Introducción a
AJAX

Javier Eguíluz Pérez

Sobre este libro...

- Los contenidos de este libro están bajo una licencia Creative Commons Reconocimiento - No Comercial - Sin Obra Derivada 3.0 (http://creativecommons.org/licenses/by-nc-nd/3.0/deed.es)
- **Esta versión impresa se creó el 7 de junio de 2008 y todavía está incompleta**. La versión más actualizada de los contenidos de este libro se puede encontrar en http://www.librosweb.es/ajax
- Si quieres aportar sugerencias, comentarios, críticas o informar sobre errores, puedes enviarnos un mensaje a **contacto@librosweb.es**

Capítulo 1. Introducción a AJAX

El término AJAX se presentó por primera vez en el artículo "Ajax: A New Approach to Web Applications (http://www.adaptivepath.com/publications/essays/archives/000385.php) " publicado por Jesse James Garrett el 18 de Febrero de 2005. Hasta ese momento, no existía un término normalizado que hiciera referencia a un nuevo tipo de aplicación web que estaba apareciendo.

En realidad, el término AJAX es un acrónimo de *Asynchronous JavaScript + XML*, que se puede traducir como "JavaScript asíncrono + XML".

El artículo define AJAX de la siguiente forma:

> " Ajax no es una tecnología en sí mismo. En realidad, se trata de varias tecnologías independientes que se unen de formas nuevas y sorprendentes."

Las tecnologías que forman AJAX son:

- XHTML y CSS, para crear una presentación basada en estándares.
- DOM, para la interacción y manipulación dinámica de la presentación.
- XML, XSLT y JSON, para el intercambio y la manipulación de información.
- XMLHttpRequest, para el intercambio asíncrono de información.
- JavaScript, para unir todas las demás tecnologías.

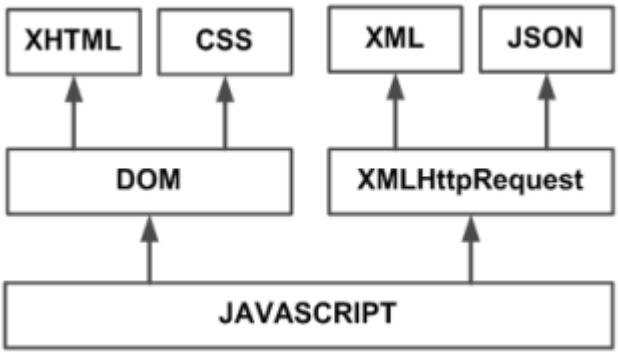

Figura 1.1. Tecnologías agrupadas bajo el concepto de AJAX

Desarrollar aplicaciones AJAX requiere un conocimiento avanzado de todas y cada una de las tecnologías anteriores.

En las aplicaciones web tradicionales, las acciones del usuario en la página (pinchar en un botón, seleccionar un valor de una lista, etc.) desencadenan llamadas al servidor. Una vez procesada la petición del usuario, el servidor devuelve una nueva página HTML al navegador del usuario.

En el siguiente esquema, la imagen de la izquierda muestra el modelo tradicional de las aplicaciones web. La imagen de la derecha muestra el nuevo modelo propuesto por AJAX:

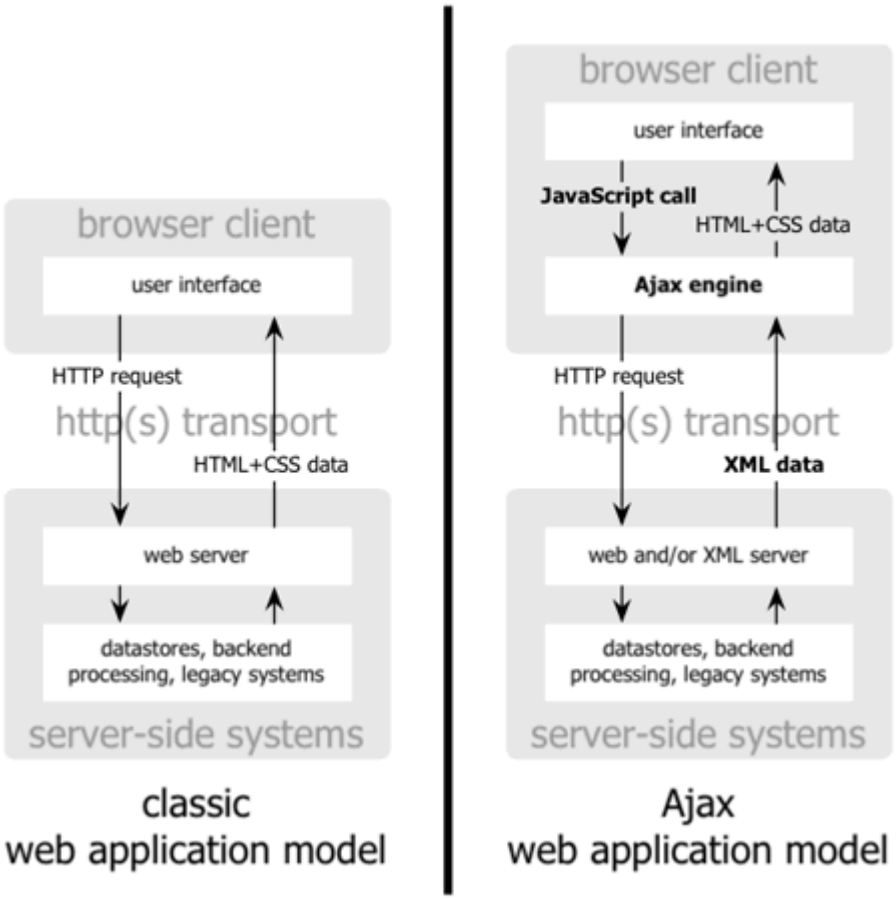

Figura 1.2. Comparación gráfica del modelo tradicional de aplicación web y del nuevo modelo propuesto por AJAX. (Imagen original creada por Adaptive Path y utilizada con su permiso)

Esta técnica tradicional para crear aplicaciones web funciona correctamente, pero no crea una buena sensación al usuario. Al realizar peticiones continuas al servidor, el usuario debe esperar a que se recargue la página con los cambios solicitados. Si la aplicación debe realizar peticiones continuas, su uso se convierte en algo molesto

AJAX permite mejorar completamente la interacción del usuario con la aplicación, evitando las recargas constantes de la página, ya que el intercambio de información con el servidor se produce en un segundo plano.

Las aplicaciones construidas con AJAX eliminan la recarga constante de páginas mediante la creación de un elemento intermedio entre el usuario y el servidor. La nueva capa intermedia de AJAX mejora la respuesta de la aplicación, ya que el usuario nunca se encuentra con una ventana del navegador vacía esperando la respuesta del servidor.

El siguiente esquema muestra la diferencia más importante entre una aplicación web tradicional y una aplicación web creada con AJAX. La imagen superior muestra la interación síncrona propia de las aplicaciones web tradicionales. La imagen inferior muestra la comunicación asíncrona de las aplicaciones creadas con AJAX.

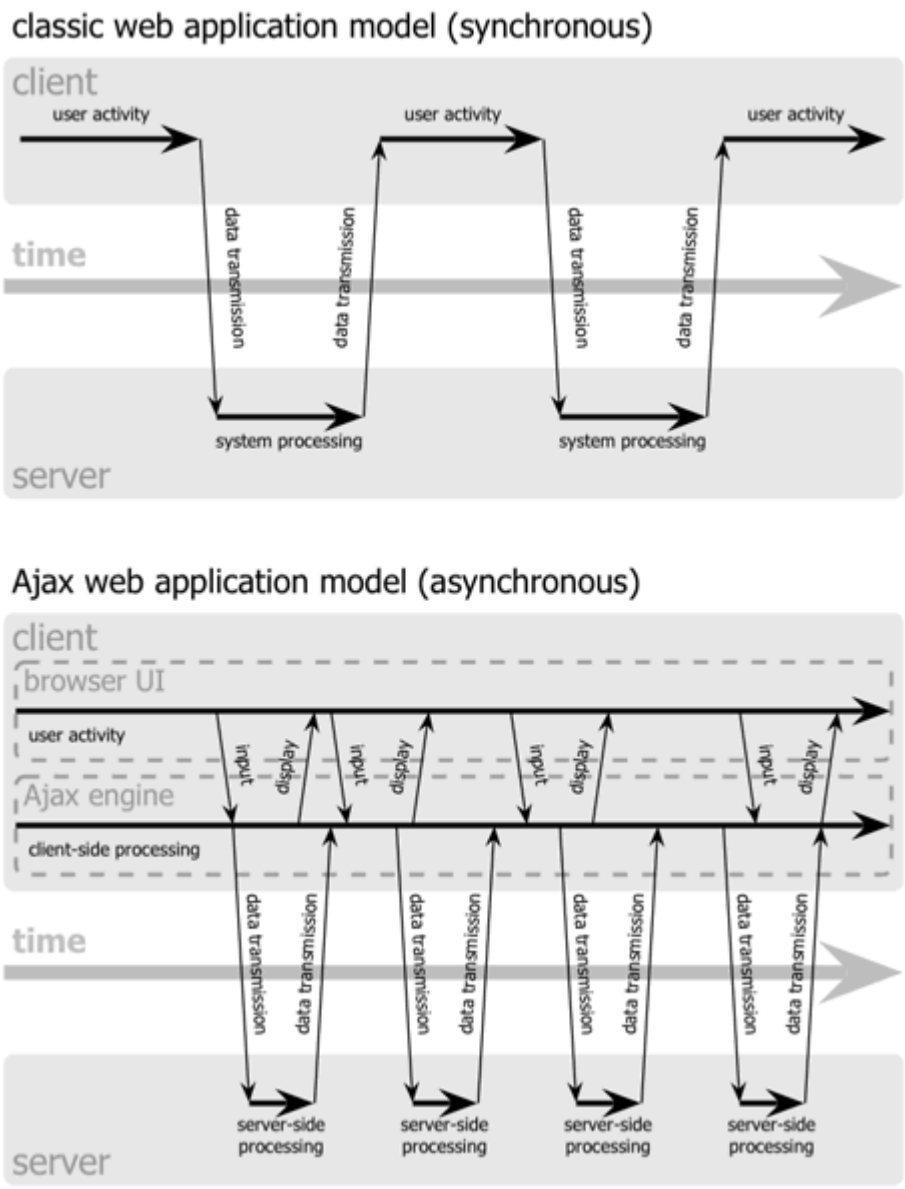

Figura 1.3. Comparación entre las comunicaciones síncronas de las aplicaciones web tradicionales y las comunicaciones asíncronas de las aplicaciones AJAX (Imagen original creada por Adaptive Path y utilizada con su permiso)

Las peticiones HTTP al servidor se sustituyen por peticiones JavaScript que se realizan al elemento encargado de AJAX. Las peticiones más simples no requieren intervención del servidor, por lo que la respuesta es inmediata. Si la interacción requiere una respuesta del servidor, la petición se realiza de forma asíncrona mediante AJAX. En este caso, la interacción del usuario tampoco se ve interrumpida por recargas de página o largas esperas por la respuesta del servidor.

