.
- Utilizados en conjunto con otros caracteres especiales que se detallan posteriormente, ofrece funcionalidad adicional.

```
var holyCow = /(sacred|holy) (cow|bovine|bull|taurus)/i;
alert(holyCow.test("Sacred bovine!"));      // true
```

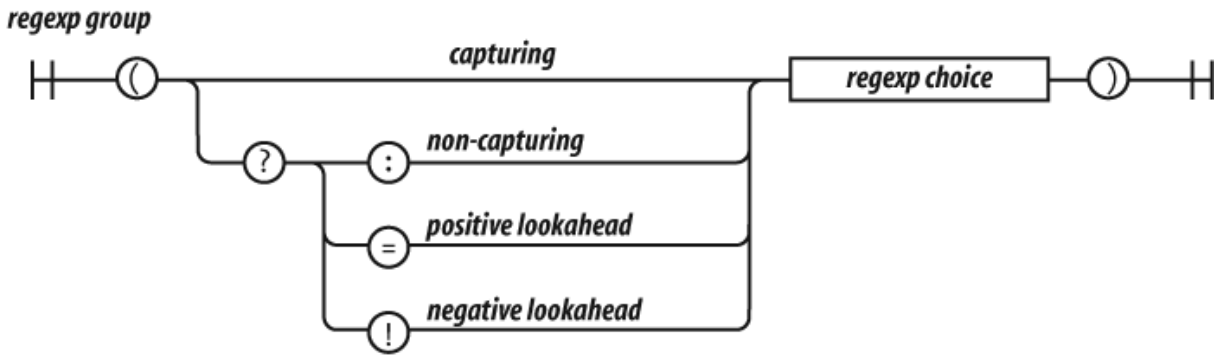


Figura 11.5 Grupos de caracteres

El signo de interrogación especifica que una parte de la búsqueda es opcional. En conjunto con los paréntesis, permite especificar que un conjunto mayor de caracteres es opcional

```
var nov = /Nov(\.|iembre|ember)?/;
alert(nov.test("Nov"));             // true
alert(nov.test("Nov."));            // true
```

```
alert(nov.test("Noviembre"));        // true
alert(nov.test("November"));         // true
```

Como hemos indicado antes, los paréntesis pueden ser utilizados como puntos de referencia dentro de las expresiones regulares. Si no es necesario, podemos evitar este comportamiento de la siguiente manera, liberando al motor de búsqueda de un trabajo extra:

```
var nov = /Nov(?:\.|iembre|ember)?/;
```

Comúnmente las llaves son caracteres literales cuando se utilizan por separado en una expresión regular. Para que adquieran su función de metacaracteres es necesario que encierren uno o varios números separados por coma y que estén colocados a la derecha de otra expresión regular de la siguiente forma:

```
var date = /^\d{2}\/\d{2}\/\d{2,4}$/;
alert(date.test("05/05/1982"));      // true
alert(date.test("05/05/82"));        // true
```

Son utilizados para indicar el número de veces que puede darse una coincidencia. Un número entre llaves (`{4}`) indica el número exacto de coincidencias. Dos números separados por una coma (`{2,4}`), indica que puede coincidir al menos tantas veces como el primer número (2), y cómo máximo tántas veces como el segundo número (4). De manera similar, `{2,}` significa que al menos ocurre dos veces y `{,4}`, que como máximo ocurre cuatro veces.

El asterisco sirve para encontrar algo que se encuentra repetido 0 o más veces. Por ejemplo:

```
var exp = /[a-zA-Z]\d*/;
alert(exp.test("A"));            // true
alert(exp.test("B0"));           // true
alert(exp.test("c01"));          // true
alert(exp.test("abc01234"));     // true
alert(exp.test("01234"));        // false
```

El asterisco equivale a la expresión `{0,}`.

Se utiliza para encontrar una cadena que se encuentre repetida una o más veces. A diferencia del asterisco, la expresión `[a-zA-Z]\d+` encontrará "H1" pero no encontrará "H".

```
var exp = /[a-zA-Z]\d+/;
alert(exp.test("A"));           // false
alert(exp.test("B0"));          // true
alert(exp.test("c01"));         // true
alert(exp.test("abc01234"));    // true
alert(exp.test("01234"));       // false
```

El signo de suma equivale a la expresión `{1,}`.

Los grupos anónimos se establecen cada vez que se encierra una expresión regular en paréntesis, por lo que la expresión `([a-zA-Z]\w*?)` define un grupo anónimo que tendrá como resultado que el motor de búsqueda almacenará una referencia al texto que corresponda a la expresión encerrada entre los paréntesis.

Es posible hacer referencia a estos grupos dentro de la propia expresión regular, o utilizarlos para extraer partes de la cadena de caracteres, si utilizamos los métodos que corresponden.

```
var exp = /<([a-zA-Z]\w*?)>.*?<\/\1>/;
alert(exp.test("<font>Text</font>"));       // true
alert(exp.test("<h1>Text</font>"));     // false
```

Éste método es el más poderoso (y lento) de todos los métodos que se pueden utilizar en expresiones regulares. Si la cadena de caracteres satisface el patrón, este método devuelve un *array*. El elemento `0` contiene la cadena de caracteres que coincide con la expresión regular, el elemento `1` el texto que coincide con el grupo 1, el elemento `2` el texto que coincide con el grupo 2... Si el patrón falla, la función devuelve `null`.

Éste método es el más simple (y rápido) de todos los métodos que se pueden utilizar en expresiones regulares. Si la cadena de caracteres satisface el patrón, este método devuelve `true`, y `false` en caso contrario.

Éste método compara la cadena de caracteres con la expresión regular, pero su comportamiento depende del parámetro `g`. Si éste parámetro no está presente, el comportamiento es el mismo que so llamamos al método `regexp.exec(string)`. En cambio, si está presente, devuelve un array con todas las coincidencias, pero excluye los grupos.

```
var text = '<html><body bgcolor=linen><p>' +
           'This is <b>bold<\/b>!<\/p><\/body><\/html>';
var tags = /[^<>]+|<(\/?)([A-Za-z]+)([^<>]*)>/g;

var a, i;
a = text.match(tags);
for (i = 0; i < a.length; i += 1) {
    document.writeln(('// [' + i + '] ' + a[i]).entityify());
}

// The result is
// [0] <html>
// [1] <body bgcolor=linen>
// [2] <p>
// [3] This is
// [4] <b>
// [5] bold
// [6] </b>
// [7] !
// [8] </p>
// [9] </body>
// [10] </html>
```

Éste método busca y reemplaza los valores indicados, devolviendo un nuevo string. El parámetro `searchValue` puede ser una cadena de caracteres o una expresión regular. Si es un `string`, únicamente la primera aparición es reemplazada, lo que puede ser un poco confuso.

```
var result = "mother_in_law".replace('_', '-');
alert(result);      //mother-in_law
```

Si el parámetro es una expresión regular, y contiene el parámetro `g`, entonces reemplazará todas las apariciones. Si no contiene este parámetro, entonces sólo se reemplaza la primera aparición.

El parámetros `replaceValue` puede ser un *string* o una fución. Si el parámetro es una función, ésta será llamada por cada coincidencia, y el *string* devuelto por la función será utilizado como cadena de reemplazo. El parámetro pasado a esta función corresponde con la coincidencia de texto.

```
var character = {
                  '<' : '&lt;',
                  '>' : '&gt;',
                  '&' : '&amp;',
                  '"' : '&quot;'
                };
var entities = "<&>";
entities.replace(/[<>&"]/g,  function (c) {
            return character[c];
         });
```

Éste método de búsqueda es igual al método `indexOf`, a excepción de que éste método toma como parámetro una expresión regular en vez de un *string*. Devuelve la posición del primer carácter que coincide con la expresión regular, si hay alguno, y `-1` si no hay coincidencias. En este caso, el parámetro `g` es ignorado.

```
var text = 'and in it he says "Any damn fool could';
var pos = text.search(/["']/);    // 18
```

Éste método crea un array de *strings* , dividiendo el *string* original en trozos. El parámetro `separator` puede ser un *string* o una expresión regular. El parámetro opcional `limit` indica el número máximo de trozos en los que se va a dividir el *string*.

```
var c = '|a|b|c|'.split('|');   // c is ['', 'a', 'b', 'c', '']

var text = 'last,  first  ,middle';
var d = text.split(/\s*,\s*/);
// d is [
//       'last',
//       'first',
```

```
//       'middle'
//     ]
```

Ejercicio 11

Ver enunciado (#ej11)

Capítulo 12

JavaScript Object Notation (JSON) es un formato de intercambio de datos muy ligero y está basdo en la notación para la representación de objetos. A pesar de estar basado en JavaScript, es independiente del lenguaje. Puede ser utilizado para intercambiar datos entre programas escritos en lenguajes totalmente diferentes. Es un formato de texto, por lo que puede ser leído por máquinas y humanos, he implementado de una manera muy sencilla. Para acceder a toda la información sobre JSON, acceder a http://www.JSON.org/.

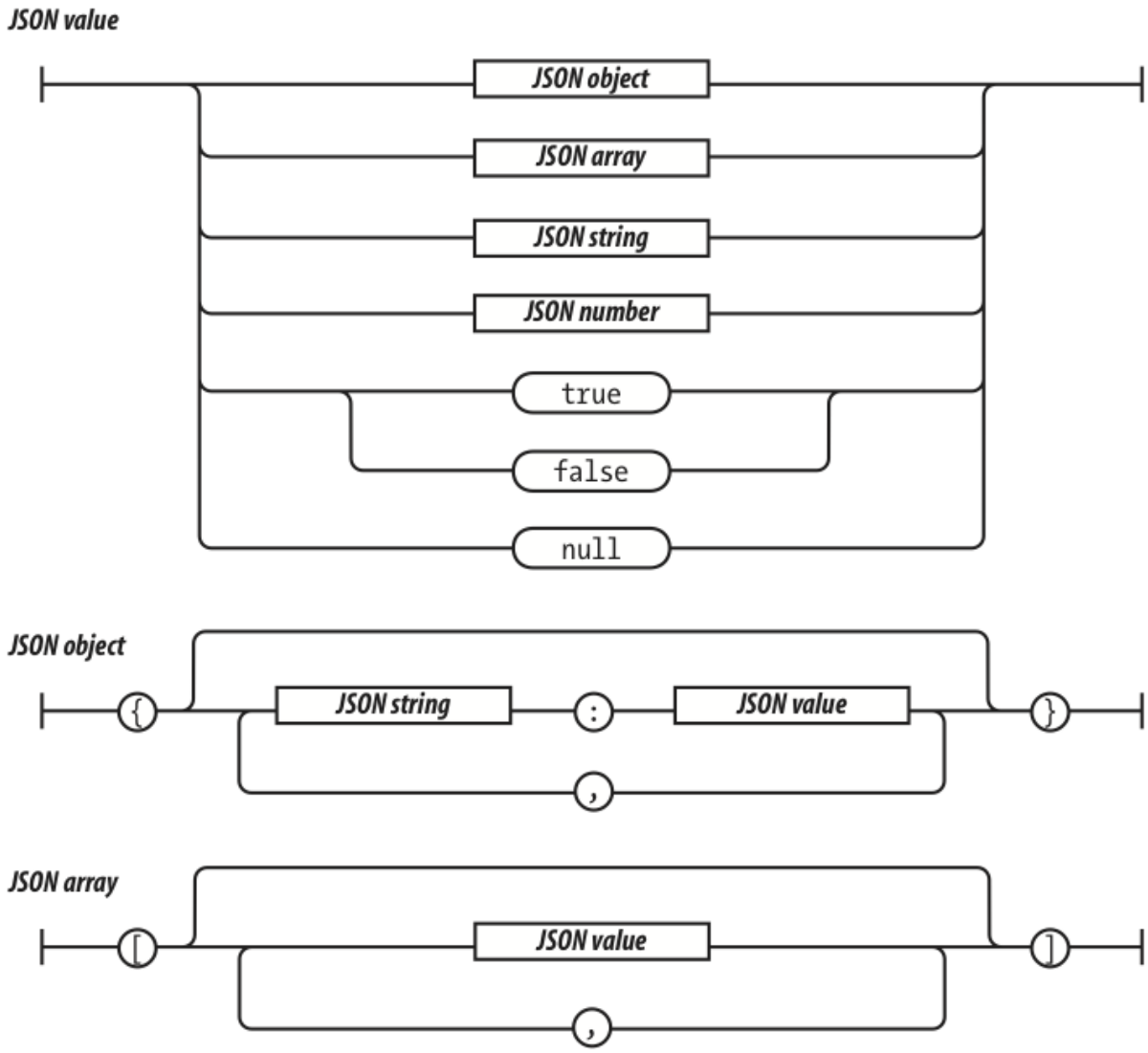
JSON define seis tipos de valores: `objects`, `arrays`, `strings`, `numbers`, `booleans` y el valor especial `null`. Los espacios (espacios en blanco, tabuladores, retornos de carro y nueva línea) pueden introducirse antes o después de cualquier valor, sin afectar a los valores representados. Esto hace que un texto JSON sea mucho más fácil de leer por humanos.

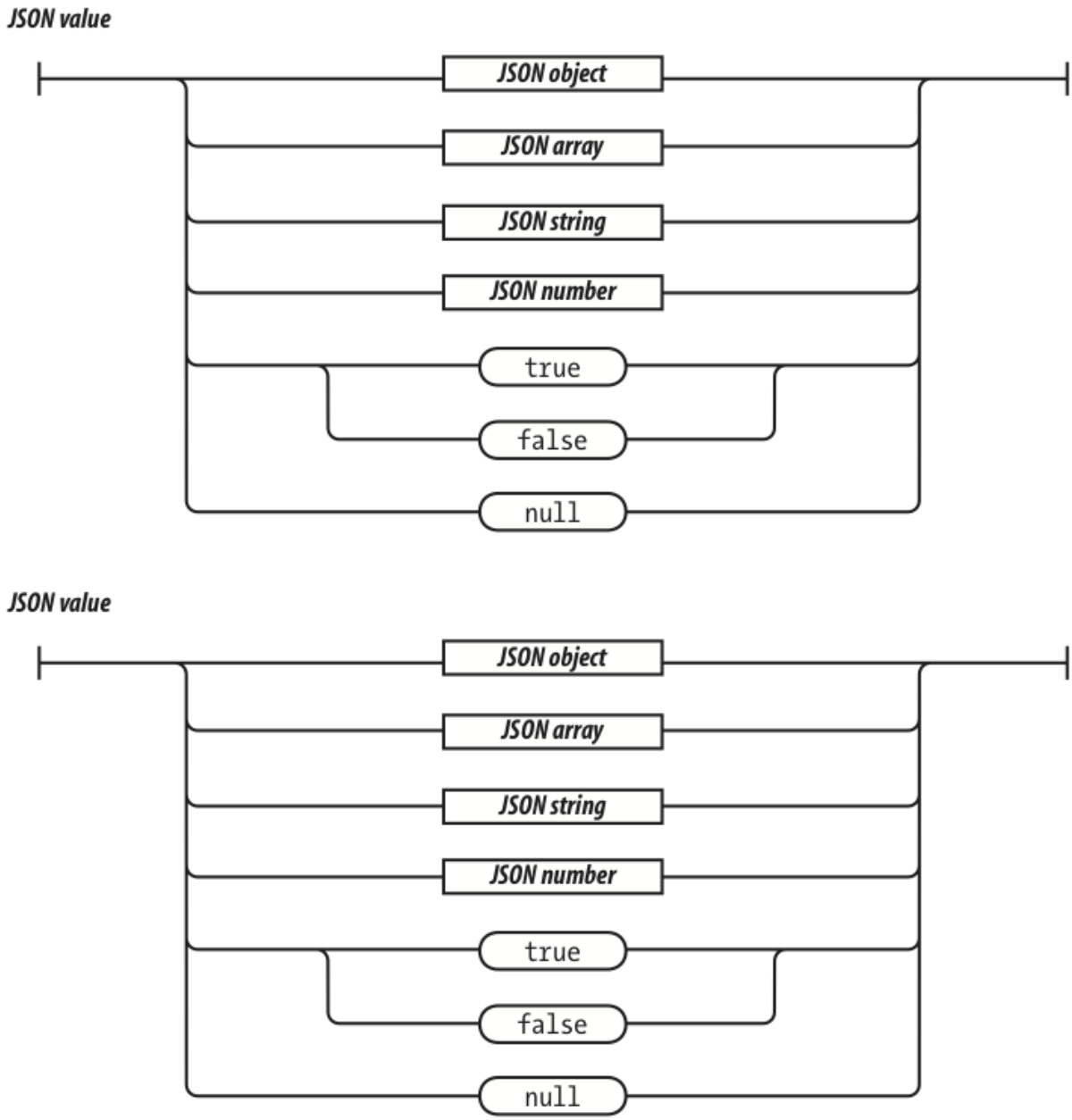
Un objeto JSON es un contenedor, no ordenado de parejas clave/valor. Una clave puede ser un string, y un valor puede ser un valor JSON (tanto un array como un objeto). Los objetos JSON se pueden anidar hasta cualquier profundidad. Un array JSON es una secuencia ordenada de valores, donde un valor puede ser un valor JSON (tanto un array como un objeto).

La gran mayoría de lenguajes incluyen características para trabajar de manera cómoda con valores JSON en ambos sentidos: partiendo de un

objeto u array para convertirlo a una cadena de caracteres, o a partir de una cadena de caracteres, obtener los valores JSON.

La sintaxis de los valores JSON es la siguiente:





Utilizar JSON es realmente sencillo, ya que JSON es JavaScript. Un cadena de texto JSON puede convertirse en una estructura de datos de JavaScript utilizando la función `eval`:

```
var myData = eval('(' + myJSONText + ')');
```

La utilización de la función `eval` conlleva graves problemas de seguridad, por lo que su uso está totalmente desaconsejado. En su lugar, es recomendable utilizar el método `JSON.parse`, disponible en github (https://github.com/douglascrockford/JSON-js) . El método JSON.parse

lanzará una excepción si la cadena que se intenta evaluar contiene un error.