Desde su aparición, se han creado cientos de aplicaciones web basadas en AJAX. En la mayoría de casos, AJAX puede sustituir completamente a otras técnicas como Flash. Además, en el caso de las aplicaciones web más avanzadas, pueden llegar a sustituir a las aplicaciones de escritorio.

A continuación se muestra una lista de algunas de las aplicaciones más conocidas basadas en AJAX:

- Gestores de correo electrónico: Gmail (http://www.gmail.com) , Yahoo Mail (http://mail.yahoo.com) , Windows Live Mail (http://www.hotmail.com) .
- Cartografía: Google Maps (http://maps.google.com) , Yahoo Maps (http://maps.yahoo.com) , Windows Live Local (http://maps.live.com) .
- Aplicaciones web y productividad: Google Docs (http://docs.google.com) , Zimbra (http://www.zimbra.com/) , Zoho (http://www.zoho.com/) .
- Otras: Netvibes (http://www.netvibes.com) [metapágina], Digg (http://www.digg.com) [noticias], Meebo (http://www.meebo.com) [mensajería], 30 Boxes (http://www.30boxes.com) [calendario], Flickr (http://www.flickr.com) [fotografía].

Capítulo 2. JavaScript básico

2.1. Sintaxis

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 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 su manejo (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 y el único problema es que la página no valida. Por el contrario, si en JavaScript se intercambian mayúsculas y minúsculas, las aplicaciones no funcionan correctamente.
- No se define el tipo de las variables: al definir 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 programa.
- No es obligatorio terminar cada sentencia con el carácter del punto y coma (`;`): al contrario de la mayoría de lenguajes de programación, en JavaScript no es obligatorio terminar cada sentencia con el carácter del punto y coma (`;`). No obstante, es muy recomendable seguir la tradición de terminar cada sentencia con el carácter `;`
- Se pueden incluir comentarios: los comentarios se utilizan para añadir alguna información relevante al código fuente del programa. Aunque no se visualizan por pantalla, su contenido se envía al navegador del usuario junto con el resto del programa, por lo que es necesario extremar las precauciones sobre el contenido de los comentarios.

JavaScript define dos tipos de comentarios: los de una sola línea y los que ocupan varias líneas. Los comentarios de una sola línea se definen añadiendo dos barras oblicuas (`//`) al principio de cada línea que forma el comentario:

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

También se pueden incluir varios comentarios seguidos de una sola línea:

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

Cuando un comentario ocupa más de una línea, es más eficiente utilizar los comentarios multilínea, que se definen encerrando el texto del comentario entre los caracteres `/*` y `*/`

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

Las normas completas de sintaxis y de cualquier otro aspecto relacionado con JavaScript se pueden consultar en el estándar oficial del lenguaje que está disponible en http://www.ecma-international.org/publications/standards/Ecma-262.htm

2.2. Variables

Las variables se definen mediante la palabra reservada `var`, que permite definir una o varias variables simultáneamente:

```
var variable1 = 16;
var variable2 = "hola", variable3 = "mundo";
var variable4 = 16, variable5 = "hola";
```

El nombre de las variables debe cumplir las dos siguientes condiciones:

- El primer carácter debe ser una letra o un guión bajo (_) o un dólar ($).
- El resto de caracteres pueden ser letras, números, guiones bajos (_) y símbolos de dólar ($).

No es obligatorio inicializar una variable al declararla:

```
var variable6;
```

Si la variable no se declara mediante el operador `var`, automáticamente se crea una variable global con ese identificador y su valor. Ejemplo:

```
var variable1 = 16;
variable2 = variable1 + 4;
```

En el ejemplo anterior, la `variable2` no ha sido declarada, por lo que al llegar a esa instrucción, JavaScript crea automáticamente una variable global llamada `variable2` y le asigna el valor correspondiente.

El ámbito de una variable (llamado *scope* en inglés) es la zona del programa en la que se define la variable. JavaScript define dos ámbitos para las variables: global y local.

El siguiente ejemplo ilustra el comportamiento de los ámbitos:

```
function muestraMensaje() {
    var mensaje = "Mensaje de prueba";
}

muestraMensaje();
alert(mensaje);
```

Cuando se ejecuta el código JavaScript anterior, su resultado no es el esperado, ya que no se muestra por pantalla ningún mensaje. La variable `mensaje` se ha definido dentro de la función y por tanto es una variable local que solamente está definida dentro de la función.

Cualquier instrucción que se encuentre dentro de la función puede hacer uso de la variable. Sin embargo, cualquier instrucción que se encuentre en otras funciones o fuera de cualquier función no tendrá definida la variable mensaje.

Además de variables locales, también existe el concepto de variable global, que está definida en cualquier punto del programa (incluso dentro de cualquier función).

```
var mensaje = "Mensaje de prueba";

function muestraMensaje() {
    alert(mensaje);
}
```

El código JavaScript anterior define una variable fuera de cualquier función. Este tipo de variables automáticamente se transforman en variables globales y están disponibles en cualquier punto del programa.

De esta forma, aunque en el interior de la función no se ha definido ninguna variable llamada `mensaje`, la variable global creada anteriormente permite que la instrucción `alert()` dentro de la función muestre el mensaje correctamente.

Si una variable se declara fuera de cualquier función, automáticamente se transforma en variable global independientemente de si se define utilizando la palabra reservada `var` o no. Sin embargo, en el interior de una función, las variables declaradas mediante `var` se consideran locales y el resto se transforman también automáticamente en variables globales.

Por lo tanto, el siguiente ejemplo si que funciona como se espera:

```
function muestraMensaje() {
    mensaje = "Mensaje de prueba";
}

muestraMensaje();
alert(mensaje);
```

En caso de colisión entre las variables globales y locales, dentro de una función prevalecen las variables locales:

```
var mensaje = "gana la de fuera";

function muestraMensaje() {
    var mensaje = "gana la de dentro";
    alert(mensaje);
}

alert(mensaje);
muestraMensaje();
alert(mensaje);
```

El código anterior muestra por pantalla los siguientes mensajes:

```
gana la de fuera
gana la de dentro
gana la de fuera
```

La variable local llamada `mensaje` dentro de la función tiene más prioridad que la variable global del mismo nombre, pero solamente dentro de la función.

Si no se define la variable dentro de la función con la palabra reservada `var`, en realidad se está modificando el valor de la variable global:

```
var mensaje = "gana la de fuera";

function muestraMensaje() {
    mensaje = "gana la de dentro";
    alert(mensaje);
}

alert(mensaje);
muestraMensaje();
alert(mensaje);
```

En este caso, los mensajes mostrados son:

```
gana la de fuera
gana la de dentro
gana la de dentro
```

La recomendación general es definir como variables locales todas las variables que sean de uso exclusivo para realizar las tareas encargadas a cada función. Las variables globales se utilizan para compartir variables entre funciones de forma rápida.

2.3. Palabras reservadas

Como cualquier otro lenguaje de programación, JavaScript utiliza una serie de palabras para crear las instrucciones que forman cada programa. Por este motivo, estas palabras se consideran reservadas y no se pueden utilizar como nombre de una variable o función.

El estándar ECMA-262 incluye la lista de las palabras reservadas que utiliza actualmente JavaScript y la lista de las palabras reservadas para su uso futuro.

Utilizadas actualmente: `break`, `else`, `new`, `var`, `case`, `finally`, `return`, `void`, `catch`, `for`, `switch`, `while`, `continue`, `function`, `this`, `with`, `default`, `if`, `throw`, `delete`, `in`, `try`, `do`, `instanceof`, `typeof`

Reservadas para su uso futuro:

`abstract`, `enum`, `int`, `short`, `boolean`, `export`, `interface`, `static`, `byte`, `extends`, `long`, `super`, `char`, `final`, `native`, `synchronized`, `class`, `float`, `package`, `throws`, `const`, `goto`, `private`, `transient`, `debugger`, `implements`, `protected`, `volatile`, `double`, `import`, `public`

2.4. Tipos de variables

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

2.4.1. Tipos primitivos

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.

2.4.1.1. El operador typeof

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 variable1 = 7;
typeof variable1;  // "number"
var variable2 = "hola mundo";
typeof variable2;  // "string"
```

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`.

2.4.1.2. Variables de tipo undefined

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"
```

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

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

2.4.1.3. Variables de tipo null

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;
```

2.4.1.4. Variables de tipo boolean

Además de variables de tipo *boolean*, también se suelen llamar variables lógicas y variables *booleanas*. 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.

2.4.1.5. Variables de tipo numérico

Las variables numéricas son muy utilizadas en las aplicaciones habituales, ya que permiten almacenar cualquier valor numérico. Si el número es entero, se indica directamente. Si el número es decimal, se debe utilizar el punto (`.`) para separar la parte entera de la decimal.

```
var variable1 = 10;
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:

Constante	Valor	Significado
`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;
```

2.4.1.6. Variables de tipo cadena de texto

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 `\`:

Si se quiere incluir...	**Se debe sustituir por...**
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";
```

2.4.1.7. Conversión entre tipos de variables

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";
parseFloat(variable4);  // devuelve 34.23
```

2.4.2. Tipos de referencia

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.

Aunque más adelante se explica en detalle, el siguiente ejemplo determina el número de caracteres de una cadena de texto:

```
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:

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

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

En el ejemplo anterior, se utiliza un tipo de dato primitivo 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. 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.

2.4.2.1. Variables de tipo Object

La clase `Object` por sí sola no es muy útil, ya que su única función es la de servir de base a partir de la cual heredan el resto de clases. Los conceptos fundamentales de los objetos son los constructores y la propiedad *prototype*, tal y como se explicarán en el siguiente capítulo.

Una utilidad práctica de `Object` es la conversión entre tipos de datos primitivos y sus correspondientes tipos de referencia:

```
var numero = new Object(5);             // numero es de tipo Number
var cadena = new Object("hola mundo");  // cadena es de tipo String
var conectado = new Object(false);      // conectado es de tipo Boolean
```

2.4.2.2. Variables de tipo Boolean

Utilizando el tipo de referencia `Boolean`, es posible crear objetos de tipo lógico o *booleano*:

```
var variable1 = new Boolean(false);
```

Sin embargo, en general no se utilizan objetos de tipo `Boolean` porque su comportamiento no siempre es idéntico al de los tipos de datos primitivos:

```
var variable1 = true, variable2 = false;
var variable3 = new Boolean(false);
variable2 && variable1;  // el resultado es false
variable3 && variable1;  // el resultado es true
```

El resultado de la última operación es realmente sorprendente, ya que se esperaba un resultado `false`. El problema reside en que los objetos no se comportan igual que los tipos primitivos. En una operación lógica, cualquier objeto que exista se convierte a `true`, independientemente de su valor.

Por este motivo, con los valores booleanos normalmente se utilizan tipos de datos primitivos en vez de objetos de tipo `Boolean`.

2.4.2.3. Variables de tipo Number

La clase `Number` permite definir variables de tipo numérico independientemente de si el valor es entero o decimal:

```
var variable1 = new Number(16);
var variable2 = new Number(3.141592);
```

Para obtener el valor numérico almacenado, se puede utilizar el método `valueOf()`:

```
var variable1 = new Number(16);
var variable2 = variable1.valueOf();  // variable2 = 16
```

Uno de los métodos más útiles para los números es `toFixed()`, que trunca el número de decimales de un número al valor indicado como parámetro:

```
var variable1 = new Number(3.141592);
var variable2 = variable1.toFixed();   // variable2 = 3
var variable3 = variable1.toFixed(2);  // variable3 = 3.14
var variable4 = variable1.toFixed(10); // variable4 = 3.1415920000
```

En ocasiones, el método `toFixed()` no funciona como debería, debido a los problemas que sufren la mayoría de lenguajes de programación con los números decimales (en realidad, se denominan *"números de coma flotante"*):

```
var numero1 = new Number(0.235);
var numero2 = new Number(1.235);

numero3 = numero1.toFixed(2);  // numero3 = 0.23
numero3 = numero2.toFixed(2);  // numero3 = 1.24
```

Como se ve en el ejemplo anterior, el redondeo de los decimales no funciona de forma consistente, ya que el número `5` a veces incrementa el decimal anterior y otras veces no. De la misma forma, se pueden producir errores de precisión en operaciones aparentemente sencillas, como en la siguiente multiplicación:

```
var numero1 = new Number(162.295);
var numero2 = numero1 * new Number(100);

// numero2 no es igual a 16229.5
// numero2 = 16229.499999999998
```

Los errores de redondeo afectan de la misma forma a las variables numéricas creadas con tipos de datos primitivos. En cualquier caso, al igual que sucede con `Boolean`, se recomienda utilizar el tipo de dato primitivo para los números, ya que la clase `Number` no aporta mejoras significativas.

2.4.2.4. Variables de tipo String

La clase `String` representa una cadena de texto, de forma similar a los tipos de datos primitivos:

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

El objeto de tipo `String` es el más complejo de JavaScript y contiene decenas de métodos y utilidades, algunos de los cuales se verán más adelante. Como ya se ha comentado, siempre que sea necesario JavaScript convierte de forma automática las cadenas de texto de dato primitivo a dato de referencia. De esta forma, no es obligatorio crear objetos de tipo `String` para acceder a todas las utilidades disponibles para las cadenas de texto.

2.4.2.5. Operador instanceof

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"
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.

2.5. Operadores

Las variables sólo se pueden utilizar para almacenar información. Sin embargo, es muy habitual que los programas tengan que manipular la información original para transformarla en otra información. Los operadores son los elementos básicos que se utilizan para modificar el valor de las variables y para combinar varios valores entre sí para obtener otro valor.

JavaScript define numerosos operadores, entre los que se encuentran los operadores matemáticos (suma, resta, multiplicación, división) y los operadores lógicos utilizados para realizar comparaciones (mayor que, igual, menor que).

2.5.1. Operador de asignación

El operador de asignación es el más utilizado y el más sencillo. Simplemente se utiliza para asignar a una variable un valor específico. El símbolo utilizado es = (no confundir con el operador ==):

```
var numero1 = 3;
var variable1 = "hola mundo";
```

2.5.2. Operadores de incremento y decremento

Solamente son válidos para las variables numéricas y son un método sencillo de incrementar o decrementar en 1 unidad el valor de una variable, tal y como se muestra en el siguiente ejemplo:

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

El anterior ejemplo es equivalente a:

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

De la misma forma, el operador -- se utiliza para decrementar el valor de la variable:

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

Como ya se supone, el anterior ejemplo es equivalente a:

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

Además de estos dos operadores, existen otros dos operadores similares pero que se diferencian en la forma en la que se realiza el incremento o decremento. En el siguiente ejemplo:

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

El resultado es el mismo que antes y puede parecer que es equivalente añadir 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.

2.5.3. Operadores lógicos

2.5.3.1. Negación

Uno de los operadores lógicos más utilizados es el de la negación. Se utiliza para obtener el valor lógico 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. Cuando la variable es de tipo booleano, obtener su valor lógico contrario es trivial:

variable	!variable
`true`	`false`
`false`	`true`

Por el contrario, si la variable almacena un número o una cadena de texto, no se puede obtener su valor lógico contrario de forma directa. En este caso, JavaScript convierte previamente la variable a un valor lógico y después obtiene su valor contrario.

Si la variable original contiene un número, su transformación en variable lógica es `false` si el número es `0` y `true` en cualquier otro caso. Si la variable original contiene una cadena de texto, su transformación en variable lógica es `false` si la cadena no contiene ningún carácter y `true` en cualquier otro caso:

```
var cantidad = 0;
vacio = !cantidad;  // vacio = true
cantidad = 2;
vacio = !cantidad;  // vacio = false

var mensaje = "";
```

```
sinMensaje = !mensaje;  // sinMensaje = true
mensaje = "hola mundo";
sinMensaje = !mensaje;  // sinMensaje = false
```

2.5.3.2. AND

La operación lógica `AND` combina dos valores booleanos para obtener como resultrado otro valor de tipo lógico. El resultado de la operación solamente es `true` si los dos operandos son `true`. El operador se indica mediante el símbolo `&&`:

variable1	variable2	variable1 && variable2
true	true	true
true	false	false
false	true	false
false	false	false

El siguiente ejemplo muestra cómo combinar valores mediante el operador `&&`:

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

2.5.3.3. OR

La operación lógica `OR` también combina dos valores booleanos para obtener como resultado otro valor de tipo lógico. El resultado de la operación es `true` si alguno de los dos operandos es `true`. El operador se indica mediante el símbolo `||`:

variable1	variable2	variable1 || variable2
true	true	true
true	false	true
false	true	true
false	false	false

El siguiente ejemplo muestra cómo combinar valores mediante el operador `||`:

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

2.5.4. Operadores matemáticos

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

Uno de los operadores matemáticos más singulares cuando se estudia por primera vez es el *módulo*, que calcula el resto de la división entera. Si se divide `10` entre `5`, la división es exacta y da un resultado de `2`. El resto de esa división es `0`, por lo que *"módulo de 10 y 5"* es igual a `0`.

Sin embargo, si se divide `9` y `5`, la división no es exacta, el resultado es `1` y el resto es `4`, por lo que *"módulo de 9 y 5"* es igual a `4`.

El módulo en JavaScript se indica mediante el símbolo `%`, que no debe confundirse con el porcentaje:

```
var numero1 = 10;
var numero2 = 5;
resultado = numero1 % numero2; // resultado = 0
numero1 = 9;
numero2 = 5;
resultado = numero1 % numero2; // resultado = 4
```

Aunque el operador módulo parece demasiado extraño como para ser útil, en muchas aplicaciones web reales se utiliza para realizar algunas técnicas habituales, tal y como se verá más adelante.

Los operadores matemáticos se pueden combinar con el operador de asginación para escribir de forma abreviada algunas operaciones comunes:

```
var numero1 = 5;
numero1 += 3; // numero1 = numero1 + 3 = 8
numero1 -= 1; // numero1 = numero1 - 1 = 4
numero1 *=2;  // numero1 = numero1 * 2 = 10
numero1 /= 2; // numero1 = numero1 / 2 = 2.5
numero1 %= 3; // numero1 = numero1 % 3 = 2
```

2.5.5. Operadores relacionales

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

El resultado de todas estas operaciones 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
```

El operador == es la mayor fuente de errores de programación incluso para los usuarios que ya tienen cierta experiencia desarrollando scripts. Si se quiere comparar el valor de dos variables, no se debe utilizar el operador =:

```
var numero1 = 5;
resultado = numero1 = 3; // numero1 = 3 y resultado = 3
```

Para comparar valores, se debe utilizar el operador == (con dos signos de igual):

```
var numero1 = 5;
resultado = numero1 == 3; // numero1 = 5 y resultado = false
```

Además de las variables numéricas, también se pueden utilizar variables de tipo cadena de texto con los operadores relacionales:

```
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 comparan cadenas de texto con los operadores > y <, el resultado obtenido puede ser poco intuitivo. JavaScript compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia entre las dos letras. Para determinar si una letra es mayor o menor que otra, se considera que:

```
A < B < ... < Z < a < b < ... < z
```

Además de los operadores básicos de igualdad, también existen los operadores *"idéntico"* (también llamado *"exactamente igual"*) y *"no idéntico"* (también llamado *"no exactamente igual"*). Cuando se utilizan estos operadores, no sólo se comparan los valores que almacenan las variables, sino que también se compara el tipo de cada variable.

```
var variable1 = 10;
var variable2 = "10";
variable1 == variable2;   // devuelve true
variable1 === variable2;  // devuelve false
```

El operador *"idéntico"* se indica mediante tres signos de igualdad (===) y devuelve `true` solamente si los dos operandos son exactamente iguales sin necesidad de realizar ninguna conversión. En el ejemplo anterior, la primera variable es de tipo numérico y su valor es `10`, mientras que la segunda variable es de tipo cadena de texto y su valor es `"10"`.

Si se utiliza el operador `==`, JavaScript convierte automáticamente el tipo de las variables para realizar la comparación. Al convertir la cadena de texto en variable numérica, se obtiene el valor `10`, por lo que los dos valores que se comparan son iguales y el resultado de la operación es `true`.

Sin embargo, en el caso del operador *"idéntico"*, las dos variables tienen que ser además del mismo tipo. Como la primera variable es de tipo numérico y la segunda es una cadena de texto, aunque sus valores son iguales, el resultado de la operación es `false`.

El operador *"no idéntico"* tiene un funcionamiento equivalente y se indica mediante los caracteres `!==`

2.6. Objetos nativos de JavaScript

JavaScript define algunos objetos de forma nativa, por lo que pueden ser utilizados directamente por los aplicaciones sin tener que declararlos. Además de las clases de tipo `Object`, `Number`, `Boolean` y `String` que ya se han visto, JavaScript define otras clases como `Function`, `Array`, `Date` y `RegExp`.

2.6.1. La clase Array

JavaScript permite definir los arrays de forma abreviada (como se verá en el capítulo de JavaScript avanzado) y también de forma tradicional mediante la clase `Array`:

```
var variable1 = new Array();
```

Si al instanciar la clase `Array()` se pasa un único argumento y es de tipo numérico, se crea un array con el número de elementos indicado:

```
var variable1 = new Array(10);
```

En el primer ejemplo, la `variable1` simplemente crea un array vacío, mientras que el segundo ejemplo crea un array de `10` elementos que todavía no están definidos. Los elementos de un array no tienen por qué ser todos del mismo tipo. Además, si al declarar el array se conocen los elementos que va a contener, es posible incluirlos en la declaración del array:

```
var variable1 = new Array(2, "hola", true, 45.34);
```

Otra forma habitual de añadir nuevos elementos al array es mediante la notación con corchetes que también se utiliza en otros lenguajes de programación:

```
var variable1 = new Array();
variable1[0] = 2;
variable1[1] = "hola";
variable1[2] = true;
variable1[3] = 45.34;
```

El primer elemento del array siempre ocupa la posición `0` (cero) y el tamaño del array aumenta de forma dinámica a medida que se añaden nuevos elementos.