Otro de los problemas puede venir de la interacción con el servidor y la propiedad `innerHTML` de los elementos del DOM. Un patrón común en los desarrollos con AJAX, es que el servidor devuelva un código HTML que nosotros asignamos directamente a un elemento del documento. Si el código HTML contiene una etiqueta `script`, entonces un script malicioso puede actuar.

¿Pero qué es lo realmente peligroso? Si un script consigue ejecutarse en una página, éste obtiene acceso al estado y las capacidades de la página. Puede interactuar con el servidor (sin que este pueda distingir una petición legítima de una maliciosa). Puede acceder al objeto global de JavaScript, a todos los datos de la aplicación (variables de JavaScript), al DOM y todo lo que el usuario está viendo e interactuando... De la misma manera que si el `script` hubiese sido programado por nosotros.

Este peligro viene dado por los objetos globales de JavaScript, no por AJAX, JSON, XMLHttpRequest o la Web 2.0. Simplemente el peligro está ahí desde la introducción de JavaScript en los navegadores, y seguirá ahí hasta que JavaScript sea reemplazado o reparado.

Capítulo 13

La creación del *Document Object Model* o **DOM** es una de las innovaciones que más ha influido en el desarrollo de las páginas web dinámicas y de las aplicaciones web más complejas.

DOM permite a los programadores web acceder y manipular las páginas XHTML como si fueran documentos XML. De hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.

A pesar de sus orígenes, DOM se ha convertido en una utilidad disponible para la mayoría de lenguajes de programación (Java, PHP, JavaScript) y cuyas únicas diferencias se encuentran en la forma de implementarlo.

Una de las tareas habituales en la programación de aplicaciones web con JavaScript consiste en la manipulación de las páginas web. De esta forma, es habitual obtener el valor almacenado por algunos elementos (por ejemplo los elementos de un formulario), crear un elemento (párrafos, `<div>`, etc.) de forma dinámica y añadirlo a la página, aplicar una animación a un elemento (que aparezca/desaparezca, que se desplace, etc.).

Todas estas tareas habituales son muy sencillas de realizar gracias a DOM. Sin embargo, para poder utilizar las utilidades de DOM, es necesario *"transformar"* la página original. Una página HTML normal no es más que una sucesión de caracteres, por lo que es un formato muy difícil de

manipular. Por ello, los navegadores web transforman automáticamente todas las páginas web en una estructura más eficiente de manipular.

Esta transformación la realizan todos los navegadores de forma automática y nos permite utilizar las herramientas de DOM de forma muy sencilla. El motivo por el que se muestra el funcionamiento de esta transformación interna es que condiciona el comportamiento de DOM y por tanto, la forma en la que se manipulan las páginas.

DOM transforma todos los documentos XHTML en un conjunto de elementos llamados nodos, que están interconectados y que representan los contenidos de las páginas web y las relaciones entre ellos. Por su aspecto, la unión de todos los nodos se llama "árbol de nodos".

La siguiente página XHTML sencilla:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Página sencilla</title>
</head>

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

Se transforma en el siguiente árbol de nodos:

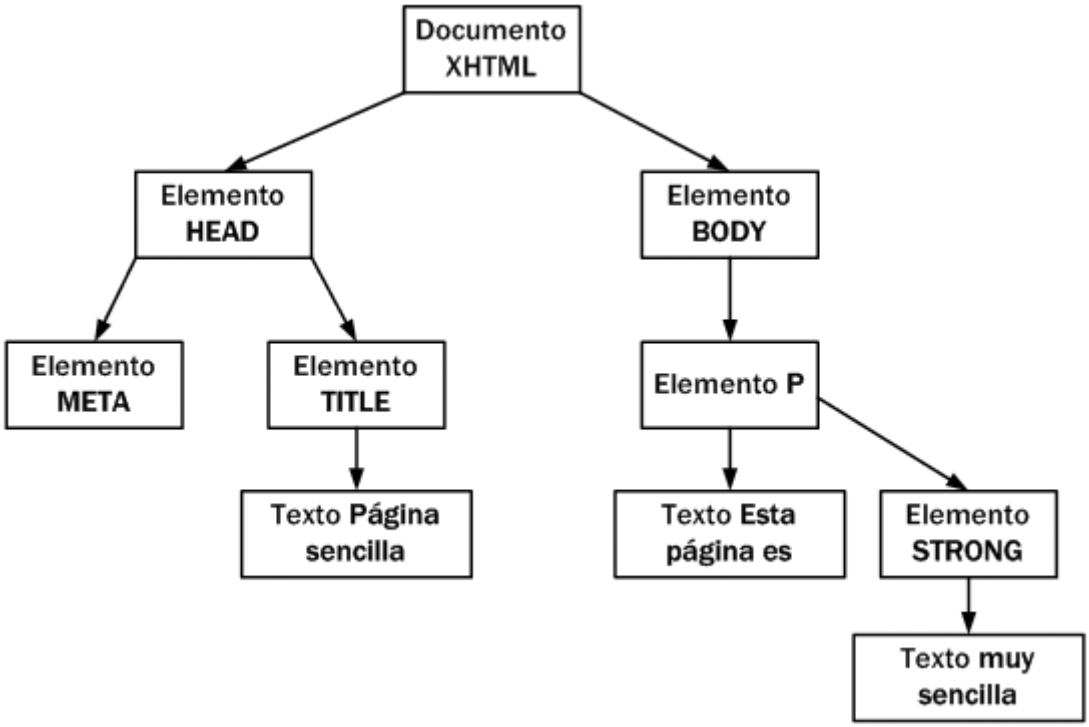


Figura 13.1 Árbol de nodos generado automáticamente por DOM a partir del código XHTML de la página

En el esquema anterior, cada rectángulo representa un nodo DOM y las flechas indican las relaciones entre nodos. Dentro de cada nodo, se ha incluido su tipo (que se verá más adelante) y su contenido.

La raíz del árbol de nodos de cualquier página XHTML siempre es la misma: un nodo de tipo especial denominado *"Documento"*.

A partir de ese nodo raíz, cada etiqueta XHTML se transforma en un nodo de tipo *"Elemento"*. La conversión de etiquetas en nodos se realiza de forma jerárquica. De esta forma, del nodo raíz solamente pueden derivar los nodos `HEAD` y `BODY`. A partir de esta derivación inicial, cada etiqueta XHTML se transforma en un nodo que deriva del nodo correspondiente a su "etiqueta padre".

La transformación de las etiquetas XHTML habituales genera dos nodos: el primero es el nodo de tipo *"Elemento"* (correspondiente a la propia etiqueta XHTML) y el segundo es un nodo de tipo *"Texto"* que contiene el texto encerrado por esa etiqueta XHTML.

Así, la siguiente etiqueta XHTML:

```
<title>Página sencilla</title>
```

Genera los siguientes dos nodos:

Figura 13.2 Nodos generados automáticamente por DOM para una etiqueta XHTML sencilla

De la misma forma, la siguiente etiqueta XHTML:

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

Genera los siguientes nodos:

- Nodo de tipo *"Elemento"* correspondiente a la etiqueta `<p>`.
- Nodo de tipo *"Texto"* con el contenido textual de la etiqueta `<p>`.
- Como el contenido de `<p>` incluye en su interior otra etiqueta XHTML, la etiqueta interior se transforma en un nodo de tipo "Elemento" que representa la etiqueta `<strong>` y que deriva del nodo anterior.
- El contenido de la etiqueta `<strong>` genera a su vez otro nodo de tipo *"Texto"* que deriva del nodo generado por `<strong>`.

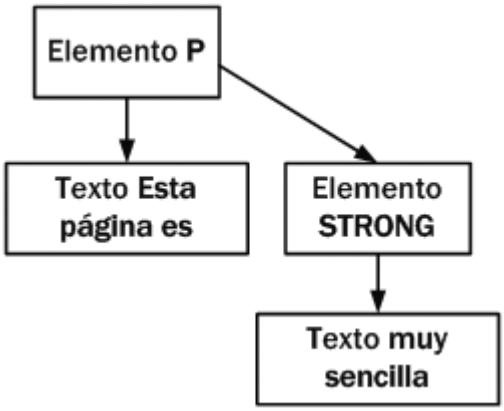


Figura 13.3 Nodos generados automáticamente por DOM para una etiqueta XHTML con otras etiquetas XHTML en su interior

La transformación automática de la página en un árbol de nodos siempre sigue las mismas reglas:

- Las etiquetas XHTML se transforman en dos nodos: el primero es la propia etiqueta y el segundo nodo es hijo del primero y consiste en el contenido textual de la etiqueta.
- Si una etiqueta XHTML se encuentra dentro de otra, se sigue el mismo procedimiento anterior, pero los nodos generados serán nodos hijo de su etiqueta padre.

Como se puede suponer, las páginas XHTML habituales producen árboles con miles de nodos. Aun así, el proceso de transformación es rápido y automático, siendo las funciones proporcionadas por DOM (que se verán más adelante) las únicas que permiten acceder a cualquier nodo de la página de forma sencilla e inmediata.

La especificación completa de DOM define 12 tipos de nodos, aunque las páginas XHTML habituales se pueden manipular manejando solamente cuatro o cinco tipos de nodos:

- `Document`, nodo raíz del que derivan todos los demás nodos del árbol.
- `Element`, representa cada una de las etiquetas XHTML. Se trata del único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- `Attr`, se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas XHTML, es decir, uno por cada par `atributo=valor`.
- `Text`, nodo que contiene el texto encerrado por una etiqueta XHTML.
- `Comment`, representa los comentarios incluidos en la página XHTML.

Los otros tipos de nodos existentes que no se van a considerar son `DocumentType`, `CDataSection`, `DocumentFragment`, `Entity`, `EntityReference`, `ProcessingInstruction` y `Notation`.

Una vez construido automáticamente el árbol completo de nodos DOM, ya es posible utilizar las funciones DOM para acceder de forma directa a cualquier nodo del árbol. Como acceder a un nodo del árbol es equivalente a acceder a "un trozo" de la página, una vez construido el árbol, ya es posible manipular de forma sencilla la página: acceder al valor de un elemento, establecer el valor de un elemento, mover un elemento de la página, crear y añadir nuevos elementos, etc.

DOM proporciona dos métodos alternativos para acceder a un nodo específico: acceso a través de sus nodos padre y acceso directo.

Las funciones que proporciona DOM para acceder a un nodo a través de sus nodos padre consisten en acceder al nodo raíz de la página y después a sus nodos hijos y a los nodos hijos de esos hijos y así sucesivamente hasta el último nodo de la rama terminada por el nodo buscado. Sin embargo, cuando se quiere acceder a un nodo específico, es mucho más rápido acceder directamente a ese nodo y no llegar hasta él descendiendo a través de todos sus nodos padre.

Por ese motivo, no se van a presentar las funciones necesarias para el acceso jerárquico de nodos y se muestran solamente las que permiten acceder de forma directa a los nodos.

Por último, es importante recordar que el acceso a los nodos, su modificación y su eliminación solamente es posible cuando el árbol DOM ha sido construido completamente, es decir, después de que la página XHTML se cargue por completo. Más adelante se verá cómo asegurar que un código JavaScript solamente se ejecute cuando el navegador ha cargado entera la página XHTML.

Como sucede con todas las funciones que proporciona DOM, la función `getElementsByTagName()` tiene un nombre muy largo, pero que lo hace autoexplicativo.

La función `getElementsByTagName(nombreEtiqueta)` obtiene todos los elementos de la página XHTML cuya etiqueta sea igual que el parámetro que se le pasa a la función.

El siguiente ejemplo muestra cómo obtener todos los párrafos de una página XHTML:

```
var parrafos = document.getElementsByTagName("p");
```

El valor que se indica delante del nombre de la función (en este caso, `document`) es el nodo a partir del cual se realiza la búsqueda de los elementos. En este caso, como se quieren obtener todos los párrafos de la página, se utiliza el valor `document` como punto de partida de la búsqueda.

El valor que devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. El valor devuelto es un array de nodos DOM, no un array de cadenas de texto o un array de objetos normales. Por lo tanto, se debe procesar cada valor del array de la forma que se muestra en las siguientes secciones.

De este modo, se puede obtener el primer párrafo de la página de la siguiente manera:

```
var primerParrafo = parrafos[0];
```

De la misma forma, se podrían recorrer todos los párrafos de la página con el siguiente código:

```
for(var i=0; i<parrafos.length; i++) {
  var parrafo = parrafos[i];
}
```

La función `getElementsByTagName()` se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función. En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
var parrafos = document.getElementsByTagName("p");
var primerParrafo = parrafos[0];
var enlaces = primerParrafo.getElementsByTagName("a");
```

La función `getElementsByName()` es similar a la anterior, pero en este caso se buscan los elementos cuyo atributo `name` sea igual al parámetro proporcionado. En el siguiente ejemplo, se obtiene directamente el único párrafo con el nombre indicado:

```
var parrafoEspecial = document.getElementsByName("especial");
```

```
<p name="prueba">...</p>
<p name="especial">...</p>
<p>...</p>
```

Normalmente el atributo `name` es único para los elementos HTML que lo definen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML *radiobutton*, el atributo `name` es común a todos los *radiobutton* que están relacionados, por lo que la función devuelve una colección de elementos.

La función `getElementById()` es la más utilizada cuando se desarrollan aplicaciones web dinámicas. Se trata de la función preferida para acceder directamente a un nodo y poder leer o modificar sus propiedades.

La función `getElementById()` devuelve el elemento XHTML cuyo atributo id coincide con el parámetro indicado en la función. Como el atributo id debe ser único para cada elemento de una misma página, la función devuelve únicamente el nodo deseado.

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

```
<div id="cabecera">
  <a href="/" id="logo">...</a>
</div>
```

La función `getElementById()` es tan importante y tan utilizada en todas las aplicaciones web, que casi todos los ejemplos y ejercicios que siguen la utilizan constantemente.

La función `querySelector()` acepta como parámetro un selector que identifica el elemento (o elementos) a seleccionar. En el caso de esta función, únicamente es devuelto **el primer elemento** que cumple la condición. Si no existe el elemento, el valor retornado es `null`.

```
var logo = document.querySelector(".enlace");
```

```
<div id="cabecera">
  <a href="/" class="enlace">...</a>
</div>
<div id="cuerpo">
```

```
    <p>Loren ipsum <a href="enlace">...</a></p>
</div>
```

En este caso, a pesar de existir varios elementos de la clase `enlace`, únicamente es seleccionado el primero de ellos.

La función `querySelectorAll()` acepta como parámetro un selector que identifica el elemento (o elementos) a seleccionar. Esta función devuelve un objeto de tipo `NodeList` con los elementos que coincidan con el selector.

```
var enlaces = document.querySelectorAll(".enlace");
```

```
<div id="cabecera">
  <a href="/" class="enlace">...</a>
</div>
<div id="cuerpo">
    <p>Loren ipsum <a href="enlace">...</a></p>
</div>
```

Para acceder a los elementos almacenados en `NodeList`, recorremos el objeto como si de un *array* se tratase:

```
for (var i=0; i<enlaces.length; i++) {
    var enlaces = enlaces[i];
}
```

Acceder a los nodos y a sus propiedades (que se verá más adelante) es sólo una parte de las manipulaciones habituales en las páginas. Las otras operaciones habituales son las de crear y eliminar nodos del árbol DOM, es decir, crear y eliminar *"trozos"* de la página web.

Como se ha visto, un elemento XHTML sencillo, como por ejemplo un párrafo, genera dos nodos: el primer nodo es de tipo `Element` y representa la etiqueta `<p>` y el segundo nodo es de tipo `Text` y representa el contenido textual de la etiqueta `<p>`.

Por este motivo, crear y añadir a la página un nuevo elemento XHTML sencillo consta de cuatro pasos diferentes:

1. Creación de un nodo de tipo `Element` que represente al elemento.
2. Creación de un nodo de tipo `Text` que represente el contenido del elemento.
3. Añadir el nodo `Text` como nodo hijo del nodo `Element`.
4. Añadir el nodo `Element` a la página, en forma de nodo hijo del nodo correspondiente al lugar en el que se quiere insertar el elemento.

De este modo, si se quiere añadir un párrafo simple al final de una página XHTML, es necesario incluir el siguiente código JavaScript:

```
// Crear nodo de tipo Element
var parrafo = document.createElement("p");

// Crear nodo de tipo Text
var contenido = document.createTextNode("Hola Mundo!");

// Añadir el nodo Text como hijo del nodo Element
parrafo.appendChild(contenido);

// Añadir el nodo Element como hijo de la pagina
document.body.appendChild(parrafo);
```

El proceso de creación de nuevos nodos puede llegar a ser tedioso, ya que implica la utilización de tres funciones DOM:

- `createElement(etiqueta)`: crea un nodo de tipo `Element` que representa al elemento XHTML cuya etiqueta se pasa como parámetro.
- `createTextNode(contenido)`: crea un nodo de tipo `Text` que almacena el contenido textual de los elementos XHTML.
- `nodoPadre.appendChild(nodoHijo)`: añade un nodo como hijo de otro nodo. Se debe utilizar al menos dos veces con los nodos habituales: en primer lugar se añade el nodo `Text` como hijo del nodo `Element` y a continuación se añade el nodo `Element` como hijo de algún nodo de la página.

Afortunadamente, eliminar un nodo del árbol DOM de la página es mucho más sencillo que añadirlo. En este caso, solamente es necesario utilizar la función `removeChild()`:

```
var parrafo = document.getElementById("provisional");
parrafo.parentNode.removeChild(parrafo);
```