Los arrays contienen decenas de propiedades y métodos muy útiles para manipular sus contenidos y realizar operaciones complejas, tal y como se verá más adelante.

2.6.2. La clase Date

Entre las utilidades que proporciona JavaScript, se encuentra la clase `Date` que permite representar y manipular valores relacionados con fechas. Para obtener la representación de la fecha actual, sólo es necesario instanciar la clase sin parámetros:

```
var fecha = new Date();
```

Además de la fecha, la instrucción anterior representa la hora en la que ha sido ejecutada la instrucción. Internamente, y como sucede en otros lenguajes de programación y otros sistemas, la fecha y hora se almacena como el número de milisegundos que han transcurrido desde el 1 de Enero de 1970 a las 00:00:00. Por este motivo, se puede construir una fecha cualquiera indicando el número de milisegundos a partir de esa referencia temporal:

```
var fecha = new Date(0);  //  "Thu Jan 01 1970 01:00:00 GMT+0100"
var fecha = new Date(10000000000000);  // "Sat Nov 20 2286 18:46:40 GMT+0100"
```

Afortunadamente, existen otras formas más sencillas de establecer la fecha y hora que se van a utilizar:

```
var fecha = new Date(2009, 5, 1);  // 1 de Junio de 2009 (00:00:00)
var fecha = new Date(2009, 5, 1, 19, 29, 39);  // 1 de Junio de 2009 (19:29:39)
```

El constructor de la clase `Date` permite establecer sólo una fecha o la fecha y hora a la vez. El formato es `(año, mes, dia)` o `(año, mes, dia, hora, minuto, segundo)`. Los meses se indican mediante un valor numérico que empieza en el `0` (Enero) y termina en el `11` (Diciembre). Los días del mes se cuentan de forma natural desde el día `1` hasta el `28`, `29`, `30` o `31` dependiendo de cada mes.

A continuación se muestran algunos de los métodos más útiles disponibles para la clase `Date`:

- `getTime()` – devuelve un número que representa la fecha como el número de milisegundos transcurridos desde la referencia de tiempos (1 de Enero de 1970).
- `getMonth()` – devuelve el número del mes de la fecha (empezando por `0` para Enero y acabando en `11` para Diciembre)
- `getFullYear()` – devuelve el año de la fecha como un número de 4 cifras.
- `getYear()` – devuelve el año de la fecha como un número de 2 cifras.
- `getDate()` – devuelve el número del día del mes.
- `getDay()` – devuelve el número del día de la semana (`0` para Domingo, `1` para Lunes, ..., `6` para Sábado)
- `getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()` – devuelve respectivamente las horas, minutos, segundos y milisegundos de la hora correspondiente a la fecha.

Cada método `get()` mostrado anteriormente tiene su correspondiente método `set()` que permite establecer el valor de cada una de las propiedades.

2.6.3. La clase Function

La clase `Function` raramente se utiliza de forma explícita para crear funciones. Utilizada de esta forma, se deben indicar todos los parámetros de la función y sus instrucciones como parámetros al instanciar la clase:

```
var miFuncion = new Function("a", "b", "return a+b;");
```

El último argumento de la llamada se considera como las instrucciones de la función y todos los anteriores son los argumentos de la misma. En cuanto se complica un poco el código de la función, este método se hace inviable.

2.7. Funciones

Las funciones de JavaScript no suelen definirse mediante la clase `Function`, sino que se crean mediante la palabra reservada `function`:

```
function suma(a, b) {
  return a+b;
}
```

No es obligatorio que las funciones tengan una instrucción de tipo `return` para devolver valores. De hecho, cuando una función no devuelve ningún valor o cuando en la instrucción `return` no se indica ningún valor, automáticamente se devuelve el valor `undefined`.

Para llamar a la función en cualquier instrucción, se indica su nombre junto con la lista de parámetros esperados:

```
var resultado = suma(2, 3);
```

Los parámetros que se pasan pueden estar definidos mediante operaciones que se evalúan antes de pasarlos a la función:

```
var resultado = suma(2+1, 3-4*3+4);
```

Como JavaScript no define tipos de variables, no es posible asegurar que los parámetros que se pasan a una función sean los del tipo adecuado para las operaciones que realiza la función.

Si a una función se le pasan más parámetros que los que ha definido, los parámetros sobrantes se ignoran. Si se pasan menos parámetros que los que ha definido la función, al resto de parámetros hasta completar el número correcto se les asigna el valor `undefined`.

Una función puede contener en su interior otras funciones anidadas:

```
function sumaCuadrados(a, b) {
  function cuadrado(x) { return x*x; }
  return cuadrado(a) + cuadrado(b);
}
```

La función anterior calcula la suma del cuadrado de dos números. Para ello, define en el interior de la función otra función que calcula el cuadrado del número que se le pasa. Para obtener el resultado final, la función `sumaCuadrados()` hace uso de la función anidada `cuadrado()`.

Las funciones también se pueden crear mediante lo que se conoce como *"function literals"* y que consiste en definir la función con una expresión en la que el nombre de la función es opcional. Debido a esta última característica, también se conocen como *funciones anónimas*. A continuación se muestra una misma función definida mediante el método tradicional y mediante una función anónima:

```
function suma(a, b) {
  return a+b;
}

var miFuncion = function(a, b) { return a+b; }
```

Las funciones anónimas son ideales para los casos en los que se necesita definir funciones sencillas que sólamente se utilizan una vez y para las que no es necesario crear una función tradicional con nombre. Más adelante en el capítulo de JavaScript avanzado se muestra en detalle el uso de funciones anónimas con objetos.

Como se ha comentado, cuando una función recibe menos parámetros de los que necesita, inicializa el valor del resto de parámetros a `undefined`. De esta forma, puede ser necesario proteger a la aplicación frente a posibles valores incorrectos en sus parámetros. El método habitual es realizar una comprobación sencilla:

```
function suma(a, b) {
  if(isNaN(b)) {
    b = 0;
  }
  return a + b;
}
```

La función del ejemplo anterior comprueba que `b` sea un número para poder realizar correctamente la suma. En caso de que no lo sea (es decir, que sea `null`, `undefined` o cualquier valor válido distinto de un número) se le asigna el valor `0` para que la función pueda devolver un resultado válido.

JavaScript permite prescindir de la comprobación anterior y obtener el mismo resultado mediante el siguiente truco que hace uso del operador `OR` (`||`):

```
function suma(a, b) {
  b = b || 0;
  return a + b;
}
```

En el ejemplo anterior, si a la función no se le pasa el parámetro `b`, automáticamente se asigna el valor `0` a ese parámetro. El truco funciona porque el comportamiento del operador lógico `OR` (y también el del operador `AND`) es más complejo de lo que se ha explicado anteriormente.

En realidad, el operador lógico `OR` funciona de la siguiente manera:

1. Si el primer operando es `true` o cualquier otro valor que se puede transformar en `true`, se devuelve directamente el valor de ese operando.
2. En otro caso, se evalúa el segundo operando y se devuelve directamente su valor.

Por lo tanto:

```
alert(true || false);  // muestra true
alert(3 || false);     // muestra 3
alert(true || 5);      // muestra true
alert(false || true);  // muestra true
alert(false || 5);     // muestra 5
alert(3 || 5);         // muestra 3
```

De esta forma, si se utilizan las siguientes llamadas a la función:

```
suma(3);
suma(3, null);
suma(3, false);
```

En todos los casos anteriores la variable `b` vale `0`. Si no se indica un parámetro, su valor es `undefined`, que se transforma en `false` y por tanto el resultado de `b || 0` es `0`. Si se indica `null` como valor del parámetro, también se transforma en `false`, por lo que nuevamente el resultado de `b || 0` es `0`. Por último, si se indica directamente el valor `false` al parámetro, también provoca que el resultado de `b || 0` sea `0`. En cualquier otro caso, el parámetro `b` valdrá lo mismo que se la haya pasado en la llamada a la función.

Como el número de argumentos que se pasan a una función de JavaScript puede ser variable e independiente del número de parámetros incluidos en su definición, JavaScript proporciona una variable especial que contiene todos los parámetros con los que se ha invocado a la función. Se trata de un array que se llama `arguments` y solamente está definido dentro de cualquier función.

```
function suma(a, b) {
  alert(arguments.length);
  alert(arguments[2]);
  return a + b;
}

suma(3, 5);
```

La propiedad `arguments.length` devuelve el número de parámetros con los que se ha llamado a la función. En el caso del ejemplo anterior, se mostraría el valor `2`. Como `arguments` es un array, se puede acceder directamente a cualquier parámetro mediante la notación tradicional de los arrays. En este caso, el valor `arguments[2]` devuelve `undefined`, ya que la función se llama con dos parámetros y por tanto el tercer parámetro no está definido.

El array `arguments` permite crear funciones con un número variable de argumentos:

```
function mayor() {
  var elMayor = arguments[0];
  for(var i=1; i<arguments.length; i++) {
    if(arguments[i] > elMayor) {
      elMayor = arguments[i];
     }
  }
  return elMayor;
}
```

```
var variable1 = mayor(1, 3, 5, 8);
var variable2 = mayor(4, 6, 8, 1, 2, 3, 4, 5);
```

Técnicamente, `arguments` no es un array, sino que es un objeto de tipo `Arguments`. Sin embargo, por sus propiedades y sus métodos de acceso, se puede considerar como si fuera un array.

Una última propiedad del objeto `arguments` que no suele utilizarse habitualmente, pero que puede ser necesaria en ocasiones es la propiedad `callee`. La propiedad `callee` hace referencia a la función que se está ejecutando. En el siguiente ejemplo se utiliza la propiedad `callee` para mostrar el código fuente de la función que se está ejecutando:

```
function suma(a, b) {
  alert(arguments.callee);
  return a + b;
}

suma(3, 5);
```

La propiedad `callee` se puede utilizar para determinar el número de parámetros que espera la función:

```
function suma(a, b) {
  alert(arguments.callee.length);
  alert(arguments.length);
  return a + b;
}

suma(3, 5, 7, 9);
```

La propiedad `arguments.callee.length` indica el número de parámetros que se incluyen en la definición de la función, en este caso 2. Como se ha visto anteriormente, la propiedad `arguments.length` indica el número de parámetros con los que se ha llamado a la función, en este caso 4.

2.8. Funciones y propiedades básicas de JavaScript

JavaScript incluye numerosas propiedades y métodos muy útiles para cada uno de los tipos de variables y clases que define.

2.8.1. Cadenas de texto

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 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 o indicarlos de 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(letra)`, calcula la primera 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.indexOf('a'); // posicion = 3
posicion = mensaje.indexOf('b'); // posicion = -1
```

La función `indexOf()` comienza su búsqueda desde el principio de la palabra y solo devuelve la primera posición de todas las existentes. Su función análoga es `lastIndexOf()`.

`lastIndexOf(letra)`, 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 solo 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"
```

Si 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 un 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 una cadena de texto dividiendo sus trozos a partir del carácter delimitador 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"]
```

2.8.2. Arrays

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`:

```
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 una unidad.

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

`push()`, añade un elemento al final del array. El array original se modifica y aumenta su longitud una unidad. 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 modifica y su longitud disminuye una unidad.

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

`unshift()`, añade un elemento al principio del array. El array original se modifica y aumenta su longitud en una unidad. 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]
```

Capítulo 3. JavaScript avanzado

3.1. Objetos

Al igual que sucede con otros lenguajes de programación, los objetos se emplean en JavaScript para organizar el código fuente de una forma más clara y para encapsular métodos y funciones comunes. La forma más sencilla de crear un objeto es mediante la palabra reservada `new` seguida del nombre de la clase que se quiere instanciar:

```
var elObjeto = new Object();
var laCadena = new String();
```

El objeto `laCadena` creado mediante el objeto nativo `String` permite almacenar una cadena de texto y aprovechar todas las herramientas y utilidades que proporciona JavaScript para su manejo. Por otra parte, la variable `elObjeto` almacena un objeto genérico de JavaScript, al que se pueden añadir propiedades y métodos propios para definir su comportamiento.

3.1.1. Definición de un objeto

Técnicamente, un objeto de JavaScript es un array asociativo formado por las propiedades y los métodos del objeto. Así, la forma más directa para definir las propiedades y métodos de un objeto es mediante la *notación de puntos* de los arrays asociativos.

Un array asociativo es aquel en el que cada elemento no está asociado a su posición numérica dentro del array, sino que está asociado a otro valor específico. Los valores de los arrays *normales* se asocian a índices que siempre son numéricos. Los valores de los arrays asociativos se asocian a claves que siempre son cadenas de texto.

La forma tradicional de definir los arrays asociativos es mediante la clase `Array`:

```
var elArray = new Array();
elArray['primero'] = 1;
elArray['segundo'] = 2;

alert(elArray['primero']);
alert(elArray[0]);
```

El primer `alert()` muestra el valor `1` correspondiente al valor asociado con la clave `primero`. El segundo `alert()` muestra `undefined`, ya que como no se trata de un array *normal*, sus elementos no se pueden acceder mediante su posición numérica.

Afortunadamente, existen métodos alternativos abreviados para crear array asociativos. El ejemplo anterior se puede rehacer de la siguiente forma:

```
var elArray = new Array();
elArray.primero = 1;
elArray.segundo = 2;

alert(elArray['primero']);
```

```
alert(elArray.primero);
alert(elArray[0]);
```

El método seguido en el ejemplo anterior para crear el array asociativo se denomina *"notación de puntos"*. Para acceder y/o establecer cada valor, se indica el nombre del array seguido de un punto y seguido del nombre de cada clave. De forma genérica, la notación de puntos tiene el siguiente formato:

```
nombreArray.nombreClave = valor;
```

Para acceder a un determinado valor, también se puede utilizar la notación de puntos en vez de la tradicional notación de los arrays, de forma que las dos instrucciones siguientes son equivalentes:

```
elArray['primero'];
elArray.primero;
```

Más adelante se muestra otra forma aún más abreviada y directa de establecer el valor tanto de los arrays *"normales"* como de los arrays asociativos.

3.1.1.1. Propiedades

Como los objetos son en realidad arrays asociativos que almacenan sus propiedades y métodos, la forma más directa para definir esas propiedades y métodos es la notación de puntos:

```
elObjeto.id = "10";
elObjeto.nombre = "Objeto de prueba";
```

Al contrario de lo que sucede en otros lenguajes orientados a objetos, como por ejemplo Java, para asignar el valor de una propiedad no es necesario que la clase tenga definida previamente esa propiedad.

También es posible utilizar la notación tradicional de los arrays para definir el valor de las propiedades:

```
elObjeto['id'] = "10";
elObjeto['nombre'] = "Objeto de prueba";
```

3.1.1.2. Métodos

Además de las propiedades, los métodos de los objetos también se pueden definir mediante la notación de puntos:

```
elObjeto.muestraId = function() {
  alert("El ID del objeto es " + this.id);
}
```

Los métodos se definen asignando funciones al objeto. Si la función no está definida previamente, es posible crear una función anónima para asignarla al nuevo método del objeto, tal y como muestra el ejemplo anterior.

Uno de los aspectos más importantes del ejemplo anterior es el uso de la palabra reservada `this`. La palabra `this` se suele utilizar habitualmente dentro de los métodos de un objeto y siempre hace referencia al objeto que está llamado a ese método.

De esta forma, en el ejemplo anterior:

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.muestraId = function() {
  alert("El ID del objeto es " + this.id);
}
```

Dentro del método, `this` apunta al objeto que llama a ese método. En este caso, `this` hace referencia a `elObjeto`. Por tanto, la instrucción del método `muestraId` es equivalente a indicar:

```
alert("El ID del objeto es " + elObjeto.id);
```

El uso de `this` es imprescindible para crear aplicaciones reales. El motivo es que nunca se puede suponer el nombre que tendrá la variable (el objeto) que incluye ese método. Como los programadores pueden elegir libremente el nombre de cada objeto, no hay forma de asegurar que la siguiente instrucción funcione siempre correctamente:

```
alert("El ID del objeto es " + elObjeto.id);
```

Si el objeto se llamara `otroObjeto`, el código anterior no funcionaría correctamente. Sin embargo, utilizando la palabra reservada `this`, el método funciona siempre bien independientemente del nombre del objeto.

Además, la palabra `this` se debe utilizar siempre que se quiera acceder a una propiedad de un objeto, ya que en otro caso, no se está accediendo correctamente a la propiedad:

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.muestraId = function() {
  alert("El ID del objeto es "+ id);
}
```

Si se ejecuta el ejemplo anterior, se muestra el error `"id is not defined"` (*la variable id no está definida*).

Además de las funciones anónimas, también es posible asignar a los métodos de un objeto funciones definidas con anterioridad:

```
function obtieneId() {
  return this.id;
}

elObjeto.obtieneId = obtieneId;
```

Para asignar una función externa al método de un objeto, sólo se debe indicar el nombre de la función externa sin paréntesis. Si se utilizaran los paréntesis:

```
function obtieneId() {
  return this.id;
}

elObjeto.obtieneId = obtieneId();
```

En el ejemplo anterior, se ejecuta la función `obtieneId()` y el resultado de la ejecución se asigna a la propiedad `obtieneId` del objeto. Así, el objeto no tendría un método llamado `obtieneId`, sino una propiedad con ese nombre y con un valor igual al resultado devuelto por la función externa.

Por otra parte, no es obligatorio que el método del objeto se llame igual que la función externa, aunque es posible que así sea.

A continuación se muestra un objeto completo formado por varias propiedades y métodos y creado con la notación de puntos:

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.nombre = "Objeto de prueba";
elObjeto.muestraId = function() {
  alert("El ID del objeto es "+ this.id);
}
elObjeto.muestraNombre = function() {
  alert(this.nombre);
}
```

Siguiendo este mismo procedimiento, es posible crear objetos complejos que contengan otros objetos:

```
var Aplicacion = new Object();

Aplicacion.Modulos = new Array();
Aplicacion.Modulos[0] = new Object();
Aplicacion.Modulos[0].titulo = "Lector RSS";

var inicial = new Object();
inicial.estado = 1;
inicial.publico = 0;
inicial.nombre = "Modulo_RSS";
inicial.datos = new Object();

Aplicacion.Modulos[0].objetoInicial = inicial;
```

En el ejemplo anterior, se define un objeto principal llamado `Aplicacion` que a su vez contiene varios objetos. La propiedad `Modulos` de la aplicación es un array en el que cada elemento es un objeto que representa a un módulo. A su vez, cada objeto `Modulo` tiene una propiedad llamada `titulo` y otra llamada `objetoInicial` que también es un objeto con las propiedades y valores iniciales del módulo.

La notación tradicional de JavaScript puede llegar a ser tediosa cuando se desarrollan aplicaciones complejas con objetos que contienen otros muchos objetos y arrays. Por este motivo, JavaScript define un método alternativo de notación llamado JSON (*JavaScript Object Notation*) y que se verá más adelante.

3.1.1.3. Métodos apply() y call()

JavaScript define un par de métodos denominados `apply()` y `call()` que son muy útiles para las funciones. Ambos métodos permiten ejecutar una función como si fuera un método de otro

objeto. La única diferencia entre los dos métodos es la forma en la que se pasan los argumentos a la función.

El siguiente ejemplo muestra cómo utilizar el método `call()` para ejecutar una función como si fuera un método del objeto `elObjeto`:

```
function miFuncion(x) {
  return this.numero + x;
}

var elObjeto = new Object();
elObjeto.numero = 5;

var resultado = miFuncion.call(elObjeto, 4);
alert(resultado);
```

El primer parámetro del método `call()` es el objeto sobre el que se va a ejecutar la función. Como la función se trata como si fuera un método del objeto, la palabra reservada `this` hace referencia al objeto indicado en la llamada a `call()`. De esta forma, si en la función se utiliza `this.numero`, en realidad se está obteniendo el valor de la propiedad `numero` del objeto.

El resto de parámetros del método `call()` son los parámetros que se pasan a la función. En este caso, solamente es necesario un parámetro, que es el número que se sumará a la propiedad `numero` del objeto.

El método `apply()` es idéntico al método `call()`, salvo que en este caso los parámetros se pasan como un array:

```
function miFuncion(x) {
  return this.numero + x;
}
var elObjeto = new Object();
elObjeto.numero = 5;

var resultado = miFuncion.apply(elObjeto, [4]);
alert(resultado);
```

3.1.2. Notación JSON

JSON (*JavaScript Object Notation*) es un formato sencillo para el intercambio de información. El formato JSON permite representar estructuras de datos (arrays) y objetos (arrays asociativos) en forma de texto. La notación de objetos mediante JSON es una de las características principales de JavaScript y es un mecanismo definido en los fundamentos básicos del lenguaje.

En los últimos años, JSON se ha convertido en una alternativa al formato XML, ya que es más fácil de leer y escribir, además de ser mucho más conciso. No obstante, XML es superior técnicamente porque es un lenguaje de marcado, mientras que JSON es simplemente un formato para intercambiar datos.

La especificación completa de JSON se puede consultar en RFC 4627 (http://tools.ietf.org/html/rfc4627) y su tipo MIME oficial es `application/json`.