```
<p id="provisional">...</p>
```

La función `removeChild()` requiere como parámetro el nodo que se va a eliminar. Además, esta función debe ser ejecutada desde el elemento padre de ese nodo que se quiere eliminar. La forma más segura y rápida de acceder al nodo padre de un elemento es mediante la propiedad `nodoHijo.parentNode`.

Así, para eliminar un nodo de una página XHTML se llama a la función `removeChild()` desde el valor `parentNode` del nodo que se quiere eliminar. Cuando se elimina un nodo, también se eliminan automáticamente todos los nodos hijos que tenga, por lo que no es necesario borrar manualmente cada nodo hijo.

Una vez que se ha accedido a un nodo, el siguiente paso natural consiste en acceder y/o modificar sus atributos y propiedades. Mediante DOM, es posible acceder de forma sencilla a todos los atributos XHTML y todas las propiedades CSS de cualquier elemento de la página.

Los atributos XHTML de los elementos de la página se transforman automáticamente en propiedades de los nodos. Para acceder a su valor, simplemente se indica el nombre del atributo XHTML detrás del nombre del nodo.

El siguiente ejemplo obtiene de forma directa la dirección a la que enlaza el enlace:

```
var enlace = document.getElementById("enlace");
alert(enlace.href); // muestra http://www...com
```

```
<a id="enlace" href="http://www...com">Enlace</a>
```

En el ejemplo anterior, se obtiene el nodo DOM que representa el enlace mediante la función `document.getElementById()`. A continuación, se obtiene el atributo `href` del enlace mediante `enlace.href`. Para obtener por ejemplo el atributo `id`, se utilizaría `enlace.id`.

Las propiedades CSS requieren un paso extra para acceder a ellas. Para obtener el valor de cualquier propiedad CSS del nodo, se debe utilizar el

atributo `style`. El siguiente ejemplo obtiene el valor de la propiedad `margin` de la imagen:

```
var imagen = document.getElementById("imagen");
alert(imagen.style.margin);
```

```
<img id="imagen" style="margin:0; border:0;" src="logo.png" />
```

Si el nombre de una propiedad CSS es compuesto, se accede a su valor modificando ligeramente su nombre:

```
var parrafo = document.getElementById("parrafo");
alert(parrafo.style.fontWeight);  // muestra "bold"
```

```
<p id="parrafo" style="font-weight: bold;">...</p>
```

La transformación del nombre de las propiedades CSS compuestas consiste en eliminar todos los guiones medios (-) y escribir en mayúscula la letra siguiente a cada guión medio (lo que se conoce como nomenclatura *lowerCamelCase*). A continuación se muestran algunos ejemplos:

- `font-weight` se transforma en `fontWeight`
- `line-height` se transforma en `lineHeight`
- `border-top-style` se transforma en `borderTopStyle`
- `list-style-image` se transforma en `listStyleImage`

El único atributo XHTML que no tiene el mismo nombre en XHTML y en las propiedades DOM es el atributo `class`. Como la palabra `class` está reservada por JavaScript, no es posible utilizarla para acceder al atributo `class` del elemento XHTML. En su lugar, DOM utiliza el nombre `className` para acceder al atributo `class` de XHTML:

```
var parrafo = document.getElementById("parrafo");
alert(parrafo.class); // muestra "undefined"
alert(parrafo.className); // muestra "normal"
```

```
<p id="parrafo" class="normal">...</p>
```

Ejercicio 12

Ver enunciado (#ej12)

Ejercicio 13

Ver enunciado (#ej13)

Ejercicio 14

Ver enunciado (#ej14)

Esta página se ha dejado vacía a propósito

Capítulo 14

Las versiones 3.0 de los navegadores Internet Explorer y Netscape Navigator introdujeron el concepto de Browser Object Model o BOM, que permite acceder y modificar las propiedades de las ventanas del propio navegador.

Mediante BOM, es posible redimensionar y mover la ventana del navegador, modificar el texto que se muestra en la barra de estado y realizar muchas otras manipulaciones no relacionadas con el contenido de la página HTML.

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

Algunos de los elementos que forman el BOM son los siguientes:

- Crear, mover, redimensionar y cerrar ventanas de navegador.
- Obtener información sobre el propio navegador.
- Propiedades de la página actual y de la pantalla del usuario.
- Gestión de cookies.
- Objetos ActiveX en Internet Explorer.

El BOM está compuesto por varios objetos relacionados entre sí. El siguiente esquema muestra los objetos de BOM y su relación:

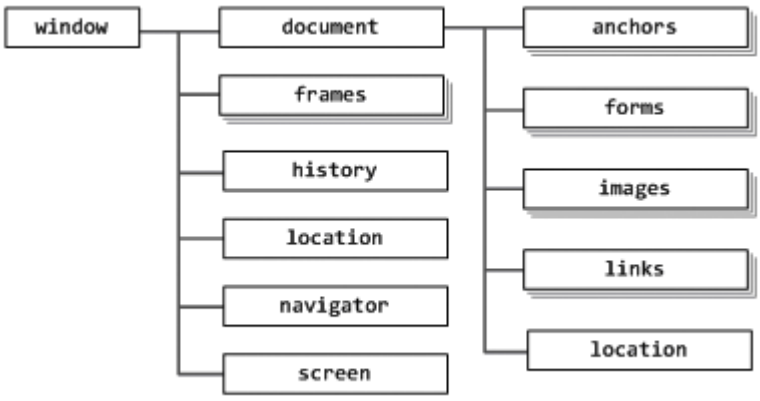


Figura 14.1 Jerarquía de objetos que forman el BOM

En el esquema anterior, los objetos mostrados con varios recuadros superpuestos son arrays. El resto de objetos, representados por un rectángulo individual, son objetos simples. En cualquier caso, todos los objetos derivan del objeto `window`.

El objeto `window` representa la ventana completa del navegador. Mediante este objeto, es posible mover, redimensionar y manipular la ventana actual del navegador. Incluso es posible abrir y cerrar nuevas ventanas de navegador.

Como todos los demás objetos heredan directa o indirectamente del objeto `window`, no es necesario indicarlo de forma explícita en el código JavaScript. En otras palabras:

```
window.forms[0] === forms[0]
window.document === document
```

BOM define cuatro métodos para manipular el tamaño y la posición de la ventana:

- `moveBy(x, y)` desplaza la posición de la ventana `x` píxel hacia la derecha y `y` píxel hacia abajo. Se permiten desplazamientos negativos para mover la ventana hacia la izquierda o hacia arriba.
- `moveTo(x, y)` desplaza la ventana del navegador hasta que la esquina superior izquierda se encuentre en la posición `(x, y)` de la pantalla del usuario. Se permiten desplazamientos negativos,

aunque ello suponga que parte de la ventana no se visualiza en la pantalla.

- `resizeBy(x, y)` redimensiona la ventana del navegador de forma que su nueva anchura sea igual a `(anchura_anterior + x)` y su nueva altura sea igual a `(altura_anterior + y)`. Se pueden emplear valores negativos para reducir la anchura y/o altura de la ventana.
- `resizeTo(x, y)` redimensiona la ventana del navegador hasta que su anchura sea igual a `x` y su altura sea igual a `y`. No se permiten valores negativos.

Los navegadores son cada vez menos permisivos con la modificación mediante JavaScript de las propiedades de sus ventanas. De hecho, la mayoría de navegadores permite a los usuarios bloquear el uso de JavaScript para realizar cambios de este tipo. De esta forma, una aplicación nunca debe suponer que este tipo de funciones están disponibles y funcionan de forma correta.

A continuación se muestran algunos ejemplos de uso de estas funciones:

```
// Mover la ventana 20 píxel hacia la derecha y 30 píxel hacia abajo
window.moveBy(20, 30);

// Redimensionar la ventana hasta un tamaño de 250 x 250
window.resizeTo(250, 250);

// Agrandar la altura de la ventana en 50 píxel
window.resizeBy(0, 50);

// Colocar la ventana en la esquina izquierda superior de la ventana
window.moveTo(0, 0);
```

Además de desplazar y redimensionar la ventana del navegador, es posible averiguar la posición y tamaño actual de la ventana. Sin embargo, la ausencia de un estándar para BOM provoca que cada navegador implemente su propio método:

- Internet Explorer proporciona las propiedades `window.screenLeft` y `window.screenTop` para obtener las coordenadas de la posición de la ventana. No es posible obtener el tamaño de la ventana completa, sino solamente del área visible de la página (es decir, sin barra de estado ni menús). Las propiedades que proporcionan estas

dimensiones son `document.body.offsetWidth` y `document.body.offsetHeight`.

- Los navegadores de la familia Mozilla, Safari y Opera proporcionan las propiedades `window.screenX` y `window.screenY` para obtener la posición de la ventana. El tamaño de la zona visible de la ventana se obtiene mediante `window.innerWidth` y `window.innerHeight`, mientras que el tamaño total de la ventana se obtiene mediante `window.outerWidth` y `window.outerHeight`.

Al contrario que otros lenguajes de programación, JavaScript no incorpora un método `wait()` que detenga la ejecución del programa durante un tiempo determinado. Sin embargo, JavaScript proporciona los métodos `setTimeout()` y `setInterval()` que se pueden emplear para realizar tareas similares.

El método `setTimeout()` permite ejecutar una función al transcurrir un determinado periodo de tiempo:

```
setTimeout("alert('Han transcurrido 3 segundos desde que me
programaron')", 3000);
```

El método `setTimeout()` requiere dos argumentos. El primero es el código que se va a ejecutar o una referencia a la función que se debe ejecutar. El segundo argumento es el tiempo, en milisegundos, que se espera hasta que comienza la ejecución del código. El ejemplo anterior se puede rehacer utilizando una función:

```
function muestraMensaje() {
  alert("Han transcurrido 3 segundos desde que me programaron");
}

setTimeout(muestraMensaje, 3000);
```

Como es habitual, cuando se indica la referencia a la función no se incluyen los paréntesis, ya que de otro modo, se ejecuta la función en el mismo instante en que se establece el intervalo de ejecución.

Cuando se establece una cuenta atrás, la función `setTimeout()` devuelve el identificador de esa nueva cuenta atrás. Empleando ese identificador y la función `clearTimeout()` es posible impedir que se ejecute el código pendiente:

```
function muestraMensaje() {
  alert("Han transcurrido 3 segundos desde que me programaron");
}
var id = setTimeout(muestraMensaje, 3000);

// Antes de que transcurran 3 segundos, se decide eliminar la ejecución
pendiente
clearTimeout(id);
```

Además de programar la ejecución futura de una función, JavaScript también permite establecer la ejecución periódica y repetitiva de una función. El método necesario es `setInterval()` y su funcionamiento es idéntico al mostrado para `setTimeout()`:

```
function muestraMensaje() {
  alert("Este mensaje se muestra cada segundo");
}

setInterval(muestraMensaje, 1000);
```

De forma análoga a `clearTimeout()`, también existe un método que permite eliminar una repetición periódica y que en este caso se denomina `clearInterval()`:

```
function muestraMensaje() {
  alert("Este mensaje se muestra cada segundo");
}

var id = setInterval(muestraMensaje, 1000);

// Despues de ejecutarse un determinado número de veces, se elimina el
intervalo
clearInterval(id);
```

El objeto `document` es el único que pertenece tanto al DOM (como se vio en el capítulo anterior) como al BOM. Desde el punto de vista del BOM, el objeto `document` proporciona información sobre la propia página HTML.

Algunas de las propiedades más importantes definidas por el objeto `document` son:

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

Las propiedades `title` y `URL` son de lectura y escritura, por lo que además de obtener su valor, se puede establecer de forma directa:

```
// modificar el título de la página
document.title = "Nuevo titulo";

// llevar al usuario a otra página diferente
document.URL = "http://nueva_pagina";
```

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

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

Los elementos de cada array del objeto `document` se pueden acceder mediante su índice numérico o mediante el nombre del elemento en la página HTML. Si se considera por ejemplo la siguiente página HTML:

```
<html>
  <head><title>Pagina de ejemplo</title></head>
  <body>
```

```
    <p>Primer parrafo de la pagina</p>
    <a href="otra_pagina.html">Un enlace</a>
    <img src="logo.jpg" name="logotipo"/>
    <form method="post" name="consultas">
      <input type="text" name="id" />
      <input type="submit" value="Enviar">
    </form>
  </body>
</html>
```

Para acceder a los elementos de la página se pueden emplear las funciones DOM o los objetos de BOM:

- Párrafo: `document.getElementsByTagName("p")`
- Enlace: `document.links[0]`
- Imagen: `document.images[0]` o `document.images["logotipo"]`
- Formulario: `document.forms[0]` o `document.forms["consultas"]`

Una vez obtenida la referencia al elemento, se puede acceder al valor de sus atributos HTML utilizando las propiedades de DOM. De esta forma, el método del formulario se obtiene mediante `document.forms["consultas"].method` y la ruta de la imagen es `document.images[0].src`.

El objeto `location` es uno de los objetos más útiles del BOM. Debido a la falta de estandarización, `location` es una propiedad tanto del objeto `window` como del objeto `document`.

El objeto `location` representa la URL de la página HTML que se muestra en la ventana del navegador y proporciona varias propiedades útiles para el manejo de la URL:

`hash`	El contenido de la URL que se encuentra después del signo `#` (para los enlaces de las anclas) `http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion` `hash` = `#seccion`
`host`	El nombre del servidor `http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion` `host` = `www.ejemplo.com`

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

De todas las propiedades, la más utilizada es `location.href`, que permite obtener o establecer la dirección de la página que se muestra en la ventana del navegador.

Además de las propiedades de la tabla anterior, el objeto `location` contiene numerosos métodos y funciones. Algunos de los métodos más útiles son los siguientes:

```
// Método assign()
location.assign("http://www.ejemplo.com");  // Equivalente a
location.href = "http://www.ejemplo.com"
```

```
// Método replace()
location.replace("http://www.ejemplo.com");
// Similar a assign(), salvo que se borra la página actual del array
history del navegador

// Método reload()
location.reload(true);
/* Recarga la página. Si el argumento es true, se carga la página desde
el servidor. Si es false, se carga desde la cache del navegador */
```

El objeto `navigator` es uno de los primeros objetos que incluyó el BOM y permite obtener información sobre el propio navegador. En Internet Explorer, el objeto `navigator` también se puede acceder a través del objeto `clientInformation`.

Aunque es uno de los objetos menos estandarizados, algunas de sus propiedades son comunes en casi todos los navegadores. A continuación se muestran algunas de esas propiedades:

`appCodeName`	Cadena que representa el nombre del navegador (normalmente es `Mozilla`)
`appName`	Cadena que representa el nombre oficial del navegador
`appMinorVersion`	(Sólo Internet Explorer) Cadena que representa información extra sobre la versión del navegador
`appVersion`	Cadena que representa la versión del navegador
`browserLanguage`	Cadena que representa el idioma del navegador
`cookieEnabled`	Boolean que indica si las cookies están habilitadas
`cpuClass`	(Sólo Internet Explorer) Cadena que representa el tipo de CPU del usuario ("x86", "68K", "PPC", "Alpha", "Other")
`javaEnabled`	Boolean que indica si Java está habilitado
`language`	Cadena que representa el idioma del navegador

`mimeTypes`	Array de los tipos MIME registrados por el navegador
`onLine`	(Sólo Internet Explorer) Boolean que indica si el navegador está conectado a Internet
`oscpu`	(Sólo Firefox) Cadena que representa el sistema operativo o la CPU
`platform`	Cadena que representa la plataforma sobre la que se ejecuta el navegador
`plugins`	Array con la lista de plugins instalados en el navegador
`preference()`	(Sólo Firefox) Método empleado para establecer preferencias en el navegador
`product`	Cadena que representa el nombre del producto (normalmente, es `Gecko`)
`productSub`	Cadena que representa información adicional sobre el producto (normalmente, la versión del motor Gecko)
`securityPolicy`	Sólo Firefox
`systemLanguage`	(Sólo Internet Explorer) Cadena que representa el idioma del sistema operativo
`userAgent`	Cadena que representa la cadena que el navegador emplea para identificarse en los servidores
`userLanguage`	(Sólo Explorer) Cadena que representa el idioma del sistema operativo
`userProfile`	(Sólo Explorer) Objeto que permite acceder al perfil del usuario

El objeto `navigator` se emplea habitualmente para detectar el tipo y/o versión del navegador en las aplicaciones cuyo código difiere para cada navegador. Además, se emplea para detectar si el navegador tiene habilitadas las cookies y Java y también para comprobar los plugins disponibles en el navegador.

El objeto screen se utiliza para obtener información sobre la pantalla del usuario. Uno de los datos más importantes que proporciona el objeto screen es la resolución del monitor en el que se están visualizando las páginas.

Las siguientes propiedades están disponibles en el objeto screen:

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

La altura/anchura de pantalla disponible para las ventanas es menor que la altura/anchura total de la pantalla, ya que se tiene en cuenta el tamaño de los elementos del sistema operativo como por ejemplo la barra de tareas y los bordes de las ventanas del navegador.

Además de la elaboración de estadísticas de los equipos de los usuarios, las propiedades del objeto `screen` se utilizan por ejemplo para determinar cómo y cuanto se puede redimensionar una ventana y para colocar una ventana centrada en la pantalla del usuario.

El siguiente ejemplo redimensiona una nueva ventana al tamaño máximo posible según la pantalla del usuario:

```
window.moveTo(0, 0);
window.resizeTo(screen.availWidth, screen.availHeight);
```

Esta página se ha dejado vacía a propósito

Capítulo 15

En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio a fin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es el de la programación basada en eventos. Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JavaScript permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos. Los eventos de JavaScript permiten la interacción entre las aplicaciones JavaScript y los usuarios. Cada vez que se pulsa un botón, se produce un evento. Cada vez que se pulsa una tecla, también se produce un evento. No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que por ejemplo, cada vez que se carga una página, también se produce un evento.

El nivel 1 de DOM no incluye especificaciones relativas a los eventos JavaScript. El nivel 2 de DOM incluye ciertos aspectos relacionados con los eventos y el nivel 3 de DOM incluye la especificación completa de los eventos de JavaScript. Desafortunadamente, la especificación de nivel 3 de DOM se publicó en el año 2004, más de diez años después de que los primeros navegadores incluyeran los eventos.

Por este motivo, muchas de las propiedades y métodos actuales relacionados con los eventos son incompatibles con los de DOM. De hecho,

navegadores como Internet Explorer tratan los eventos siguiendo su propio modelo incompatible con el estándar.

El modelo simple de eventos se introdujo en la versión 4 del estándar HTML y se considera parte del nivel más básico de DOM. Aunque sus características son limitadas, es el único modelo que es compatible con todos los navegadores y por tanto, el único que permite crear aplicaciones que funcionan de la misma manera en todos los navegadores.

Cada elemento XHTML tiene definida su propia lista de posibles eventos que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos XHTML y un mismo elemento XHTML puede tener asociados diferentes eventos.

El nombre de los eventos se construye mediante el prefijo `on`, seguido del nombre en inglés de la acción asociada al evento. Así, el evento de pinchar un elemento con el ratón se denomina `onclick` y el evento asociado a la acción de mover el ratón se denomina `onmousemove`.

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

`onblur`	Un elemento pierde el foco	`<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>`
`onchange`	Un elemento ha sido modificado	`<input>`, `<select>`, `<textarea>`
`onclick`	Pulsar y soltar el ratón	Todos los elementos
`ondblclick`	Pulsar dos veces seguidas con el ratón	Todos los elementos
`onfocus`	Un elemento obtiene el foco	`<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>`
`onkeydown`	Pulsar una tecla y no soltarla	Elementos de formulario y `<body>`

onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón "sale" del elemento	Todos los elementos
onmouseover	El ratón "entra" en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos
onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

Los eventos más utilizados en las aplicaciones web tradicionales son `onload` para esperar a que se cargue la página por completo, los eventos `onclick`, `onmouseover`, `onmouseout` para controlar el ratón y `onsubmit` para controlar el envío de los formularios.

Algunos eventos de la tabla anterior (`onclick`, `onkeydown`, `onkeypress`, `onreset`, `onsubmit`) permiten evitar la "acción por defecto" de ese evento. Más adelante se muestra en detalle este comportamiento, que puede resultar muy útil en algunas técnicas de programación.

Las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar por ejemplo sobre un bo-

tón de tipo `<input type="submit">` se desencadenan los eventos `onmousedown`, `onclick`, `onmouseup` y `onsubmit` de forma consecutiva.

Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede responder ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *manejador de eventos* (*event handlers* en inglés) y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:

- Manejadores como atributos de los elementos XHTML.
- Manejadores como funciones JavaScript externas.
- Manejadores "semánticos".

Se trata del método más sencillo y a la vez *menos profesional* de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento. En este caso, el código se incluye en un atributo del propio elemento XHTML. En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pinche con el ratón sobre un botón:

```
<input type="button" value="Pinchame y verás" onclick="alert('Gracias
por pinchar');" />
```

En este método, se definen atributos XHTML con el mismo nombre que los eventos que se quieren manejar. El ejemplo anterior sólo quiere controlar el evento de pinchar con el ratón, cuyo nombre es `onclick`. Así, el elemento XHTML para el que se quiere definir este evento, debe incluir un atributo llamado `onclick`.

El contenido del atributo es una cadena de texto que contiene todas las instrucciones JavaScript que se ejecutan cuando se produce el evento. En este caso, el código JavaScript es muy sencillo (`alert('Gracias por pinchar');`), ya que solamente se trata de mostrar un mensaje.

En este otro ejemplo, cuando el usuario pincha sobre el elemento `<div>` se muestra un mensaje y cuando el usuario pasa el ratón por encima del elemento, se muestra otro mensaje:

```
<div onclick="alert('Has pinchado con el ratón');"
onmouseover="alert('Acabas de pasar el ratón por encima');">
    Puedes pinchar sobre este elemento o simplemente pasar el ratón por
encima
</div>
```

Este otro ejemplo incluye una de las instrucciones más utilizadas en las aplicaciones JavaScript más antiguas:

```
<body onload="alert('La página se ha cargado completamente');">
 ...
</body>
```

El mensaje anterior se muestra después de que la página se haya cargado completamente, es decir, después de que se haya descargado su código HTML, sus imágenes y cualquier otro objeto incluido en la página.

El evento `onload` es uno de los más utilizados ya que, como se vio en el capítulo de DOM, las funciones que permiten acceder y manipular los nodos del árbol DOM solamente están disponibles cuando la página se ha cargado completamente.

JavaScript define una variable especial llamada `this` que se crea automáticamente y que se emplea en algunas técnicas avanzadas de programación. En los eventos, se puede utilizar la variable `this` para referirse al elemento XHTML que ha provocado el evento. Esta variable es muy útil para ejemplos como el siguiente:

Cuando el usuario pasa el ratón por encima del `<div>`, el color del borde se muestra de color negro. Cuando el ratón sale del `<div>`, se vuelve a mostrar el borde con el color gris claro original.

```
<div id="contenidos" style="width:150px; height:60px; border:thin solid
silver">
  Sección de contenidos...
</div>
```

Si no se utiliza la variable `this`, el código necesario para modificar el color de los bordes, sería el siguiente:

```
<div id="contenidos" style="width:150px; height:60px; border:thin solid
silver"
onmouseover="document.getElementById('contenidos').style.borderColor='black';"
onmouseout="document.getElementById('contenidos').style.borderColor='silver';">
    Sección de contenidos...
</div>
```

El código anterior es demasiado largo y demasiado propenso a cometer errores. Dentro del código de un evento, JavaScript crea automáticamente la variable `this`, que hace referencia al elemento XHTML que ha provocado el evento. Así, el ejemplo anterior se puede reescribir de la siguiente manera:

```
<div id="contenidos" style="width:150px; height:60px; border:thin solid
silver" onmouseover="this.style.borderColor='black';"
onmouseout="this.style.borderColor='silver';">
    Sección de contenidos...
</div>
```

El código anterior es mucho más compacto, más fácil de leer y de escribir y sigue funcionando correctamente aunque se modifique el valor del atributo id del `<div>`.

La definición de manejadores de eventos en los atributos XHTML es un método sencillo pero poco aconsejable para tratar con los eventos en JavaScript. El principal inconveniente es que se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para los casos más sencillos.

Cuando el código de la función manejadora es más complejo, como por ejemplo la validación de un formulario, es aconsejable agrupar todo el código JavaScript en una función externa que se invoca desde el código XHTML cuando se produce el evento.

De esta forma, el siguiente ejemplo:

```
<input type="button" value="Pinchame y verás" onclick="alert('Gracias
por pinchar');" />
```

Se puede transformar en:

```
function muestraMensaje() {
  alert('Gracias por pinchar');
}

<input type="button" value="Pinchame y verás" onclick="muestraMensaje()"
/>
```

En las funciones externas no es posible utilizar la variable `this` de la misma forma que en los manejadores insertados en los atributos XHTML. Por tanto, es necesario pasar la variable `this` como parámetro a la función manejadora:

```
function resalta(elemento) {
  switch(elemento.style.borderColor) {
    case 'silver':
    case 'silver silver silver silver':
    case '#c0c0c0':
      elemento.style.borderColor = 'black';
      break;
    case 'black':
    case 'black black black black':
    case '#000000':
      elemento.style.borderColor = 'silver';
      break;
  }
}

<div style="padding: .2em; width: 150px; height: 60px; border: thin
solid silver" onmouseover="resalta(this)" onmouseout="resalta(this)">
  Sección de contenidos...
</div>
```

En el ejemplo anterior, a la función externa se le pasa el parámetro `this`, que dentro de la función se denomina `elemento`. Al pasar `this` como parámetro, es posible acceder de forma directa desde la función externa a las propiedades del elemento que ha provocado el evento.

Por otra parte, el ejemplo anterior se complica por la forma en la que los distintos navegadores almacenan el valor de la propiedad `borderColor`. Mientras que Firefox almacena (en caso de que los cuatro bordes coincidan en color) el valor simple `black`, Internet Explorer lo almacena como

`black black black black` y Opera almacena su representación hexadecimal `#000000`.

Utilizar los atributos XHTML o las funciones externas para añadir manejadores de eventos tiene un grave inconveniente: "ensucian" el código XHTML de la página.

Como es conocido, al crear páginas web se recomienda separar los contenidos (XHTML) de la presentación (CSS). En lo posible, también se recomienda separar los contenidos (XHTML) de la programación (JavaScript). Mezclar JavaScript y XHTML complica excesivamente el código fuente de la página, dificulta su mantenimiento y reduce la semántica del documento final producido.

Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica consiste en asignar las funciones externas mediante las propiedades DOM de los elementos XHTML. Así, el siguiente ejemplo:

```
<input id="pinchable" type="button" value="Pinchame y verás"
onclick="alert('Gracias por pinchar');" />
Se puede transformar en:
```

```
function muestraMensaje() {
  alert('Gracias por pinchar');
}
document.getElementById("pinchable").onclick = muestraMensaje;
```

```
<input id="pinchable" type="button" value="Pinchame y verás" />
```

El código XHTML resultante es muy "limpio", ya que no se mezcla con el código JavaScript. La técnica de los manejadores semánticos consiste en:

1. Asignar un identificador único al elemento XHTML mediante el atributo id.
2. Crear una función de JavaScript encargada de manejar el evento.
3. Asignar la función a un evento concreto del elemento XHTML mediante DOM.

Otra ventaja adicional de esta técnica es que las funciones externas pueden utilizar la variable `this` referida al elemento que original el evento.

Asignar la función manejadora mediante DOM es un proceso que requiere una explicación detallada. En primer lugar, se obtiene la referencia del elemento al que se va a asignar el manejador:

```
document.getElementById("pinchable");
```

A continuación, se asigna la función externa al evento deseado mediante una propiedad del elemento con el mismo nombre del evento:

```
document.getElementById("pinchable").onclick = ...
```

Por último, se asigna la función externa. Como ya se ha comentado en capítulos anteriores, lo más importante (y la causa más común de errores) es indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

```
document.getElementById("pinchable").onclick = muestraMensaje;
```

Si se añaden los paréntesis al final, en realidad se está invocando la función y asignando el valor devuelto por la función al evento `onclick` de elemento.

El único inconveniente de este método es que los manejadores se asignan mediante las funciones DOM, que solamente se pueden utilizar después de que la página se ha cargado completamente. De esta forma, para que la asignación de los manejadores no resulte errónea, es necesario asegurarse de que la página ya se ha cargado.

Una de las formas más sencillas de asegurar que cierto código se va a ejecutar después de que la página se cargue por completo es utilizar el evento `onload`:

```
window.onload = function() {
  document.getElementById("pinchable").onclick = muestraMensaje;
}
```

La técnica anterior utiliza una función anónima para asignar algunas instrucciones al evento `onload` de la página (en este caso se ha establecido mediante el objeto `window`). De esta forma, para asegurar que cierto código se va a ejecutar después de que la página se haya cargado, sólo es necesario incluirlo en el interior de la siguiente construcción:

```
window.onload = function() {
  ...
}
```

Además de los eventos básicos que se han visto, los navegadores incluyen un mecanismo relacionado llamado flujo de eventos o *"event flow"*. El flujo de eventos permite que varios elementos diferentes puedan responder a un mismo evento.

Si en una página HTML se define un elemento `<div>` con un botón en su interior, cuando el usuario pulsa sobre el botón, el navegador permite asignar una función de respuesta al botón, otra función de respuesta al `<div>` que lo contiene y otra función de respuesta a la página completa. De esta forma, un solo evento (la pulsación de un botón) provoca la respuesta de tres elementos de la página (incluyendo la propia página).

El orden en el que se ejecutan los eventos asignados a cada elemento de la página es lo que constituye el flujo de eventos. Además, existen muchas diferencias en el flujo de eventos de cada navegador.

En este modelo de flujo de eventos, el orden que se sigue es desde el elemento más específico hasta el elemento menos específico.

En los próximos ejemplos se emplea la siguiente página HTML:

```
<html onclick="procesaEvento()">
  <head><title>Ejemplo de flujo de eventos</title></head>
  <body onclick="procesaEvento()">
    <div onclick="procesaEvento()">Pincha aqui</div>
  </body>
</html>
```

Cuando se pulsa sobre el texto "Pincha aquí" que se encuentra dentro del `<div>`, se ejecutan los siguientes eventos en el orden que muestra el siguiente esquema:

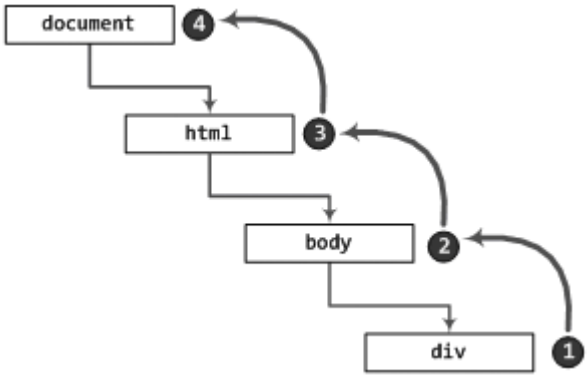


Figura 15.1 Esquema del funcionamiento del "event bubbling"

El primer evento que se tiene en cuenta es el generado por el `<div>` que contiene el mensaje. A continuación el navegador recorre los ascendentes del elemento hasta que alcanza el nivel superior, que es el elemento `document`.

Este modelo de flujo de eventos es el que incluye el navegador Internet Explorer. Los navegadores de la familia Mozilla (por ejemplo Firefox) también soportan este modelo, pero ligeramente modificado. El anterior ejemplo en un navegador de la familia Mozilla presenta el siguiente flujo de eventos:

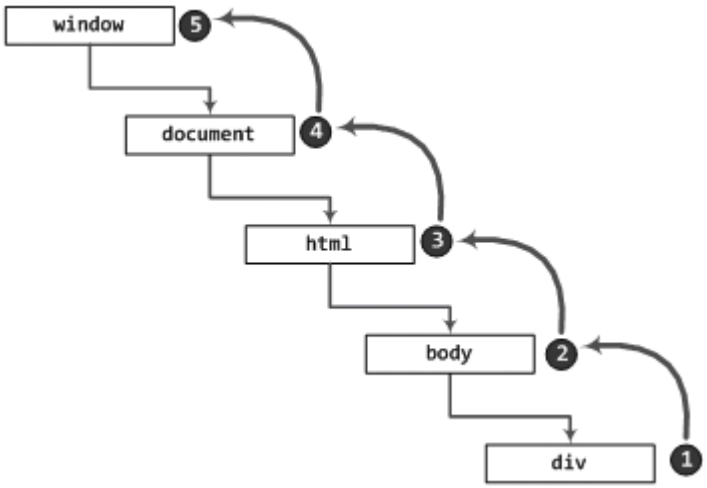


Figura 15.2 Esquema del funcionamiento del "event bubbling" en los navegadores de Mozilla

Aunque el objeto `window` no es parte del DOM, el flujo de eventos implementado por Mozilla recorre los ascendentes del elemento hasta el mismo objeto `window`, añadiendo por tanto un evento más al modelo de Internet Explorer.

En ese otro modelo, el flujo de eventos se define desde el elemento menos específico hasta el elemento más específico. En otras palabras, el mecanismo definido es justamente el contrario al "event bubbling". Este modelo lo utilizaba el desaparecido navegador Netscape Navigator 4.0.

El flujo de eventos definido en la especificación DOM soporta tanto el *bubbling* como el *capturing*, pero el *"event capturing"* se ejecuta en primer lugar. Los dos flujos de eventos recorren todos los objetos DOM desde el objeto `document` hasta el elemento más específico y viceversa. Además, la mayoría de navegadores que implementan los estándares, continúan el flujo hasta el objeto `window`.

El flujo de eventos DOM del ejemplo anterior se muestra a continuación:

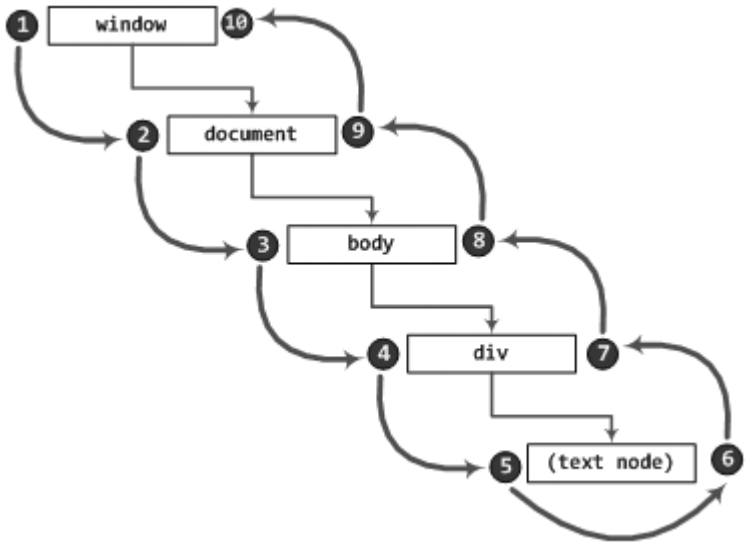


Figura 15.3 Esquema del flujo de eventos del modelo DOM

El elemento más específico del flujo de eventos no es el `<div>` que desencadena la ejecución de los eventos, sino el nodo de tipo `TextNode` que contiene el `<div>`. El hecho de combinar los dos flujos de eventos, provoca que el nodo más específico pueda ejecutar dos eventos de forma consecutiva.

En las secciones anteriores se introdujo el concepto de "event handler" o manejador de eventos, que son las funciones que responden a los eventos que se producen. Además, se vieron tres formas de definir los manejadores de eventos para el modelo básico de eventos:

1. Código JavaScript dentro de un atributo del propio elemento HTML
2. Definición del evento en el propio elemento HTML pero el manejador es una función externa
3. Manejadores semánticos asignados mediante DOM sin necesidad de modificar el código HTML de la página

Cualquiera de estos tres modelos funciona correctamente en todos los navegadores disponibles en la actualidad. Las diferencias entre navegadores surgen cuando se define más de un manejador de eventos para un mismo evento de un elemento. La forma de asignar y *"desasignar"* manejadores múltiples depende completamente del navegador utilizado.

La especificación DOM define otros dos métodos similares a los disponibles para Internet Explorer y denominados `addEventListener()` y `removeEventListener()` para asociar y desasociar manejadores de eventos.

La principal diferencia entre estos métodos y los anteriores es que en este caso se requieren tres parámetros: el nombre del *"event listener"*, una referencia a la función encargada de procesar el evento y el tipo de flujo de eventos al que se aplica.

El primer argumento no es el nombre completo del evento como sucede en el modelo de Internet Explorer, sino que se debe eliminar el prefijo `on`. En otras palabras, si en Internet Explorer se utilizaba el nombre `onclick`, ahora se debe utilizar `click`.

Si el tercer parámetro es `true`, el manejador se emplea en la fase de *captura*. Si el tercer parámetro es `false`, el manejador se asocia a la fase de *bubbling*.

A continuación, se muestran los ejemplos anteriores empleando los métodos definidos por DOM:

```
function muestraMensaje() {
  alert("Has pulsado el ratón");
}
var elDiv = document.getElementById("div_principal");
elDiv.addEventListener("click", muestraMensaje, false);

// Más adelante se decide desasociar la función al evento
elDiv.removeEventListener("click", muestraMensaje, false);
```

Asociando múltiples funciones a un único evento:

```
function muestraMensaje() {
  alert("Has pulsado el ratón");
}

function muestraOtroMensaje() {
  alert("Has pulsado el ratón y por eso se muestran estos mensajes");
}

var elDiv = document.getElementById("div_principal");
elDiv.addEventListener("click", muestraMensaje, true);
elDiv.addEventListener("click", muestraOtroMensaje, true);
```

Si se asocia una función a un flujo de eventos determinado, esa función sólo se puede desasociar en el mismo tipo de flujo de eventos. Si se considera el siguiente ejemplo:

```
function muestraMensaje() {
  alert("Has pulsado el ratón");
}

var elDiv = document.getElementById("div_principal");
elDiv.addEventListener("click", muestraMensaje, false);

// Más adelante se decide desasociar la función al evento
elDiv.removeEventListener("click", muestraMensaje, true);
```

La última instrucción intenta desasociar la función `muestraMensaje` en el flujo de eventos de *capture*, mientras que al asociarla, se indicó el flujo de eventos de *bubbling*. Aunque la ejecución de la aplicación no se detiene y no se produce ningún error, la última instrucción no tiene ningún efecto.

Cuando se produce un evento, no es suficiente con asignarle una función responsable de procesar ese evento. Normalmente, la función que procesa el evento necesita información relativa al evento producido: la tecla que se ha pulsado, la posición del ratón, el elemento que ha producido el evento, etc.

El objeto `event` es el mecanismo definido por los navegadores para proporcionar toda esa información. Se trata de un objeto que se crea auto-

máticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento.

El estándar DOM especifica que el objeto `event` es el único parámetro que se debe pasar a las funciones encargadas de procesar los eventos. Por tanto, en los navegadores que siguen los estándares, se puede acceder al objeto `event` a través del array de los argumentos de la función:

```
elDiv.onclick = function() {
  var elEvento = arguments[0];
}
```

También es posible indicar el nombre argumento de forma explícita:

```
elDiv.onclick = function(event) {
  ...
}
```

El funcionamiento de los navegadores que siguen los estándares puede parecer "mágico", ya que en la declaración de la función se indica que tiene un parámetro, pero en la aplicación no se pasa ningún parámetro a esa función. En realidad, los navegadores que siguen los estándares crean automáticamente ese parámetro y lo pasan siempre a la función encargada de manejar el evento.

A pesar de que el mecanismo definido por los navegadores para el objeto `event` es similar, existen numerosas diferencias en cuanto las propiedades y métodos del objeto.

La siguiente tabla recoge las propiedades definidas para el objeto `event` en los navegadores que siguen los estándares:

`altKey`	Boolean	Devuelve `true` si se ha pulsado la tecla `ALT` y `false` en otro caso
`bubbles`	Boolean	Indica si el evento pertenece al flujo de eventos de bubbling

`button`	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: `0` – Ningún botón pulsado `1` – Se ha pulsado el botón `izquierdo2– Se ha pulsado el botón derecho3– Se pulsan a la vez el botón izquierdo y el derecho4– Se ha pulsado el botón central5– Se pulsan a la vez el botón izquierdo y el central6– Se pulsan a la vez el botón derecho y el central7` – Se pulsan a la vez los 3 botones
`cancelable`	Boolean	Indica si el evento se puede cancelar
`cancelBubble`	Boolean	Indica si se ha detenido el flujo de eventos de tipo *bubbling*
`charCode`	Número entero	El código unicode del carácter correspondiente a la tecla pulsada
`clientX`	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
`clientY`	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
`ctrlKey`	Boolean	Devuelve `true` si se ha pulsado la tecla `CTRL` y `false` en otro caso
`currentTarget`	Element	El elemento que es el objetivo del evento
`detail`	Número entero	El número de veces que se han pulsado los botones del ratón
`eventPhase`	Número entero	La fase a la que pertenece el evento: `0` – Fase capturing `1` – En el elemento destino `2` – Fase bubbling
`isChar`	Boolean	Indica si la tecla pulsada corresponde a un carácter
`keyCode`	Número entero	Indica el código numérico de la tecla pulsada

`metaKey`	Número entero	Devuelve `true` si se ha pulsado la tecla `META` y `false` en otro caso
`pageX`	Número entero	Coordenada X de la posición del ratón respecto de la página
`pageY`	Número entero	Coordenada Y de la posición del ratón respecto de la página
`preventDefault()`	Función	Se emplea para cancelar la acción predefinida del evento
`relatedTarget`	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
`screenX`	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
`screenY`	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
`shiftKey`	Boolean	Devuelve `true` si se ha pulsado la tecla `SHIFT` y `false` en otro caso
`stopPropagation()`	Función	Se emplea para detener el flujo de eventos de tipo bubbling
`target`	Element	El elemento que origina el evento
`timeStamp`	Número	La fecha y hora en la que se ha producido el evento
`type`	Cadena de texto	El nombre del evento

Al contrario de lo que sucede con Internet Explorer, la mayoría de propiedades del objeto `event` de DOM son de sólo lectura. En concreto, solamente las siguientes propiedades son de lectura y escritura: `altKey`, `button` y `keyCode`.

La tecla `META` es una tecla especial que se encuentra en algunos teclados de ordenadores muy antiguos. Actualmente, en los ordenadores tipo PC

se asimila a la tecla `Alt` o a la tecla de Windows, mientras que en los ordenadores tipo Mac se asimila a la tecla `Command`.

En ambos casos se utiliza la propiedad type para obtener el tipo de evento que se trata:

```
function procesaEvento(elEvento) {
  if(elEvento.type == "click") {
    alert("Has pulsado el raton");
  }
  else if(elEvento.type == "mouseover") {
    alert("Has movido el raton");
  }
}

elDiv.onclick = procesaEvento;
elDiv.onmouseover = procesaEvento;
```

Mientras que el manejador del evento incluye el prefijo `on` en su nombre, el tipo de evento devuelto por la propiedad `type` prescinde de ese prefijo. Por eso en el ejemplo anterior se compara su valor con `click` y `mouseover` y no con `onclick` y `onmouseover`.

Otra similitud es el uso de la propiedad `keyCode` para obtener el código correspondiente al carácter de la tecla que se ha pulsado. La tecla pulsada no siempre representa un carácter alfanumérico. Cuando se pulsa la tecla `ENTER` por ejemplo, se obtiene el código `13`. La barra espaciadora se corresponde con el código `32` y la tecla de borrado tiene un código igual a `8`.

Una forma más inmediata de comprobar si se han pulsado algunas teclas especiales, es utilizar las propiedades `shiftKey`, `altKey` y `ctrlKey`.

Para obtener la posición del ratón respecto de la parte visible de la ventana, se emplean las propiedades `clientX` y `clientY`. De la misma forma, para obtener la posición del puntero del ratón respecto de la pantalla completa, se emplean las propiedades `screenX` y `screenY`.

Una de las principales diferencias es la forma en la que se obtiene el elemento que origina el evento. Si un elemento `<div>` tiene asignado un evento `onclick`, al pulsar con el ratón el interior del `<div>` se origina un evento cuyo objetivo es el elemento `<div>`.

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

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

Otra diferencia importante es la relativa a la obtención del carácter correspondiente a la tecla pulsada. Cada tecla pulsada tiene asociados dos códigos diferentes: el primero es el código de la tecla que ha sido presionada y el otro código es el que se refiere al carácter pulsado.

El primer código es un código de tecla interno para JavaScript. El segundo código coincide con el código ASCII del carácter. De esta forma, la letra `a` tiene un código interno igual a `65` y un código ASCII de `97`. Por otro lado, la letra `A` tiene un código interno también de `65` y un código ASCII de `95`.

En Internet Explorer, el contenido de la propiedad `keyCode` depende de cada evento. En los eventos de "pulsación de teclas" (`onkeyup` y `onkeydown`) su valor es igual al código interno. En los eventos de "escribir con teclas" (`onkeypress`) su valor es igual al código ASCII del carácter pulsado.

Por el contrario, en los navegadores que siguen los estándares la propiedad `keyCode` es igual al código interno en los eventos de "pulsación de teclas" (`onkeyup` y `onkeydown`) y es igual a `0` en los eventos de "escribir con teclas" (`onkeypress`).

En la práctica, esto supone que en los eventos `onkeyup` y `onkeydown` se puede utilizar la misma propiedad en todos los navegadores:

```
function manejador(elEvento) {
  var evento = elEvento || window.event;
  alert("["+evento.type+"] El código de la tecla pulsada es " +
evento.keyCode);
}
```

```
document.onkeyup = manejador;
document.onkeydown = manejador;
```

En este caso, si se carga la página en cualquier navegador y se pulsa por ejemplo la tecla a, se muestra el siguiente mensaje:

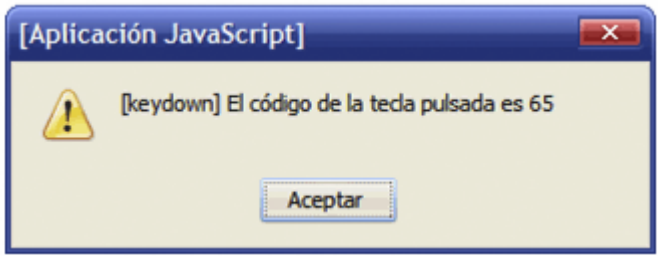


Figura 15.4 Mensaje mostrado en el navegador Firefox

La gran diferencia se produce al intentar obtener el carácter que se ha pulsado, en este caso la letra `a`. Para obtener la letra, en primer lugar se debe obtener su código ASCII. Como se ha comentado, en Internet Explorer el valor de la propiedad `keyCode` en el evento `onkeypress` es igual al carácter ASCII:

```
function manejador() {
  var evento = window.event;

  // Internet Explorer
  var codigo = evento.keyCode;
}

document.onkeypress = manejador;
```

Sin embargo, en los navegadores que no son Internet Explorer, el código anterior es igual a `0` para cualquier tecla pulsada. En estos navegadores que siguen los estándares, se debe utilizar la propiedad `charCode`, que devuelve el código de la tecla pulsada, pero solo para el evento `onkeypress`:

```
function manejador(elEvento) {
  var evento = elEvento;

  // Navegadores que siguen los estándares
  var codigo = evento.charCode;
}

document.onkeypress = manejador;
```

Una vez obtenido el código en cada navegador, se debe utilizar la función `String.fromCharCode()` para obtener el carácter cuyo código ASCII se pasa como parámetro. Por tanto, la solución completa para obtener la tecla pulsada en cualquier navegador es la siguiente:

```
function manejador(elEvento) {
  var evento = elEvento || window.event;
  var codigo = evento.charCode || evento.keyCode;
  var caracter = String.fromCharCode(codigo);
}

document.onkeypress = manejador;
```

Una de las propiedades más interesantes es la posibilidad de impedir que se complete el comportamiento normal de un evento. En otras palabras, con JavaScript es posible no mostrar ningún carácter cuando se pulsa una tecla, no enviar un formulario después de pulsar el botón de envío, no cargar ninguna página al pulsar un enlace, etc. El método avanzado de impedir que un evento ejecute su acción asociada depende de cada navegador:

```
// Navegadores que siguen los estandares
elEvento.preventDefault();
```

En el modelo básico de eventos también es posible impedir el comportamiento por defecto de algunos eventos. Si por ejemplo en un elemento `<textarea>` se indica el siguiente manejador de eventos:

```
<textarea onkeypress="return false;"></textarea>
```

En el `<textarea>` anterior no será posible escribir ningún carácter, ya que el manejador de eventos devuelve `false` y ese es el valor necesario para impedir que se termine de ejecutar el evento y por tanto para evitar que la letra se escriba.

Así, es posible definir manejadores de eventos que devuelvan `true` o `false` en función de algunos parámetros. Por ejemplo se puede diseñar un limitador del número de caracteres que se pueden escribir en un `<textarea>`:

```
function limita(maximoCaracteres) {
  var elemento = document.getElementById("texto");
  if(elemento.value.length >= maximoCaracteres ) {
```

```
    return false;
  }
  else {
    return true;
  }
}
```

```
<textarea id="texto" onkeypress="return limita(100);"></textarea>
```

El funcionamiento del ejemplo anterior se detalla a continuación:

1. Se utiliza el evento `onkeypress` para controlar si la tecla se escribe o no.
2. En el manejador del evento se devuelve el valor devuelto por la función externa `limita()` a la que se pasa como parámetro el valor `100`.
3. Si el valor devuelto por `limita()` es `true`, el evento se produce de forma normal y el carácter se escribe en el `<textarea>`. Si el valor devuelto por `limita()` es `false`, el evento no se produce y por tanto el carácter no se escribe en el `<textarea>`.
4. La función `limita()` devuelve `true` o `false` después de comprobar si el número de caracteres del `<textarea>` es superior o inferior al máximo número de caracteres que se le ha pasado como parámetro.

El objeto `event` también permite detener completamente la ejecución del flujo normal de eventos:

```
// Navegadores que siguen los estandares
elEvento.stopPropagation();
```

Al detener el flujo de eventos pendientes, se invalidan y no se ejecutan los eventos que restan desde ese momento hasta que se recorren todos los elementos pendientes hasta el elemento `window`.

La lista completa de eventos que se pueden generar en un navegador se puede dividir en cuatro grandes grupos. La especificación de DOM define los siguientes grupos:

- Eventos de ratón: se originan cuando el usuario emplea el ratón para realizar algunas acciones.

- Eventos de teclado: se originan cuando el usuario pulsa sobre cualquier tecla de su teclado.
- Eventos HTML: se originan cuando se producen cambios en la ventana del navegador o cuando se producen ciertas interacciones entre el cliente y el servidor.
- Eventos DOM: se originan cuando se produce un cambio en la estructura DOM de la página. También se denominan "eventos de mutación".

Los eventos de ratón son, con mucha diferencia, los más empleados en las aplicaciones web. Los eventos que se incluyen en esta clasificación son los siguientes:

`click`	Se produce cuando se pulsa el botón izquierdo del ratón. También se produce cuando el foco de la aplicación está situado en un botón y se pulsa la tecla `ENTER`
`dblclick`	Se produce cuando se pulsa dos veces el botón izquierdo del ratón
`mousedown`	Se produce cuando se pulsa cualquier botón del ratón
`mouseout`	Se produce cuando el puntero del ratón se encuentra en el interior de un elemento y el usuario mueve el puntero a un lugar fuera de ese elemento
`mouseover`	Se produce cuando el puntero del ratón se encuentra fuera de un elemento y el usuario mueve el puntero hacia un lugar en el interior del elemento
`mouseup`	Se produce cuando se suelta cualquier botón del ratón que haya sido pulsado
`mousemove`	Se produce (de forma continua) cuando el puntero del ratón se encuentra sobre un elemento

Todos los elementos de las páginas soportan los eventos de la tabla anterior.

El objeto event contiene las siguientes propiedades para los eventos de ratón:

- Las coordenadas del ratón (todas las coordenadas diferentes relativas a los distintos elementos)
- La propiedad `type`
- La propiedad `srcElement` (Internet Explorer) o `target` (DOM)
- Las propiedades `shiftKey`, `ctrlKey`, `altKey` y `metaKey` (sólo DOM)
- La propiedad `button` (sólo en los eventos `mousedown`, `mousemove`, `mouseout`, `mouseover` y `mouseup`)

Los eventos `mouseover` y `mouseout` tienen propiedades adicionales. Internet Explorer define la propiedad `fromElement`, que hace referencia al elemento desde el que el puntero del ratón se ha movido y `toElement` que es el elemento al que el puntero del ratón se ha movido. De esta forma, en el evento `mouseover`, la propiedad `toElement` es idéntica a `srcElement` y en el evento `mouseout`, la propiedad `fromElement` es idéntica a `srcElement`.

En los navegadores que soportan el estándar DOM, solamente existe una propiedad denominada `relatedTarget`. En el evento `mouseout`, `relatedTarget` apunta al elemento al que se ha movido el ratón. En el evento `mouseover`, `relatedTarget` apunta al elemento desde el que se ha movido el puntero del ratón.

Cuando se pulsa un botón del ratón, la secuencia de eventos que se produce es la siguiente: `mousedown`, `mouseup`, `click`. Por tanto, la secuencia de eventos necesaria para llegar al doble click llega a ser tan compleja como la siguiente: `mousedown`, `mouseup`, `click`, `mousedown`, `mouseup`, `click`, `dblclick`.

Los eventos que se incluyen en esta clasificación son los siguientes:

`keydown`	Se produce cuando se pulsa cualquier tecla del teclado. También se produce de forma continua si se mantiene pulsada la tecla

keypress	Se produce cuando se pulsa una tecla correspondiente a un carácter alfanumérico (no se tienen en cuenta telas como SHIFT, ALT, etc.). También se produce de forma continua si se mantiene pulsada la tecla
keyup	Se produce cuando se suelta cualquier tecla pulsada

El objeto event contiene las siguientes propiedades para los eventos de teclado:

- La propiedad keyCode
- La propiedad charCode (sólo DOM)
- La propiedad srcElement (Internet Explorer) o target (DOM)
- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)

Cuando se pulsa una tecla correspondiente a un carácter alfanumérico, se produce la siguiente secuencia de eventos: keydown, keypress, keyup. Cuando se pulsa otro tipo de tecla, se produce la siguiente secuencia de eventos: keydown, keyup. Si se mantiene pulsada la tecla, en el primer caso se repiten de forma continua los eventos keydown y keypress y en el segundo caso, se repite el evento keydown de forma continua.

Los eventos HTML definidos se recogen en la siguiente tabla:

load	Se produce en el objeto window cuando la página se carga por completo. En el elemento <img> cuando se carga por completo la imagen. En el elemento` cuando se carga el objeto
unload	Se produce en el objeto window cuando la página desaparece por completo (al cerrar la ventana del navegador por ejemplo). En el elemento <object> cuando desaparece el objeto.
abort	Se produce en un elemento <object> cuando el usuario detiene la descarga del elemento antes de que haya terminado

`error`	Se produce en el objeto window cuando se produce un error de JavaScript. En el elemento `<img>` cuando la imagen no se ha podido cargar por completo y en el elemento `<object>` cuando el elemento no se carga correctamente
`select`	Se produce cuando se seleccionan varios caracteres de un cuadro de texto (`<input>` y `<textarea>`)
`change`	Se produce cuando un cuadro de texto (`<input>` y `<textarea>`) pierde el foco y su contenido ha variado. También se produce cuando varía el valor de un elemento `<select>`
`submit`	Se produce cuando se pulsa sobre un botón de tipo submit (`<input type="submit">`)
`reset`	Se produce cuando se pulsa sobre un botón de tipo reset (`<input type="reset">`)
`resize`	Se produce en el objeto window cuando se redimensiona la ventana del navegador
`scroll`	Se produce en cualquier elemento que tenga una barra de scroll, cuando el usuario la utiliza. El elemento `<body>` contiene la barra de scroll de la página completa
`focus`	Se produce en cualquier elemento (incluido el objeto window) cuando el elemento obtiene el foco
`blur`	Se produce en cualquier elemento (incluido el objeto window) cuando el elemento pierde el foco

Uno de los eventos más utilizados es el evento `load`, ya que todas las manipulaciones que se realizan mediante DOM requieren que la página esté cargada por completo y por tanto, el árbol DOM se haya construido completamente.

El elemento `<body>` define las propiedades `scrollLeft` y `scrollTop` que se pueden emplear junto con el evento `scroll`.

Aunque los eventos de este tipo son parte de la especificación oficial de DOM, aún no han sido implementados en todos los navegadores. La siguiente tabla recoge los eventos más importantes de este tipo:

`DOMSubtreeModified`	Se produce cuando se añaden o eliminan nodos en el subárbol de un documento o elemento
`DOMNodeInserted`	Se produce cuando se añade un nodo como hijo de otro nodo
`DOMNodeRemoved`	Se produce cuando se elimina un nodo que es hijo de otro nodo
`DOMNodeRemovedFromDocument`	Se produce cuando se elimina un nodo del documento
`DOMNodeInsertedIntoDocument`	Se produce cuando se añade un nodo al documento

Ejercicio 15

Ver enunciado (#ej15)

Ejercicio 16

Ver enunciado (#ej16)

Ejercicio 17

Ver enunciado (#ej17)

Esta página se ha dejado vacía a propósito

Capítulo 16

La programación de aplicaciones que contienen formularios web siempre ha sido una de las tareas fundamentales de JavaScript. De hecho, una de las principales razones por las que se inventó el lenguaje de programación JavaScript fue la necesidad de validar los datos de los formularios directamente en el navegador del usuario. De esta forma, se evitaba recargar la página cuando el usuario cometía errores al rellenar los formularios.

No obstante, la aparición de las aplicaciones AJAX ha relevado al tratamiento de formularios como la principal actividad de JavaScript. Ahora, el principal uso de JavaScript es el de las comunicaciones asíncronas con los servidores y el de la manipulación dinámica de las aplicaciones. De todas formas, el manejo de los formularios sigue siendo un requerimiento imprescindible para cualquier programador de JavaScript.

JavaScript dispone de numerosas propiedades y funciones que facilitan la programación de aplicaciones que manejan formularios. En primer lugar, cuando se carga una página web, el navegador crea automáticamente un array llamado `forms` y que contiene la referencia a todos los formularios de la página.

Para acceder al array `forms`, se utiliza el objeto `document`, por lo que `document.forms` es el array que contiene todos los formularios de la página. Como se trata de un array, el acceso a cada formulario se realiza con la

misma sintaxis de los arrays. La siguiente instrucción accede al primer formulario de la página:

```
document.forms[0];
```

Además del array de formularios, el navegador crea automáticamente un array llamado `elements` por cada uno de los formularios de la página. Cada array `elements` contiene la referencia a todos los elementos (cuadros de texto, botones, listas desplegables, etc.) de ese formulario. Utilizando la sintaxis de los arrays, la siguiente instrucción obtiene el primer elemento del primer formulario de la página:

```
document.forms[0].elements[0];
```

La sintaxis de los arrays no siempre es tan concisa. El siguiente ejemplo muestra cómo obtener directamente el último elemento del primer formulario de la página:

```
document.forms[0].elements[document.forms[0].elements.length-1];
```

Aunque esta forma de acceder a los formularios es rápida y sencilla, tiene un inconveniente muy grave. ¿Qué sucede si cambia el diseño de la página y en el código HTML se cambia el orden de los formularios originales o se añaden nuevos formularios? El problema es que *"el primer formulario de la página"* ahora podría ser otro formulario diferente al que espera la aplicación.

En un entorno tan cambiante como el diseño web, es muy difícil confiar en que el orden de los formularios se mantenga estable en una página web. Por este motivo, siempre debería evitarse el acceso a los formularios de una página mediante el array `document.forms`.

Una forma de evitar los problemas del método anterior consiste en acceder a los formularios de una página a través de su nombre (atributo `name`) o a través de su atributo `id`. El objeto `document` permite acceder directamente a cualquier formulario mediante su atributo `name`:

```
var formularioPrincipal = document.formulario;
var formularioSecundario = document.otro_formulario;
```

```
<form name="formulario" >
  ...
</form>
```

```
<form name="otro_formulario" >
  ...
</form>
```

Accediendo de esta forma a los formularios de la página, el script funciona correctamente aunque se reordenen los formularios o se añadan nuevos formularios a la página. Los elementos de los formularios también se pueden acceder directamente mediante su atributo name:

```
var formularioPrincipal = document.formulario;
var primerElemento = document.formulario.elemento;
```

```
<form name="formulario">
  <input type="text" name="elemento" />
</form>
```

Obviamente, también se puede acceder a los formularios y a sus elementos utilizando las funciones DOM de acceso directo a los nodos. El siguiente ejemplo utiliza la habitual función `document.getElementById()` para acceder de forma directa a un formulario y a uno de sus elementos:

```
var formularioPrincipal = document.getElementById("formulario");
var primerElemento = document.getElementById("elemento");
```

```
<form name="formulario" id="formulario" >
  <input type="text" name="elemento" id="elemento" />
</form>
```

Independientemente del método utilizado para obtener la referencia a un elemento de formulario, cada elemento dispone de las siguientes propiedades útiles para el desarrollo de las aplicaciones:

- `type`: indica el tipo de elemento que se trata. Para los elementos de tipo `<input>` (`text`, `button`, `checkbox`, etc.) coincide con el valor de su atributo `type`. Para las listas desplegables normales (elemento `<select>`) su valor es `select-one`, lo que permite diferenciarlas de las listas que permiten seleccionar varios elementos a la vez y cuyo tipo es `select-multiple`. Por último, en los elementos de tipo `<textarea>`, el valor de type es `textarea`.
- `form`: es una referencia directa al formulario al que pertenece el elemento. Así, para acceder al formulario de un elemento, se puede utilizar `document.getElementById("id_del_elemento").form`

- `name`: obtiene el valor del atributo `name` de XHTML. Solamente se puede leer su valor, por lo que no se puede modificar.
- `value`: permite leer y modificar el valor del atributo `value` de XHTML. Para los campos de texto (`<input type="text">` y `<textarea>`) obtiene el texto que ha escrito el usuario. Para los botones obtiene el texto que se muestra en el botón. Para los elementos *checkbox* y *radiobutton* no es muy útil, como se verá más adelante

Por último, los eventos más utilizados en el manejo de los formularios son los siguientes:

- `onclick`: evento que se produce cuando se pincha con el ratón sobre un elemento. Normalmente se utiliza con cualquiera de los tipos de botones que permite definir XHTML (`<input type="button">`, `<input type="submit">`, `<input type="image">`).
- `onchange`: evento que se produce cuando el usuario cambia el valor de un elemento de texto (`<input type="text">` o `<textarea>`). También se produce cuando el usuario selecciona una opción en una lista desplegable (`<select>`). Sin embargo, el evento sólo se produce si después de realizar el cambio, el usuario pasa al siguiente campo del formulario, lo que técnicamente se conoce como que *"el otro campo de formulario ha perdido el foco"*.
- `onfocus`: evento que se produce cuando el usuario selecciona un elemento del formulario.
- `onblur`: evento complementario de `onfocus`, ya que se produce cuando el usuario ha deseleccionado un elemento por haber seleccionado otro elemento del formulario. Técnicamente, se dice que el elemento anterior "ha perdido el foco".

La mayoría de técnicas JavaScript relacionadas con los formularios requieren leer y/o modificar el valor de los campos del formulario. Por tanto, a continuación se muestra cómo obtener el valor de los campos de formulario más utilizados.

El valor del texto mostrado por estos elementos se obtiene y se establece directamente mediante la propiedad value.

```
<input type="text" id="texto" />
var valor = document.getElementById("texto").value;

<textarea id="parrafo"></textarea>
var valor = document.getElementById("parrafo").value;
```

Cuando se dispone de un grupo de *radiobuttons*, generalmente no se quiere obtener el valor del atributo `value` de alguno de ellos, sino que lo importante es conocer cuál de todos los *radiobuttons* se ha seleccionado. La propiedad `checked` devuelve `true` para el *radiobutton* seleccionado y `false` en cualquier otro caso. Si por ejemplo se dispone del siguiente grupo de *radiobuttons*:

```
<input type="radio" value="si" name="pregunta" id="pregunta_si"/> SI
<input type="radio" value="no" name="pregunta" id="pregunta_no"/> NO
<input type="radio" value="nsnc" name="pregunta" id="pregunta_nsnc"/> NS/
NC
```

El siguiente código permite determinar si cada radiobutton ha sido seleccionado o no:

```
var elementos = document.getElementsByName("pregunta");

for(var i=0; i<elementos.length; i++) {
  alert(" Elemento: " + elementos[i].value + "\n Seleccionado: " +
elementos[i].checked);
}
```

Los elementos de tipo *checkbox* son muy similares a los radiobutton, salvo que en este caso se debe comprobar cada *checkbox* de forma independiente del resto. El motivo es que los grupos de radiobutton son mutuamente excluyentes y sólo se puede seleccionar uno de ellos cada vez. Por su parte, los *checkbox* se pueden seleccionar de forma independiente respecto de los demás.

Si se dispone de los siguientes checkbox:

```
<input type="checkbox" value="condiciones" name="condiciones"
id="condiciones"/> He leído y acepto las condiciones
<input type="checkbox" value="privacidad" name="privacidad"
id="privacidad"/> He leído la política de privacidad
```

Utilizando la propiedad checked, es posible comprobar si cada checkbox ha sido seleccionado:

```
var elemento = document.getElementById("condiciones");
alert(" Elemento: " + elemento.value + "\n Seleccionado: " +
elemento.checked);

elemento = document.getElementById("privacidad");
alert(" Elemento: " + elemento.value + "\n Seleccionado: " +
elemento.checked);
```

Las listas desplegables (`<select>`) son los elementos en los que es más difícil obtener su valor. Si se dispone de una lista desplegable como la siguiente:

```
<select id="opciones" name="opciones">
  <option value="1">Primer valor</option>
  <option value="2">Segundo valor</option>
  <option value="3">Tercer valor</option>
  <option value="4">Cuarto valor</option>
</select>
```

En general, lo que se requiere es obtener el valor del atributo `value` de la opción (`<option>`) seleccionada por el usuario. Obtener este valor no es sencillo, ya que se deben realizar una serie de pasos. Además, para obtener el valor seleccionado, deben utilizarse las siguientes propiedades:

- `options`, es un array creado automáticamente por el navegador para cada lista desplegable y que contiene la referencia a todas las opciones de esa lista. De esta forma, la primera opción de una lista se puede obtener mediante `document.getElementById("id_de_la_lista").options[0]`.
- `selectedIndex`, cuando el usuario selecciona una opción, el navegador actualiza automáticamente el valor de esta propiedad, que guarda el índice de la opción seleccionada. El índice hace referencia al array `options` creado automáticamente por el navegador para cada lista.

```
// Obtener la referencia a la lista
var lista = document.getElementById("opciones");

// Obtener el índice de la opción que se ha seleccionado
```

```
var indiceSeleccionado = lista.selectedIndex;
// Con el índice y el array "options", obtener la opción seleccionada
var opcionSeleccionada = lista.options[indiceSeleccionado];

// Obtener el valor y el texto de la opción seleccionada
var textoSeleccionado = opcionSeleccionada.text;
var valorSeleccionado = opcionSeleccionada.value;

alert("Opción seleccionada: " + textoSeleccionado + "\n Valor de la
opción: " + valorSeleccionado);
```

Como se ha visto, para obtener el valor del atributo `value` correspondiente a la opción seleccionada por el usuario, es necesario realizar varios pasos. No obstante, normalmente se abrevian todos los pasos necesarios en una única instrucción:

```
var lista = document.getElementById("opciones");

// Obtener el valor de la opción seleccionada
var valorSeleccionado = lista.options[lista.selectedIndex].value;

// Obtener el texto que muestra la opción seleccionada
var valorSeleccionado = lista.options[lista.selectedIndex].text;
```

Lo más importante es no confundir el valor de la propiedad `selectedIndex` con el valor correspondiente a la propiedad `value` de la opción seleccionada. En el ejemplo anterior, la primera opción tiene un `value` igual a `1`. Sin embargo, si se selecciona esta opción, el valor de `selectedIndex` será `0`, ya que es la primera opción del array `options` (y los arrays empiezan a contar los elementos en el número `0`).

En programación, cuando un elemento está seleccionado y se puede escribir directamente en el o se puede modificar alguna de sus propiedades, se dice que tiene el foco del programa.

Si un cuadro de texto de un formulario tiene el foco, el usuario puede escribir directamente en el sin necesidad de pinchar previamente con el ratón en el interior del cuadro. Igualmente, si una lista desplegable tiene el foco, el usuario puede seleccionar una opción directamente subiendo y bajando con las flechas del teclado.

Al pulsar repetidamente la tecla TABULADOR sobre una página web, los diferentes elementos (enlaces, imágenes, campos de formulario, etc.) van obteniendo el foco del navegador (el elemento seleccionado cada vez suele mostrar un pequeño borde punteado).

Si en una página web el formulario es el elemento más importante, como por ejemplo en una página de búsqueda o en una página con un formulario para registrarse, se considera una buena práctica de usabilidad el asignar automáticamente el foco al primer elemento del formulario cuando se carga la página.

Para asignar el foco a un elemento de XHTML, se utiliza la función `focus()`. El siguiente ejemplo asigna el foco a un elemento de formulario cuyo atributo `id` es igual a `primero`:

```
document.getElementById("primero").focus();
```

```
<form id="formulario" action="#">
  <input type="text" id="primero" />
</form>
```

Ampliando el ejemplo anterior, se puede asignar automáticamente el foco del programa al primer elemento del primer formulario de la página, independientemente del id del formulario y de los elementos:

```
if(document.forms.length > 0) {
  if(document.forms[0].elements.length > 0) {
    document.forms[0].elements[0].focus();
  }
}
```

El código anterior comprueba que existe al menos un formulario en la página mediante el tamaño del array `forms`. Si su tamaño es mayor que `0`, se utiliza este primer formulario. Empleando la misma técnica, se comprueba que el formulario tenga al menos un elemento (`if(document.forms[0].elements.length > 0)`). En caso afirmativo, se establece el foco del navegador en el primer elemento del primer formulario (`document.forms[0].elements[0].focus();`).

Para que el ejemplo anterior sea completamente correcto, se debe añadir una comprobación adicional. El campo de formulario que se selecciona no debería ser de tipo `hidden`:

```
if(document.forms.length > 0) {
  for(var i=0; i < document.forms[0].elements.length; i++) {
    var campo = document.forms[0].elements[i];
    if(campo.type != "hidden") {
      campo.focus();
      break;
    }
  }
}
```

Uno de los problemas habituales con el uso de formularios web es la posibilidad de que el usuario pulse dos veces seguidas sobre el botón "Enviar". Si la conexión del usuario es demasiado lenta o la respuesta del servidor se hace esperar, el formulario original sigue mostrándose en el navegador y por ese motivo, el usuario tiene la tentación de volver a pinchar sobre el botón de "Enviar".

En la mayoría de los casos, el problema no es grave e incluso es posible controlarlo en el servidor, pero puede complicarse en formularios de aplicaciones importantes como las que implican transacciones económicas.

Por este motivo, una buena práctica en el diseño de aplicaciones web suele ser la de deshabilitar el botón de envío después de la primera pulsación. El siguiente ejemplo muestra el código necesario:

```
<form id="formulario" action="#">
  ...
  <input type="button" value="Enviar" onclick="this.disabled=true;
this.value='Enviando...'; this.form.submit()" />
</form>
```

Cuando se pulsa sobre el botón de envío del formulario, se produce el evento `onclick` sobre el botón y por tanto, se ejecutan las instrucciones JavaScript contenidas en el atributo `onclick`:

1. En primer lugar, se deshabilita el botón mediante la instrucción `this.disabled = true;`. Esta es la única instrucción necesaria si sólo se quiere deshabilitar un botón.
2. A continuación, se cambia el mensaje que muestra el botón. Del original "Enviar" se pasa al más adecuado "Enviando..."

3. Por último, se envía el formulario mediante la función `submit()` en la siguiente instrucción: `this.form.submit()`

El botón del ejemplo anterior está definido mediante un botón de tipo `<input type="button" />`, ya que el código JavaScript mostrado no funciona correctamente con un botón de tipo `<input type="submit" />`. Si se utiliza un botón de tipo`submit`, el botón se deshabilita antes de enviar el formulario y por tanto el formulario acaba sin enviarse.

La carencia más importante de los campos de formulario de tipo textarea es la imposibilidad de limitar el máximo número de caracteres que se pueden introducir, de forma similar al atributo `maxlength` de los cuadros de texto normales.

JavaScript permite añadir esta característica de forma muy sencilla. En primer lugar, hay que recordar que con algunos eventos (como `onkeypress`, `onclick` y `onsubmit`) se puede evitar su comportamiento normal si se devuelve el valor `false`.

Evitar el comportamiento normal equivale a modificar completamente el comportamiento habitual del evento. Si por ejemplo se devuelve el valor `false` en el evento `onkeypress`, la tecla pulsada por el usuario no se tiene en cuenta. Si se devuelve `false` en el evento `onclick` de un elemento como un enlace, el navegador no carga la página indicada por el enlace.

Si un evento devuelve el valor `true`, su comportamiento es el habitual:

```
<textarea onkeypress="return true;"></textarea>
```

En el textarea del ejemplo anterior, el usuario puede escribir cualquier carácter, ya que el evento `onkeypress` devuelve `true` y por tanto, su comportamiento es el normal y la tecla pulsada se transforma en un carácter dentro del textarea.

Sin embargo, en el siguiente ejemplo:

```
<textarea onkeypress="return false;"></textarea>
```

Como el valor devuelto por el evento `onkeypress` es igual a `false`, el navegador no ejecuta el comportamiento por defecto del evento, es decir, la tecla presionada no se transforma en ningún carácter dentro del texta-

rea. No importa las veces que se pulsen las teclas y no importa la tecla pulsada, ese textarea no permitirá escribir ningún carácter.

Aprovechando esta característica, es sencillo limitar el número de caracteres que se pueden escribir en un elemento de tipo textarea: se comprueba si se ha llegado al máximo número de caracteres permitido y en caso afirmativo se evita el comportamiento habitual del evento y por tanto, los caracteres adicionales no se añaden al textarea:

```
function limita(maximoCaracteres) {
  var elemento = document.getElementById("texto");
  if(elemento.value.length >= maximoCaracteres ) {
    return false;
  }
  else {
    return true;
  }
}
```

```
<textarea id="texto" onkeypress="return limita(100);"></textarea>
```

En el ejemplo anterior, con cada tecla pulsada se compara el número total de caracteres del textarea con el máximo número de caracteres permitido. Si el número de caracteres es igual o mayor que el límite, se devuelve el valor `false` y por tanto, se evita el comportamiento por defecto de `onkeypress` y la tecla no se añade.

En ocasiones, puede ser útil bloquear algunos caracteres determinados en un cuadro de texto. Si por ejemplo un cuadro de texto espera que se introduzca un número, puede ser interesante no permitir al usuario introducir ningún carácter que no sea numérico.

Igualmente, en algunos casos puede ser útil impedir que el usuario introduzca números en un cuadro de texto. Utilizando el evento `onkeypress` y unas cuantas sentencias JavaScript, el problema se resuelve fácilmente:

```
function permite(elEvento, permitidos) {
  // Variables que definen los caracteres permitidos
  var numeros = "0123456789";
  var caracteres = "
abcdefghijklmnñopqrstuvwxyzABCDEFGHIJKLMNÑOPQRSTUVWXYZ";
```

```
  var numeros_caracteres = numeros + caracteres;
  var teclas_especiales = [8, 37, 39, 46];
  // 8 = BackSpace, 46 = Supr, 37 = flecha izquierda, 39 = flecha derecha


  // Seleccionar los caracteres a partir del parámetro de la función
  switch(permitidos) {
    case 'num':
      permitidos = numeros;
      break;
    case 'car':
      permitidos = caracteres;
      break;
    case 'num_car':
      permitidos = numeros_caracteres;
      break;
  }

  // Obtener la tecla pulsada
  var evento = elEvento || window.event;
  var codigoCaracter = evento.charCode || evento.keyCode;
  var caracter = String.fromCharCode(codigoCaracter);

  // Comprobar si la tecla pulsada es alguna de las teclas especiales
  // (teclas de borrado y flechas horizontales)
  var tecla_especial = false;
  for(var i in teclas_especiales) {
    if(codigoCaracter == teclas_especiales[i]) {
      tecla_especial = true;
      break;
    }
  }

  // Comprobar si la tecla pulsada se encuentra en los caracteres
permitidos
  // o si es una tecla especial
  return permitidos.indexOf(caracter) != -1 || tecla_especial;
}
```

```
// Sólo números
<input type="text" id="texto" onkeypress="return permite(event, 'num')"
/>

// Sólo letras
<input type="text" id="texto" onkeypress="return permite(event, 'car')"
```

```
/>

// Sólo letras o números
<input type="text" id="texto" onkeypress="return permite(event,
'num_car')" />
```

El funcionamiento del script anterior se basa en permitir o impedir el comportamiento habitual del evento `onkeypress`. Cuando se pulsa una tecla, se comprueba si el carácter de esa tecla se encuentra dentro de los caracteres permitidos para ese elemento `<input>`.

Si el carácter se encuentra dentro de los caracteres permitidos, se devuelve `true` y por tanto el comportamiento de `onkeypress` es el habitual y la tecla se escribe. Si el carácter no se encuentra dentro de los caracteres permitidos, se devuelve `false` y por tanto se impide el comportamiento normal de `onkeypress` y la tecla no llega a escribirse en el input.

Además, el script anterior siempre permite la pulsación de algunas teclas *especiales*. En concreto, las teclas `BackSpace` y `Supr` para borrar caracteres y las teclas `Flecha Izquierda` y `Flecha Derecha` para moverse en el cuadro de texto siempre se pueden pulsar independientemente del tipo de caracteres permitidos.

La principal utilidad de JavaScript en el manejo de los formularios es la validación de los datos introducidos por los usuarios. Antes de enviar un formulario al servidor, se recomienda validar mediante JavaScript los datos insertados por el usuario. De esta forma, si el usuario ha cometido algún error al rellenar el formulario, se le puede notificar de forma instantánea, sin necesidad de esperar la respuesta del servidor.

Notificar los errores de forma inmediata mediante JavaScript mejora la satisfacción del usuario con la aplicación (lo que técnicamente se conoce como "mejorar la experiencia de usuario") y ayuda a reducir la carga de procesamiento en el servidor.

Normalmente, la validación de un formulario consiste en llamar a una función de validación cuando el usuario pulsa sobre el botón de envío del formulario. En esta función, se comprueban si los valores que ha introducido el usuario cumplen las restricciones impuestas por la aplicación.

Aunque existen tantas posibles comprobaciones como elementos de formulario diferentes, algunas comprobaciones son muy habituales: que se rellene un campo obligatorio, que se seleccione el valor de una lista desplegable, que la dirección de email indicada sea correcta, que la fecha introducida sea lógica, que se haya introducido un número donde así se requiere, etc.

A continuación se muestra el código JavaScript básico necesario para incorporar la validación a un formulario:

```
<form action="" method="" id="" name="" onsubmit="return validacion()">
  ...
</form>
```

Y el esquema de la función validacion() es el siguiente:

```
function validacion() {
  if (condicion que debe cumplir el primer campo del formulario) {
    // Si no se cumple la condicion...
    alert('[ERROR] El campo debe tener un valor de...');
    return false;
  }
  else if (condicion que debe cumplir el segundo campo del formulario) {
    // Si no se cumple la condicion...
    alert('[ERROR] El campo debe tener un valor de...');
    return false;
  }
  ...
  else if (condicion que debe cumplir el último campo del formulario) {
    // Si no se cumple la condicion...
    alert('[ERROR] El campo debe tener un valor de...');
    return false;
  }

  // Si el script ha llegado a este punto, todas las condiciones
  // se han cumplido, por lo que se devuelve el valor true
  return true;
}
```

El funcionamiento de esta técnica de validación se basa en el comportamiento del evento `onsubmit` de JavaScript. Al igual que otros eventos como `onclick` y `onkeypress`, el evento `'onsubmit` varía su comportamiento en función del valor que se devuelve.

Así, si el evento `onsubmit` devuelve el valor `true`, el formulario se envía como lo haría normalmente. Sin embargo, si el evento `onsubmit` devuelve el valor `false`, el formulario no se envía. La clave de esta técnica consiste en comprobar todos y cada uno de los elementos del formulario. En cuando se encuentra un elemento incorrecto, se devuelve el valor `false`. Si no se encuentra ningún error, se devuelve el valor `true`.

Por lo tanto, en primer lugar se define el evento `onsubmit` del formulario como:

```
onsubmit="return validacion()"
```

Como el código JavaScript devuelve el valor resultante de la función `validacion()`, el formulario solamente se enviará al servidor si esa función devuelve `true`. En el caso de que la función `validacion()` devuelva `false`, el formulario permanecerá sin enviarse.

Dentro de la función `validacion()` se comprueban todas las condiciones impuestas por la aplicación. Cuando no se cumple una condición, se devuelve `false` y por tanto el formulario no se envía. Si se llega al final de la función, todas las condiciones se han cumplido correctamente, por lo que se devuelve `true` y el formulario se envía.

La notificación de los errores cometidos depende del diseño de cada aplicación. En el código del ejemplo anterior simplemente se muestran mensajes mediante la función `alert()` indicando el error producido. Las aplicaciones web mejor diseñadas muestran cada mensaje de error al lado del elemento de formulario correspondiente y también suelen mostrar un mensaje principal indicando que el formulario contiene errores.

Una vez definido el esquema de la función `validacion()`, se debe añadir a esta función el código correspondiente a todas las comprobaciones que se realizan sobre los elementos del formulario. A continuación, se muestran algunas de las validaciones más habituales de los campos de formulario.

Se trata de forzar al usuario a introducir un valor en un cuadro de texto o textarea en los que sea obligatorio. La condición en JavaScript se puede indicar como:

```
valor = document.getElementById("campo").value;
if( valor == null || valor.length == 0 || /^\s+$/.test(valor) ) {
  return false;
}
```

Para que se de por completado un campo de texto obligatorio, se comprueba que el valor introducido sea válido, que el número de caracteres introducido sea mayor que cero y que no se hayan introducido sólo espacios en blanco.

La palabra reservada `null` es un valor especial que se utiliza para indicar *"ningún valor"*. Si el valor de una variable es `null`, la variable no contiene ningún valor de tipo objeto, array, numérico, cadena de texto o booleano.

La segunda parte de la condición obliga a que el texto introducido tenga una longitud superior a cero caracteres, esto es, que no sea un texto vacío.

Por último, la tercera parte de la condición `(/^\s+$/.test(valor))` obliga a que el valor introducido por el usuario no sólo esté formado por espacios en blanco. Esta comprobación se basa en el uso de "expresiones regulares", un recurso habitual en cualquier lenguaje de programación pero que por su gran complejidad no se van a estudiar. Por lo tanto, sólo es necesario copiar literalmente esta condición, poniendo especial cuidado en no modificar ningún carácter de la expresión.

Se trata de obligar al usuario a introducir un valor numérico en un cuadro de texto. La condición JavaScript consiste en:

```
valor = document.getElementById("campo").value;
if( isNaN(valor) ) {
  return false;
}
```

Si el contenido de la variable `valor` no es un número válido, no se cumple la condición. La ventaja de utilizar la función interna `isNaN()` es que simplifica las comprobaciones, ya que JavaScript se encarga de tener en cuenta los decimales, signos, etc.

A continuación se muestran algunos resultados de la función `isNaN()`:

```
isNaN(3);          // false
isNaN("3");        // false
isNaN(3.3545);     // false
isNaN(32323.345);  // false
isNaN(+23.2);      // false
isNaN("-23.2");    // false
isNaN("23a");      // true
isNaN("23.43.54"); // true
```

Se trata de obligar al usuario a seleccionar un elemento de una lista desplegable. El siguiente código JavaScript permite conseguirlo:

```
indice = document.getElementById("opciones").selectedIndex;
if( indice == null || indice == 0 ) {
  return false;
}
```

```
<select id="opciones" name="opciones">
  <option value="">- Selecciona un valor -</option>
  <option value="1">Primer valor</option>
  <option value="2">Segundo valor</option>
  <option value="3">Tercer valor</option>
</select>
```

A partir de la propiedad `selectedIndex`, se comprueba si el índice de la opción seleccionada es válido y además es distinto de cero. La primera opción de la lista (`- Selecciona un valor –`) no es válida, por lo que no se permite el valor `0` para esta propiedad `selectedIndex`.

Se trata de obligar al usuario a introducir una dirección de email con un formato válido. Por tanto, lo que se comprueba es que la dirección parezca válida, ya que no se comprueba si se trata de una cuenta de correo electrónico real y operativa. La condición JavaScript consiste en:

```
valor = document.getElementById("campo").value;
if( !(/\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)/.test(valor)) ) {
  return false;
}
```

La comprobación se realiza nuevamente mediante las expresiones regulares, ya que las direcciones de correo electrónico válidas pueden ser

muy diferentes. Por otra parte, como el estándar que define el formato de las direcciones de correo electrónico es muy complejo, la expresión regular anterior es una simplificación. Aunque esta regla valida la mayoría de direcciones de correo electrónico utilizadas por los usuarios, no soporta todos los diferentes formatos válidos de email.

Las fechas suelen ser los campos de formulario más complicados de validar por la multitud de formas diferentes en las que se pueden introducir. El siguiente código asume que de alguna forma se ha obtenido el año, el mes y el día introducidos por el usuario:

```
var ano = document.getElementById("ano").value;
var mes = document.getElementById("mes").value;
var dia = document.getElementById("dia").value;

valor = new Date(ano, mes, dia);

if( !isNaN(valor) ) {
  return false;
}
```

La función `Date(ano, mes, dia)` es una función interna de JavaScript que permite construir fechas a partir del año, el mes y el día de la fecha. Es muy importante tener en cuenta que el número de mes se indica de `0` a `11`, siendo `0` el mes de Enero y `11` el mes de Diciembre. Los días del mes siguen una numeración diferente, ya que el mínimo permitido es `1` y el máximo `31`.

La validación consiste en intentar construir una fecha con los datos proporcionados por el usuario. Si los datos del usuario no son correctos, la fecha no se puede construir correctamente y por tanto la validación del formulario no será correcta.

Se trata de comprobar que el número proporcionado por el usuario se corresponde con un número válido de Documento Nacional de Identidad o DNI. Aunque para cada país o región los requisitos del documento de identidad de las personas pueden variar, a continuación se muestra un ejemplo genérico fácilmente adaptable. La validación no sólo debe comprobar que el número esté formado por ocho cifras y una letra, sino que

también es necesario comprobar que la letra indicada es correcta para el número introducido:

```
valor = document.getElementById("campo").value;
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X',
'B', 'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];

if( !(/^\d{8}[A-Z]$/.test(valor)) ) {
  return false;
}

if(valor.charAt(8) != letras[(valor.substring(0, 8))%23]) {
  return false;
}
```

La primera comprobación asegura que el formato del número introducido es el correcto, es decir, que está formado por 8 números seguidos y una letra. Si la letra está al principio de los números, la comprobación sería `/^[A-Z]\d{8}$/`. Si en vez de ocho números y una letra, se requieren diez números y dos letras, la comprobación sería `/^\d{10}[A-Z]{2}$/` y así sucesivamente.

La segunda comprobación aplica el algoritmo de cálculo de la letra del DNI y la compara con la letra proporcionada por el usuario. El algoritmo de cada documento de identificación es diferente, por lo que esta parte de la validación se debe adaptar convenientemente.

Los números de teléfono pueden ser indicados de formas muy diferentes: con prefijo nacional, con prefijo internacional, agrupado por pares, separando los números con guiones, etc.

El siguiente script considera que un número de teléfono está formado por nueve dígitos consecutivos y sin espacios ni guiones entre las cifras:

```
valor = document.getElementById("campo").value;
if( !(/^\d{9}$/.test(valor)) ) {
  return false;
}
```

Una vez más, la condición de JavaScript se basa en el uso de expresiones regulares, que comprueban si el valor indicado es una sucesión de nueve números consecutivos. A continuación se muestran otras expre-

siones regulares que se pueden utilizar para otros formatos de número de teléfono:

`900900900`	`/^\d{9}$/`	9 cifras seguidas
`900-900-900`	`/^\d{3}-\d{3}-\d{3}$/`	9 cifras agrupadas de 3 en 3 y separadas por guiones
`900 900900`	`/^\d{3}\s\d{6}$/`	9 cifras, las 3 primeras separadas por un espacio
`900 90 0900`	`/^\d{3}\s\d{2}\s\d{2}\s\d{2}$/`	9 cifras, las 3 primeras separadas por un espacio, las siguientes agrupadas de 2 en 2
`(900) 900900`	`/^\(\d{3}\)\s\d{6}$/`	9 cifras, las 3 primeras encerradas por paréntesis y un espacio de separación respecto del resto
`+34 900900900`	`/^\+\d{2,3}\s\d{9}$/`	Prefijo internacional (+ seguido de 2 o 3 cifras), espacio en blanco y 9 cifras consecutivas

Si un elemento de tipo *checkbox* se debe seleccionar de forma obligatoria, JavaScript permite comprobarlo de forma muy sencilla:

```
elemento = document.getElementById("campo");
if( !elemento.checked ) {
  return false;
}
```

Si se trata de comprobar que todos los checkbox del formulario han sido seleccionados, es más fácil utilizar un bucle:

```
formulario = document.getElementById("formulario");
for(var i=0; i<formulario.elements.length; i++) {
  var elemento = formulario.elements[i];
  if(elemento.type == "checkbox") {
    if(!elemento.checked) {
      return false;
    }
```

```
    }
  }
```

Aunque se trata de un caso similar al de los *checkbox*, la validación de los *radiobutton* presenta una diferencia importante: en general, la comprobación que se realiza es que el usuario haya seleccionado algún *radiobutton* de los que forman un determinado grupo. Mediante JavaScript, es sencillo determinar si se ha seleccionado algún *radiobutton* de un grupo:

```
opciones = document.getElementsByName("opciones");

var seleccionado = false;
for(var i=0; i<opciones.length; i++) {
  if(opciones[i].checked) {
    seleccionado = true;
    break;
  }
}

if(!seleccionado) {
  return false;
}
```

El anterior ejemplo recorre todos los *radiobutton* que forman un grupo y comprueba elemento por elemento si ha sido seleccionado. Cuando se encuentra el primer *radiobutton* seleccionado, se sale del bucle y se indica que al menos uno ha sido seleccionado.

Esta página se ha dejado vacía a propósito

Capítulo 17

Cuando se desarrollan aplicaciones complejas, es habitual encontrarse con decenas de archivos JavaScript de miles de líneas de código. Estructurar las aplicaciones de esta forma es correcto y facilita el desarrollo de la aplicación, pero penaliza en exceso el rendimiento de la aplicación.

La primera recomendación para mejorar el rendimiento de la aplicación consiste en unir en un único archivo JavaScript el contenido de todos los diferentes archivos JavaScript. En Windows, se puede crear un pequeño programa ejecutable que copia el contenido de varios archivos JavaScript en uno solo:

```
more archivo1.js > archivoUnico.js
more archivo2.js >> archivoUnico.js
more archivo3.js >> archivoUnico.js
...
```

La primera instrucción tiene un solo símbolo `>` para borrar el contenido del `archivoUnico.js` cada vez que se ejecuta el comando. El resto de instrucciones tienen un símbolo `>>` para añadir el contenido de los demás archivos al final del `archivoUnico.js`

En sistemas operativos de tipo Linux es todavía más sencillo unir varios archivos en uno solo:

```
cat archivo1.js archivo2.js archivo3.js > archivoUnico.js
```

La única consideración que se debe tener en cuenta con este método es el de las dependencias entre archivos. Si por ejemplo el `archivo1.js` contiene funciones que dependen de otras funciones definidas en el `archivo3.js`, los archivos deberían unirse en este otro orden:

```
cat archivo3.js archivo1.js archivo2.js > archivoUnico.js
```

Otra recomendación muy útil para mejorar el rendimiento de la aplicación es la de comprimir el código de JavaScript. Este tipo de herramientas compresoras de código no modifican el comportamiento de la aplicación, pero pueden reducir mucho su tamaño.

El proceso de compresión consiste en eliminar todos los espacios en blanco sobrantes, eliminar todos los comentarios del código y convertir toda la aplicación en una única línea de código JavaScript muy larga. Algunos compresores van más allá y sustituyen el nombre de las variables y funciones por nombres más cortos.

ShrinkSafe (http://dojotoolkit.org/shrinksafe/) es una de las herramientas que proporciona el framework Dojo (http://dojotoolkit.org/) y que puede ser utilizada incluso de forma online. Los creadores de la aplicación aseguran de que es la herramienta más segura para reducir el tamaño del código, ya que no modifica ningún elemento que pueda provocar errores en la aplicación.

El código de las aplicaciones JavaScript, al igual que el resto de contenidos de las páginas web, está disponible para ser accedido y visualizado por cualquier usuario. Con la aparición de las aplicaciones basadas en AJAX, muchas empresas han desarrollado complejas aplicaciones cuyo código fuente está a disposición de cualquier usuario.

Aunque se trata de un problema casi imposible de solucionar, existen técnicas que minimizan el problema de que se pueda acceder libremente al código fuente de la aplicación. La principal técnica es la de ofuscar el código fuente de la aplicación.

Los *ofuscadores* utilizan diversos mecanismos para hacer casi imposible de entender el código fuente de una aplicación. Manteniendo el compor-

tamiento de la aplicación, consiguen ensuciar y dificultar tanto el código que no es mayor problema que alguien pueda acceder a ese código.

El programa ofuscador Jasob (http://www.jasob.com/) ofrece un ejemplo del resultado de ofuscar cierto código JavaScript. Este es el código original antes de ofuscarlo:

```
//---------------------------------------------------
// Calculate salary for each employee in "aEmployees".
// "aEmployees" is array of "Employee" objects.
//---------------------------------------------------
function CalculateSalary(aEmployees)
{
   var nEmpIndex = 0;
   while (nEmpIndex < aEmployees.length)
   {
      var oEmployee = aEmployees[nEmpIndex];
      oEmployee.fSalary = CalculateBaseSalary(oEmployee.nType,
                                              oEmployee.nWorkingHours);
      if (oEmployee.bBonusAllowed == true)
      {
         oEmployee.fBonus = CalculateBonusSalary(oEmployee.nType,
                                                 oEmployee.nWorkingHours,
                                                 oEmployee.fSalary);
      }
      else
      {
         oEmployee.fBonus = 0;
      }
      oEmployee.sSalaryColor = GetSalaryColor(oEmployee.fSalary +
                                              oEmployee.fBonus);
      nEmpIndex++;
   }
}
```

Después de pasar el código anterior por el ofuscador el resultado es:

```
function c(g){var m=0;while(m<g.length){var
r=g[m];r.l=d(r.n,r.o);if(r.j==true){r.k=e(r.n,r.o,r.l);}else{r.k=0;}r.t=f(r.l+r.k);
```

Al sustituir todos los nombres de las variables y de las funciones por nombres de una sola letra, es prácticamente imposible comprender el código del programa. En ocasiones, también se utilizan ofuscadores de este tipo con el propósito de reducir el tamaño del código fuente.

Además de aplicaciones comerciales específicamente diseñadas para ofuscar código JavaScript, también se pueden utilizar las herramientas que minimizan el tamaño de los scripts. Eliminando los comentarios y reduciendo el nombre de todas las variables, los programas que minimizan el tamaño de los scripts también consiguen ofuscar su código.

La aplicación packer (http://dean.edwards.name/packer/) es gratuita, se puede acceder via web y consigue una excelente compresión del código original. También se puede utilizar jsjuicer (http://adrian3.googlepages.com/jsjuicer.html) , que está disponible como aplicación descargable y también se puede utilizar vía web (http://gueschla.com/labs/jsjuicer/) .

Capítulo 18

Modificar el siguiente script para que:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"
/>
<title>El primer script</title>

<script type="text/javascript">
  alert("Hola Mundo!");
</script>
</head>

<body>
<p>Esta página contiene el primer script</p>
</body>
</html>
```

1. Todo el código JavaScript se encuentre en un archivo externo llamado `codigo.js` y el script siga funcionando de la misma manera.
2. Después del primer mensaje, se debe mostrar otro mensaje que diga "Soy el primer script"

3. Añadir algunos comentarios que expliquen el funcionamiento del código
4. Añadir en la página XHTML un mensaje de aviso para los navegadores que no tengan activado el soporte de JavaScript

Modificar el script del ejercicio anterior para que:

1. El mensaje que se muestra al usuario se almacene en una variable llamada mensaje y el funcionamiento del script sea el mismo.
2. El mensaje mostrado sea el de la siguiente imagen:

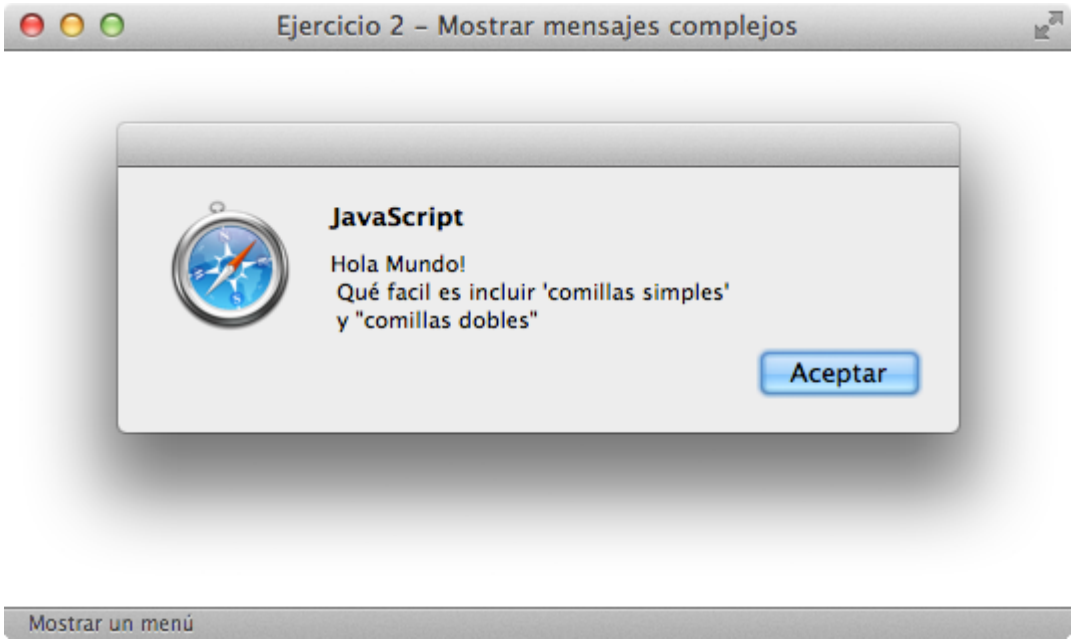


Figura 18.1 Nuevo mensaje que debe mostrar el script

Crear un array llamado meses y que almacene el nombre de los doce meses del año. Mostrar por pantalla los doce nombres utilizando la función `console.log()`.

A partir del siguiente array que se proporciona: `var valores = [true, 5, false, "hola", "adios", 2];`

1. Determinar cual de los dos elementos de texto es mayor

2. Utilizando exclusivamente los dos valores booleanos del array, determinar los operadores necesarios para obtener un resultado true y otro resultado false
3. Determinar el resultado de las cinco operaciones matemáticas realizadas con los dos elementos numéricos

Completar las condiciones de los if del siguiente script para que los mensajes se muestren siempre de forma correcta:

```
var numero1 = 5;
var numero2 = 8;

if(...) {
  console.log("numero1 no es mayor que numero2");
}
if(...) {
  console.log("numero2 es positivo");
}
if(...) {
  console.log("numero1 es negativo o distinto de cero");
}
if(...) {
  console.log("Incrementar en 1 unidad el valor de numero1 no lo hace
mayor o igual que numero2");
}
```

El cálculo de la letra del Documento Nacional de Identidad (DNI) es un proceso matemático sencillo que se basa en obtener el resto de la división entera del número de DNI y el número 23. A partir del resto de la división, se obtiene la letra seleccionándola dentro de un array de letras.

El array de letras es:

```
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X',
'B', 'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];
```

Por tanto si el resto de la división es `0`, la letra del DNI es la `T` y si el resto es `3` la letra es la `A`. Con estos datos, elaborar un pequeño script que:

1. Almacene en una variable el número de DNI indicado por el usuario y en otra variable la letra del DNI que se ha indicado.

2. En primer lugar (y en una sola instrucción) se debe comprobar si el número es menor que `0` o mayor que `99999999`. Si ese es el caso, se muestra un mensaje al usuario indicando que el número proporcionado no es válido y el programa no muestra más mensajes.
3. Si el número es válido, se calcula la letra que le corresponde según el método explicado anteriormente.
4. Una vez calculada la letra, se debe comparar con la letra indicada por el usuario. Si no coinciden, se muestra un mensaje al usuario diciéndole que la letra que ha indicado no es correcta. En otro caso, se muestra un mensaje indicando que el número y la letra de DNI son correctos.

El factorial de un número entero n es una operación matemática que consiste en multiplicar todos los factores `n x (n-1) x (n-2) x ... x 1`. Así, el factorial de `5` (escrito como `5!`) es igual a: `5! = 5 x 4 x 3 x 2 x 1 = 120`

Utilizando la estructura `for`, crear un script que calcule el factorial de un número entero.

Escribir el código de una función a la que se pasa como parámetro un número entero y devuelve como resultado una cadena de texto que indica si el número es par o impar. Mostrar por pantalla el resultado devuelto por la función.

Definir una función que muestre información sobre una cadena de texto que se le pasa como argumento. A partir de la cadena que se le pasa, la función determina si esa cadena está formada sólo por mayúsculas, sólo por minúsculas o por una mezcla de ambas.

Definir una función que determine si la cadena de texto que se le pasa como parámetro es un palíndromo, es decir, si se lee de la misma forma desde la izquierda y desde la derecha. Ejemplo de palíndromo complejo: "La ruta nos aporto otro paso natural".

Crear las expresiones regulares necesarias para resolver los siguientes puntos:

- Crear una expresión regular valide una fecha en formato "XX/XX/XXXX", donde "X" es un dígito. Probarlo con la expresión: "Nací el 05/04/1982 en Donostia.".
- Crear una expresión regular que valide una dirección de email. Para simplificar, los valores antes de la @ pueden contener cualquier carácter alfanumérico, y los caracteres `.` y `-`, mientras que los valores tra la @ pueden contener caracteres alfanuméricos, y el tipo de dominio puede tener una longitud de 2 o 3 caracteres.
- Dada la siguiente función que de reemplazo de caracteres, reescribirla utilizando expresiones regulares.

```
function escapeHTML(text) {
    var replacements = [["&", "&amp;"], ["\"", "&quot;"],
                        ["<", "&lt;"], [">", "&gt;"]];
    forEach(replacements, function(replace) {
        text = text.replace(replace[0], replace[1]);
    });
    return text;
}
```

- Dados un nombre y un apellido, crear la expresión regulatr necesaria para mostrarlos en orden inverso y separados por una coma. Por ejemplo, la cadena "John Smith", convertirla en "Smith, John".
- Crear una expresión regular que elimine las etiquetas potencialmente peligrosas (`<script>...</script>`) y todo su contenido de una cadena HTML.

A partir de la página web proporcionada y utilizando las funciones DOM, mostrar por pantalla la siguiente información:

1. Número de enlaces de la página
2. Dirección a la que enlaza el penúltimo enlace
3. Numero de enlaces que enlazan a http://prueba

4. Número de enlaces del tercer párrafo

Descargar página HTML (snippets/cap18/ej12.html)

Completar el código JavaScript proporcionado para que cuando se pulse sobre el enlace se muestre completo el contenido de texto. Además, el enlace debe dejar de mostrarse después de pulsarlo por primera vez. La acción de pulsar sobre un enlace forma parte de los "Eventos" de JavaScript que se ven en el siguiente capítulo. En este ejercicio, sólo se debe saber que al pinchar sobre el enlace, se ejecuta la función llamada muestra().

Descargar página HTML (snippets/cap18/ej13.html)

Completar el código JavaScript proporcionado para que se añadan nuevos elementos a la lista cada vez que se pulsa sobre el botón. Utilizar las funciones DOM para crear nuevos nodos y añadirlos a la lista existente. Al igual que sucede en el ejercicio anterior, la acción de pinchar sobre un botón forma parte de los "Eventos" de JavaScript que se ven en el siguiente capítulo. En este ejercicio, sólo se debe saber que al pinchar sobre el botón, se ejecuta la función llamada anade().

Descargar página HTML (snippets/cap18/ej14.html)

A partir de la página web proporcionada, completar el código JavaScript para que:

1. Cuando se pinche sobre el primer enlace, se oculte su sección relacionada
2. Cuando se vuelva a pinchar sobre el mismo enlace, se muestre otra vez esa sección de contenidos
3. Completar el resto de enlaces de la página para que su comportamiento sea idéntico al del primer enlace
4. Cuando una sección se oculte, debe cambiar el mensaje del enlace asociado

Descargar página HTML (snippets/cap18/ej15.html)

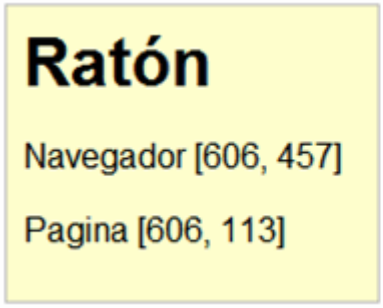
Completar el código JavaScript proporcionado para que:

- Al mover el ratón en cualquier punto de la ventana del navegador, se muestre la posición del puntero respecto del navegador y respecto de la página:

Para mostrar los mensajes, utilizar la función `muestraInformacion()` deduciendo su funcionamiento a partir de su código fuente.

- Al pulsar cualquier tecla, el mensaje mostrado debe cambiar para indicar el nuevo evento y su información asociada:

- Añadir la siguiente característica al script: cuando se pulsa un botón del ratón, el color de fondo del cuadro de mensaje debe ser amarillo (`#FFFFCC`) y cuando se pulsa una tecla, el color de fondo debe ser azul (`#CCE6FF`). Al volver a mover el ratón, el color de fondo vuelve a ser blanco.



Descargar página HTML (snippets/cap18/ej16.html)

Crear un script que informe al usuario en que zona de la pantalla ha pulsado el ratón. Las zonas definidas son las siguientes: izquierda arriba, izquierda abajo, derecha arriba y derecha abajo. Para determinar el tamaño de la ventana del navegador, utilizar la función `tamanoVentanaNavegador()` proporcionada.

Descargar página HTML (snippets/cap18/ej17.html)

Mejorar el ejemplo anterior indicando en todo momento al usuario el número de caracteres que aún puede escribir. Además, se debe permitir pulsar las teclas Backspace, Supr. y las flechas horizontales cuando se haya llegado al máximo número de caracteres.