Como ya se sabe, la notación tradicional de los arrays es tediosa cuando existen muchos elementos:

```
var modulos = new Array();
modulos[0] = "Lector RSS";
modulos[1] = "Gestor email";
modulos[2] = "Agenda";
modulos[3] = "Buscador";
modulos[4] = "Enlaces";
```

Para crear un array normal mediante JSON, se indican sus valores separados por comas y encerrados entre corchetes. Por lo tanto, el ejemplo anterior se puede reescribir de la siguiente manera utilizando la notación JSON:

```
var modulos = ["Lector RSS", "Gestor email", "Agenda", "Buscador", "Enlaces"];
```

Por su parte, la notación tradicional de los arrays asociativos es igual de tediosa que la de los arrays normales:

```
var modulos = new Array();
modulos.titulos = new Array();
modulos.titulos['rss'] = "Lector RSS";
modulos.titulos['email'] = "Gestor de email";
modulos.titulos['agenda'] = "Agenda";
```

En este caso, se puede utilizar la notación de puntos para abreviar ligeramente su definición:

```
var modulos = new Array();
modulos.titulos = new Array();
modulos.titulos.rss = "Lector RSS";
modulos.titulos.email = "Gestor de email";
modulos.titulos.agenda = "Agenda";
```

En cualquier caso, la notación JSON permite definir los arrays asociativos de una forma mucho más concisa. De hecho, el ejemplo anterior se puede reescribir de la siguiente manera utilizando la notación JSON:

```
var modulos = new Array();
modulos.titulos = {rss: "Lector RSS", email: "Gestor de email", agenda: "Agenda"};
```

La notación JSON para los arrays asociativos se compone de tres partes:

1. Los contenidos del array asociativo se encierran entre llaves ({ y })
2. Los elementos del array se separan mediante una coma (,)
3. La clave y el valor de cada elemento se separan mediante dos puntos (:)

Si la clave no contiene espacios en blanco, es posible prescindir de las comillas que encierran sus contenidos. Sin embargo, las comillas son obligatorias cuando las claves pueden contener espacios en blanco:

```
var titulosModulos = {"Lector RSS": "rss", "Gestor de email": "email", "Agenda":
"agenda"};
```

Como JavaScript ignora los espacios en blanco sobrantes, es posible reordenar las claves y valores para que se muestren más claramente en el código fuente de la aplicación. El ejemplo

anterior se puede rehacer de la siguiente manera añadiendo nuevas líneas para separar los elementos y añadiendo espacios en blanco para tabular las claves y para alinear los valores:

```
var titulos = {
  rss:    "Lector RSS",
  email:  "Gestor de email",
  agenda: "Agenda"
};
```

Combinando la notación de los arrays simples y asociativos, es posible construir objetos muy complejos de forma sencilla. Con la notación tradicional, un objeto complejo se puede crear de la siguiente manera:

```
var modulo = new Object();
modulo.titulo = "Lector RSS";
modulo.objetoInicial = new Object();
modulo.objetoInicial.estado = 1;
modulo.objetoInicial.publico = 0;
modulo.objetoInicial.nombre = "Modulo_RSS";
modulo.objetoInicial.datos = new Object();
```

Utilizando JSON, es posible reescribir el ejemplo anterior de forma mucho más concisa:

```
var modulo = {
  titulo : "Lector RSS",
  objetoInicial : { estado : 1, publico : 0, nombre : "Modulo RSS", datos : {} }
};
```

Los objetos se pueden definir en forma de pares clave/valor separados por comas y encerrados entre llaves. Para crear objetos vacíos, se utilizan un par de llaves sin contenido en su interior `{}`.

A continuación se muestra la notación JSON genérica para crear arrays y objetos:

Arrays

```
var array = [valor1, valor2, valor3, ..., valorN];
```

Objetos

```
var objeto = { clave1: valor1, clave2: valor2, clave3: valor3, ..., claveN: valorN };
```

La notación abreviada se puede combinar para crear arrays de objetos, objetos con arrays, objetos con objetos y arrays, etc. A continuación se muestran algunos ejemplos de aplicaciones web reales que crean objetos mediante esta notación.

Ejemplo extraído del código fuente de Blinksale.com

```
var Invoice = Class.create();
Invoice.prototype = {
  initialize: function(line_html, currency_symbol) {
    this.line_html = line_html;
    this.currency_symbol = currency_symbol;
    this.line_index = 1;
    this.update();
  },

  kinds: ['Service','Hours','Days','Product'],
```

```
  change_kind: function(i) {
    if($F('lines_'+i+'_kind')=='Hours') {
      $('lines_'+i+'_unit_price').value = $F('default_hourly_rate');
      this.update();
    }
  },

  focus_num: function() {
    $('invoice_number').focus();
  },

  use_freight: function() {
    return $('invoice_use_freight').checked
  },

  freight: function() {
    return this.use_freight() ? Number(noCommas($('invoice_freight').value)) : 0 ;
  },
  ...
}
```

Ejemplo extraído del código fuente de Gmail.com

```
pf.prototype = {
  Ed:function(a){this.hm=a},
  dh:function(){if(this.eb.Th>0) {var a=Math.random()*100; return a<this.eb.Th} return
false },
  Sg:function(a,b,c){ this.Vd=2; this.kb=Ic(a,true); this.Pc=b; this.Vl=c; this.Be() },
  ne:function(a,b,c){ this.Vd=2; this.kb=Ic(a,true); this.Pc=null; this.Vl=b;
if(c){this.vm=false} this.Be()},
  ...
}
```

Ejemplo extraído del código fuente de la librería prototype.js

```
var Prototype = {
  Version: '1.6.0.2',

  ScriptFragment: '<script[^>]*>([\\S\\s]*?)<\/script>',
  JSONFilter: /^\/\*-secure-([\s\S]*)\*\/\s*$/,

  emptyFunction: function() { },
  K: function(x) { return x }
};
```

Ejemplo extraído del código fuente de Netvibes.com

```
var App = new Object();
App.mode = 'userPage';
App.lang = 'es';
App.Modules = new Object();

App.Modules.RssReaderInfos = {
  infos: App.Loc.defaultRssReader_infos,
  defaultObj: {status:1, share:0, title:"", moduleName:"RssReader", data:{}}
}
```

```
App.Modules.GmailInfos = {
  title: App.Loc.defaultGmail_title,
  infos: App.Loc.defaultGmail_infos,
  defaultObj:{status:1, share:0, title:App.Loc.defaultGmail_title, moduleName:"Gmail",
data:{}},
  path: NV_PATH+"modules/gmail/gmail.js?v=5",
  ico: NV_PATH+"img/gmail.gif"
}
App.Modules.WeatherInfos = {
  title: App.Loc.defaultWeather_title,
  infos: App.Loc.defaultWeather_infos,
  defaultObj:{status:1, share:0, title:App.Loc.defaultWeather_title,
moduleName:"Weather", data:{town:"FRXX0076"}},
  path: NV_PATH+"modules/weather/weather.js?v=2",
  ico: NV_PATH+"img/weather.gif"
}
```

Ejemplo extraído del código fuente de Writeboard.com

```
// ================================================
// EXPORT MANAGER
// ================================================
var exportManager = {
  show: function() {
    $('exportButton').src = "/images/b-export-on.gif"
    showElement('download_export')
  },

  hide: function() {
    $('exportButton').src = "/images/b-export.gif"
    hideElement('download_export')
  },

  toggle: function() {
    Element.visible('download_export') ? this.hide() : this.show()
  }
}
```

Ejemplo extraído del código fuente del framework Dojo

```
var dojo;
if(dj_undef("dojo")){ dojo = {}; }

dojo.version = {
  major: 0, minor: 2, patch: 2, flag: "",
  revision: Number("$Rev: 2836 $".match(/[0-9]+/)[0]),
  toString: function() {
    with (dojo.version) {
      return major + "." + minor + "." + patch + flag + " (" + revision + ")";
    }
  }
};
```

A partir de los ejemplos anteriores, se deduce que la forma habitual para definir los objetos en JavaScript se basa en el siguiente modelo creado con la notación JSON:

```
var objeto = {
  "propiedad1": valor_simple_1,
  "propiedad2": valor_simple_2,
  "propiedad3": [array1_valor1, array1_valor2],
  "propiedad4": { "propiedad anidada": valor },
  "metodo1": nombre_funcion_externa,
  "metodo2": function() { ... },
  "metodo3": function() { ... },
  "metodo4": function() { ... }
};
```

En un mismo objeto se puede utilizar de forma simultánea la notación tradicional de JavaScript y la notación JSON:

```
var libro = new Object();
libro.numeroPaginas = 150;
libro.autores = [ {id: 50}, {id: 67} ];
```

El ejemplo anterior se puede reescribir utilizando solamente la notación tradicional:

```
var libro = { numeroPaginas: 150 };
libro.autores = new Array();
libro.autores[0] = new Object();
libro.autores[0].id = 50;
libro.autores[1] = new Object();
libro.autores[1].id = 67;
```

El ejemplo anterior también se puede reescribir utilizando exclusivamente la notación JSON:

```
var libro = { numeroPaginas: 150, autores: [{id: 50}, {id: 67}] };
```

Ejercicio 1

Definir la estructura de un objeto que almacena una factura. Las facturas están formadas por la información de la propia empresa (nombre de la empresa, dirección, teléfono, NIF), la información del cliente (similar a la de la empresa), una lista de elementos (cada uno de los cuales dispone de descripción, precio, cantidad) y otra información básica de la factura (importe total, tipo de iva, forma de pago).

Una vez definidas las propiedades del objeto, añadir un método que calcule el importe total de la factura y actualice el valor de la propiedad correspondiente. Por último, añadir otro método que muestre por pantalla el importe total de la factura.

3.2. Clases

Los objetos que se han visto hasta ahora son una simple colección de propiedades y métodos que se definen para cada objeto individual. Sin embargo, en la programación orientada a objetos, el concepto fundamental es el de clase.

La forma habitual de trabajo consiste en definir clases a partir de las cuales se crean los objetos con los que trabajan las aplicaciones. Sin embargo, JavaScript no permite crear clases similares a las de lenguajes como Java o C++. De hecho, la palabra `class` sólo está reservada para su uso en futuras versiones de JavaScript.

A pesar de estas limitaciones, es posible utilizar en JavaScript unos elementos parecidos a las clases y que se denominan *pseudoclases*. Los conceptos que se utilizan para simular las clases son las funciones constructoras y el *prototype* de los objetos.

3.2.1. Funciones constructoras

Al contrario que en los lenguajes orientados a objetos, en JavaScript no existe el concepto de constructor. Por lo tanto, al definir una clase no se incluyen uno o varios constructores. En realidad, JavaScript emula el funcionamiento de los constructores mediante el uso de funciones.

En el siguiente ejemplo, se crea un objeto genérico y un objeto de tipo array:

```
var elObjeto = new Object();
var elArray = new Array(5);
```

En los dos casos, se utiliza la palabra reservada `new` y el nombre del tipo de objeto que se quiere crear. En realidad, ese nombre es el nombre de una función que se ejecuta para crear el nuevo objeto. Además, como se trata de funciones, es posible incluir parámetros en la creación del objeto.

JavaScript utiliza funciones para simular los constructores de objetos, por lo que estas funciones se denominan *"funciones constructoras"*. El siguiente ejemplo crea una función llamada `Factura` que se utiliza para crear objetos que representan una factura.

```
function Factura(idFactura, idCliente) {
  this.idFactura = idFactura;
  this.idCliente = idCliente;
}
```

Cuando se crea un objeto es habitual pasar al constructor de la clase una serie de valores para inicializar algunas propiedades. Este concepto también se utiliza en JavaSript, aunque su realización es diferente. En este caso, la función constructora inicializa las propiedades de cada objeto mediante el uso de la palabra reservada `this`.

La función constructora puede definir todos los parámetros que necesita para construir los nuevos objetos y posteriormente utilizar esos parámetros para la inicialización de las propiedades. En el caso anterior, la factura se inicializa mediante el identificador de factura y el identificador de cliente.

Después de definir la función anterior, es posible crear un objeto de tipo `Factura` y simular el funcionamiento de un constructor:

```
var laFactura = new Factura(3, 7);
```

Así, el objeto `laFactura` es de tipo `Factura`, con todas sus propiedades y métodos y se puede acceder a ellos utilizando la notación de puntos habitual:

```
alert("cliente = " + laFactura.idCliente + ", factura = " + laFactura.idFactura);
```

Normalmente, las funciones constructoras no devuelven ningún valor y se limitan a definir las propiedades y los métodos del nuevo objeto.

3.2.2. Prototype

Las funciones constructoras no solamente pueden establecer las propiedades del objeto, sino que también pueden definir sus métodos. Siguiendo con el ejemplo anterior, se puede crear un objeto completo llamado `Factura` con sus propiedades y métodos:

```
function Factura(idFactura, idCliente) {
  this.idFactura = idFactura;
  this.idCliente = idCliente;

  this.muestraCliente = function() {
    alert(this.idCliente);
  }

  this.muestraId = function() {
    alert(this.idFactura);
  }
}
```

Una vez definida la *pseudoclase* mediante la función constructora, ya es posible crear objetos de ese tipo. En el siguiente ejemplo se crean dos objetos diferentes y se emplean sus métodos:

```
var laFactura = new Factura(3, 7);
laFactura.muestraCliente();
var otraFactura = new Factura(5, 4);
otraFactura.muestraId();
```

Incluir los métodos de los objetos como funciones dentro de la propia función constructora, es una técnica que funciona correctamente pero que tiene un gran inconveniente que la hace poco aconsejable.

En el ejemplo anterior, las funciones `muestraCliente()` y `muestraId()` se crean de nuevo por cada objeto creado. En efecto, con esta técnica, cada vez que se instancia un objeto, se definen tantas nuevas funciones como métodos incluya la función constructora. La penalización en el rendimiento y el consumo excesivo de recursos de esta técnica puede suponer un inconveniente en las aplicaciones profesionales realizadas con JavaScript.

Afortunadamente, JavaScript incluye una propiedad que no está presente en otros lenguajes de programación y que soluciona este inconveniente. La propiedad se conoce con el nombre de *prototype* y es una de las características más poderosas de JavaScript.

Todos los objetos de JavaScript incluyen una referencia interna a otro objeto llamado *prototype* o *"prototipo"*. Cualquier propiedad o método que contenga el objeto prototipo, está presente de forma automática en el objeto original.

Realizando un símil con los lenguajes orientados a objetos, es como si cualquier objeto de JavaScript heredara de forma automática todas las propiedades y métodos de otro objeto llamado prototype. Cada tipo de objeto diferente hereda de un objeto prototype diferente.

En cierto modo, se puede decir que el prototype es el *molde* con el que se fabrica cada objeto de ese tipo. Si se modifica el molde o se le añaden nuevas características, todos los objetos fabricados con ese molde tendrán esas características.

Normalmente los métodos no varían de un objeto a otro del mismo tipo, por lo que se puede evitar el problema de rendimiento comentado anteriormente añadiendo los métodos al prototipo a partir del cual se crean los objetos.

Si se considera de nuevo la clase `Factura` del ejemplo anterior:

```
function Factura(idFactura, idCliente) {
  this.idFactura = idFactura;
  this.idCliente = idCliente;

  this.muestraCliente = function() {
    alert(this.idCliente);
  }

  this.muestraId = function() {
    alert(this.idFactura);
  }
}
```

La clase anterior que incluye los métodos en la función constructora, se puede reescribir utilizando el objeto prototype:

```
function Factura(idFactura, idCliente) {
  this.idFactura = idFactura;
  this.idCliente = idCliente;
}

Factura.prototype.muestraCliente = function() {
  alert(this.idCliente);
}

Factura.prototype.muestraId = function() {
  alert(this.idFactura);
}
```

Para incluir un método en el prototipo de un objeto, se utiliza la propiedad `prototype` del objeto. En el ejemplo anterior, se han añadido los dos métodos del objeto en su prototipo. De esta forma, todos los objetos creados con esta función constructora incluyen por defecto estos dos métodos. Además, no se crean dos nuevas funciones por cada objeto, sino que se definen únicamente dos funciones para todos los objetos creados.

Evidentemente, en el `prototype` de un objeto sólo se deben añadir aquellos elementos comunes para todos los objetos. Normalmente se añaden los métodos y las constantes (propiedades cuyo valor no varía durante la ejecución de la aplicación). Las propiedades del objeto permanecen en la función constructora para que cada objeto diferente pueda tener un valor distinto en esas propiedades.

El mayor inconveniente de la propiedad `prototype` es que se pueden reescribir propiedades y métodos de forma accidental. Si se considera el siguiente ejemplo:

```
Factura.prototype.iva = 16;
var laFactura = new Factura(3, 7);   // laFactura.iva = 16

Factura.prototype.iva = 7;
```

```
var otraFactura = new Factura(5, 4);
// Ahora, laFactura.iva = otraFactura.iva = 7
```

El primer objeto creado de tipo `Factura` dispone de una propiedad llamada `iva` cuyo valor es `16`. Más adelante, se modifica el prototipo del objeto `Factura` durante la ejecución del programa y se establece un nuevo valor en la propiedad `iva`.

De esta forma, la propiedad `iva` del segundo objeto creado vale `7`. Además, el valor de la propiedad `iva` del primer objeto ha cambiado y ahora vale `7` y no `16`. Aunque la modificación del prototipo en tiempo de ejecución no suele ser una operación que se realice habitualmente, sí que es posible modificarlo de forma accidental.

Ejercicio 2

Modificar el ejercicio anterior del objeto Factura para crear una *pseudoclase* llamada `Factura` y que permita crear objetos de ese tipo. Se deben utilizar las funciones constructoras y la propiedad `prototype`.

Para instanciar la clase, se debe utilizar la instrucción `Factura(cliente, elementos)`, donde `cliente` también es una *pseudoclase* que guarda los datos del cliente y `elementos` es un array simple que contiene las *pseudoclases* de todos los elementos que forman la factura.

Una de las posibilidades más interesantes de la propiedad `prototype` es que también permite añadir y/o modificar las propiedades y métodos de los objetos predefinidos por JavaScript. Por lo tanto, es posible redefinir el comportamiento habitual de algunos métodos de los objetos nativos de JavaScript. Además, se pueden añadir propiedades o métodos completamente nuevos.

Si por ejemplo se considera la clase `Array`, esta no dispone de un método que indique la posición de un elemento dentro de un array (como la función `indexOf()` de Java). Modificando el prototipo con el que se construyen los objetos de tipo `Array`, es posible añadir esta funcionalidad:

```
Array.prototype.indexOf = function(objeto) {
  var resultado = -1;
  for(var i=0; i<this.length; i++) {
    if(this[i] == objeto) {
      resultado = i;
      break;
    }
  }
  return resultado;
}
```

El código anterior permite que todos los arrays de JavaScript dispongan de un método llamado `indexOf` que devuelve el índice de la primera posición de un elemento dentro del array o `-1` si el elemento no se encuentra en el array, tal y como sucede en otros lenguajes de programación.

El funcionamiento del método anterior consiste en recorrer el array actual (obteniendo su número de elementos mediante `this.length`) y comparar cada elemento del array con el elemento que se quiere encontrar. Cuando se encuentra por primera vez el elemento, se devuelve su posición dentro del array. En otro caso, se devuelve `-1`.

Como se verá más adelante, existen librerías de JavaScript formadas por un conjunto de utilidades que facilitan la programación de las aplicaciones. Una de las características habituales de estas librerías es el uso de la propiedad `prototype` para mejorar las funcionalidades básicas de JavaScript.

A continuación se muestra el ejemplo de una librería llamada *Prototype* que utiliza la propiedad `prototype` para ampliar las funcionalidades de los arrays de JavaScript:

```
Object.extend(Array.prototype, {
  _each: function(iterator) {
    for (var i = 0; i < this.length; i++)
      iterator(this[i]);
  },
  clear: function() {
    this.length = 0;
    return this;
  },
  first: function() {
    return this[0];
  },
  last: function() {
    return this[this.length - 1];
  },
  compact: function() {
    return this.select(function(value) {
      return value != undefined || value != null;
    });
  },
  flatten: function() {
    return this.inject([], function(array, value) {
      return array.concat(value.constructor == Array ?
        value.flatten() : [value]);
    });
  },
  without: function() {
    var values = $A(arguments);
    return this.select(function(value) {
      return !values.include(value);
    });
  },
  indexOf: function(object) {
    for (var i = 0; i < this.length; i++)
      if (this[i] == object) return i;
    return -1;
  },
  reverse: function(inline) {
    return (inline !== false ? this : this.toArray())._reverse();
  },
  shift: function() {
    var result = this[0];
    for (var i = 0; i < this.length - 1; i++)
      this[i] = this[i + 1];
    this.length--;
    return result;
```

```
  },
  inspect: function() {
    return '[' + this.map(Object.inspect).join(', ') + ']';
  }
});
```

El código anterior añade a la clase `Array` de JavaScript varias utilidades que no disponen los arrays por defecto: obtener (sin extraerlo) el primer o último elemento del array, filtrar sus valores, determinar la posición de un elemento, borrar el array, etc.

Ejercicio 3

Extender el objeto `Array` para que permita añadir nuevos elementos al final del array:

```
var array1 = [0, 1, 2];
array1.anadir(3);
// array1 = [0, 1, 2, 3]
```

Incluir la opción de controlar si se permiten elementos duplicados o no:

```
var array1 = [0, 1, 2];

array1.anadir(2);
// array1 = [0, 1, 2, 2]

array1.anadir(2, false);
// array1 = [0, 1, 2]
```

Cualquier clase nativa de JavaScript puede ser modificada mediante la propiedad `prototype`, incluso la clase `Object`. Se puede modificar por ejemplo la clase `String` para añadir un método que convierta una cadena en un array:

```
String.prototype.toArray = function() {
  return this.split('');
}
```

La función `split()` divide una cadena de texto según el separador indicado. Si no se indica ningún separador, se divide carácter a carácter. De este modo, la función anterior devuelve un array en el que cada elemento es una letra de la cadena de texto original.

Una función muy común en otros lenguajes de programación y que no dispone JavaScript es la función `trim()`, que elimina el espacio en blanco que pueda existir al principio y al final de una cadena de texto:

```
String.prototype.trim = function() {
  return this.replace(/^\s*|\s*$/g, '');
}

var cadena = "   prueba   de     cadena      ";
cadena.trim();
// Ahora cadena = "prueba   de     cadena"
```

La función `trim()` añadida al prototipo de la clase `String` hace uso de las expresiones regulares para detectar todos los espacios en blanco que puedan existir tanto al principio como al final de la cadena y se sustituyen por una cadena vacía, es decir, se eliminan.

Con este método, es posible definir las funciones asociadas `rtrim()` y `ltrim()` que eliminan los espacios en blanco a la derecha (final) de la cadena y a su izquierda (principio).

```
String.prototype.rtrim = function() {
  return this.replace(/\s*$/g, '');
}
String.prototype.ltrim = function() {
  return this.replace(/^\s*/g, '');
}
String.prototype.trim = function() {
  return this.ltrim().rtrim();
}
```

Otra función muy útil para las cadenas de texto es la de eliminar todas las etiquetas HTML que pueda contener. Cualquier aplicación en la que el usuario pueda introducir información, debe tener especial cuidado con los datos introducidos por el usuario. Mediante JavaScript se puede modificar la clase `String` para incluir una utilidad que elimine cualquier etiqueta de código HTML de la cadena de texto:

```
String.prototype.stripTags = function() {
  return this.replace(/<\/?[^>]+>/gi, '');
}

var cadena = '<html><head><meta content="text/html; charset=UTF-8"
http-equiv="content-type"></head><body><p>Parrafo de prueba</p></body></html>';
cadena.stripTags();
// Ahora cadena = "Parrafo de prueba"
```

El ejemplo anterior también hace uso de expresiones regulares complejas para eliminar cualquier trozo de texto que sea similar a una etiqueta HTML, por lo que se buscan patrones como `<...>` y `</...>`.

Ejercicio 4

Extender la clase `String` para que permita truncar una cadena de texto a un tamaño indicado como parámetro:

```
var cadena = "hola mundo";
cadena2 = cadena.truncar(6); // cadena2 = "hola m"
```

Modificar la función anterior para que permita definir el texto que indica que la cadena se ha truncado:

```
var cadena = "hola mundo";
cadena2 = cadena.truncar(6, '...'); // cadena2 = "hol..."
```

Ejercicio 5

Añadir a la clase Array un método llamado `sin()` que permita filtrar los elementos del array original y obtenga un nuevo array con todos los valores diferentes al indicado:

```
var array1 = [1, 2, 3, 4, 5];
var filtrado = array1.sin(4);  // filtrado = [1, 2, 3, 5]
```

3.2.3. Herencia y ámbito (scope)

Los lenguajes orientados a objetos disponen, entre otros, de los conceptos de herencia entre clases y de ámbito (*scope*) de sus métodos y propiedades (`public`, `private`, `protected`).

Sin embargo, JavaScript no dispone de forma nativa ni de herencia ni de ámbitos. Si se requieren ambos mecanismos, la única opción es simular su funcionamiento mediante clases, funciones y métodos desarrollados a medida.

Algunas técnicas simulan el uso de propiedades privadas prefijando su nombre con un guión bajo, para distinguirlas del resto de propiedades públicas. Para simular la herencia de clases, algunas librerías como Prototype añaden un método a la clase `Object` llamado `extend()` y que copia las propiedades de una clase origen en otra clase destino:

```
Object.extend = function(destination, source) {
  for (var property in source) {
    destination[property] = source[property];
  }
  return destination;
}
```

3.3. Otros conceptos

3.3.1. Excepciones

JavaScript dispone de un mecanismo de tratamiento de excepciones muy similar al de otros lenguajes de programación como Java. Para ello, define las palabras reservadas `try`, `catch` y `finally`.

La palabra reservada `try` se utiliza para encerrar el bloque de código JavaScript en el que se van a controlar las excepciones. Normalmente, el bloque definido por `try` va seguido de otro bloque de código definido por `catch`.

Cuando se produce una excepción en el bloque `try`, se ejecutan las instrucciones contenidas dentro del bloque `catch` asociado. Después del bloque `catch`, es posible definir un bloque con la palabra reservada `finally`. Todo el código contenido en el bloque `finally` se ejecuta independientemente de la excepción ocurrida en el bloque `try`.

Un bloque `try` debe ir seguido obligatoriamente de un bloque `catch` o de un bloque `finally`. También es posible que vaya seguido de los dos bloques.

A continuación se muestra un ejemplo de excepción y uso de los bloques `try` y `catch`:

```
try {
  var resultado = 5/a;
} catch(excepcion) {
  alert(excepcion);
}
```

El bloque `catch` permite indicar el nombre del parámetro que se crea automáticamente al producirse una excepción. Este identificador de la variable sólo está definido dentro del bloque `catch` y se puede utilizar para obtener más información sobre la excepción producida.

En este caso, al intentar dividir el número `5` por la variable `a` que no está definida, se produce una excepción que muestra el siguiente mensaje dentro del bloque `catch`:

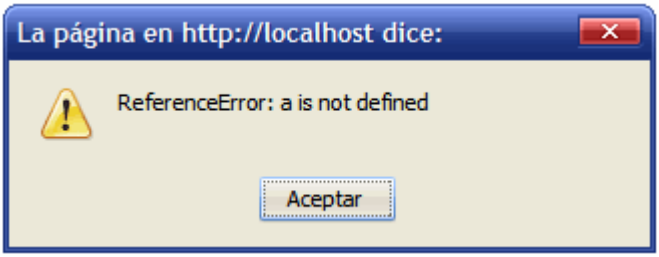

Figura 3.1. Excepción JavaScript provocada por una variable no definida

El funcionamiento del bloque `finally` no es tan sencillo como el bloque `catch`. Si se ejecuta cualquier parte del código que se encuentra dentro del bloque `try`, siempre se ejecuta el bloque `finally`, independientemente del resultado de la ejecución del bloque `try`.

Si dentro del bloque `try` se ejecuta una instrucción de tipo `return`, `continue` o `break`, también se ejecuta el bloque `finally` antes de ejecutar cualquiera de esas instrucciones. Si se produce una excepción en el bloque `try` y están definidos los bloques `catch` y `finally`, en primer lugar se ejecuta el bloque `catch` y a continuación el bloque `finally`.

JavaScript también permite lanzar excepciones manualmente mediante la palabra reservada `throw`:

```
try {
  if(typeof a == "undefined" || isNaN(a)) {
    throw new Error('La variable "a" no es un número');
  }
  var resultado = 5/a;
} catch(excepcion) {
  alert(excepcion);
} finally {
  alert("Se ejecuta");
}
```

En este caso, al ejecutar el script se muestran los dos siguientes mensajes de forma consecutiva:

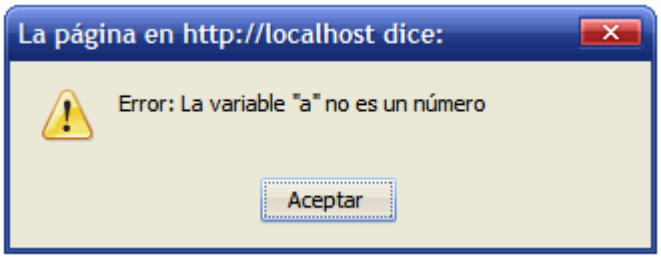

Figura 3.2. Manejando las excepciones en JavaScript para mostrar mensajes al usuario

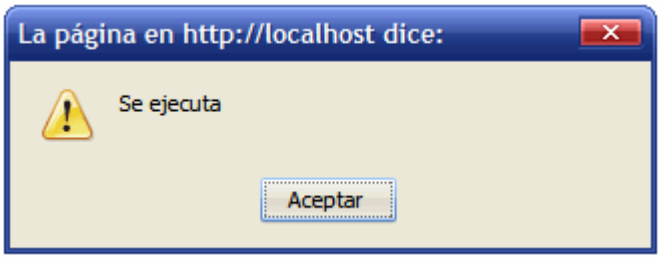

Figura 3.3. El bloque "finally" siempre se ejecuta cuando se produce una excepción en JavaScript

3.3.2. Clausura (closure)

El concepto de *clausura* (*closure* en inglés) es conocido y utilizado en varios lenguajes de programación desde hace décadas. JavaScript también permite utilizar *clausuras* en el código de las aplicaciones. En este caso, las *clausuras* se crean de forma implícita, ya que no existe ninguna forma de declararlas explícitamente.

A continuación se muestra un ejemplo sencillo de *clausura* en JavaScript:

```
var x = "estoy fuera";

function funcionExterna() {
  var x = "estoy dentro";
  function funcionAnidada() { alert(x); }
  funcionAnidada();
}

funcionExterna();
```

Al ejecutar el código anterior, se muestra el mensaje *"estoy dentro"*. Cuando se define una función dentro de otra, todas las variables de la primera función están disponibles de forma directa en la función anidada.

Técnicamente, una *clausura* es una porción de código (normalmente una función) que se evalúa en un entorno de ejecución que contiene variables de otro entorno de ejecución. JavaScript crea *clausuras* para todas las funciones definidas dentro de otra función. Sin embargo, el uso expreso de *clausuras* es muy limitado y se reserva sólo para las técnicas más avanzadas y complejas de JavaScript. Utilizando *clausuras* es posible por ejemplo simular el funcionamiento de las propiedades privadas en los objetos de JavaScript.

Existen recursos online con una explicación detallada del funcionamiento y aplicaciones de las *clausuras* en JavaScript, como por ejemplo http://www.jibbering.com/faq/faq_notes/closures.html

3.3.3. Reflexión

Al igual que la mayor parte de los lenguajes de programación más utilizados, JavaScript define mecanismos que permiten la *reflexión* sobre los objetos. La reflexión es un proceso mediante el cual un programa es capaz de obtener información sobre si mismo y por tanto es capaz de auto modificarse en tiempo de ejecución.

JavaScript emplea el concepto de reflexión para permitir descubrir propiedades y métodos de objetos externos. El ejemplo más sencillo es el de averiguar si un objeto posee un determinado método y así poder ejecutarlo. Si se dispone de un objeto llamado `elObjeto`, el código necesario para descubrir si posee una determinada propiedad llamada `laPropiedad` sería el siguiente:

```
if(elObjeto.laPropiedad) {
  // el objeto posee la propiedad buscada
}
```

Si el objeto no dispone de la propiedad buscada, la respuesta será `undefined`, que se transformará en un valor `false` que hace que no se ejecute el interior del bloque `if`.

Sin embargo, el código anterior no es del todo correcto, ya que si la propiedad buscada tiene un valor de `false`, `null` o el número `0`, el anterior código no se ejecutará correctamente. En tal caso, el código necesario es el siguiente:

```
if(typeof(elObjeto.laPropiedad) != 'undefined') {
  // el objeto posee la propiedad buscada
}
```

El ejemplo anterior hace uso del operador `typeof`, que devuelve el tipo del objeto o variable que se le pasa como parámetro. Los valores que devuelve este operador son: `undefined`, `number`, `object`, `boolean`, `string` o `function`.

El otro operador que ya se comentó anteriormente es `instanceof`, que comprueba si un objeto es una instancia de otro objeto:

```
if(elObjeto instanceof Factura) {
  alert("Se trata de un objeto de tipo Factura");
}
```

Este operador también se puede emplear con objetos nativos de JavaScript:

```
var elObjeto = [];
if(elObjeto instanceof Array) {
  alert("Es un array");
}
else if(elObjeto instanceof Object) {
  alert("Es un objeto");
}
```

En el ejemplo anterior, el programa muestra por pantalla el mensaje `"Es un array"` ya que se cumple la primera comprobación por ser la variable `elObjeto` un array. Sin embargo, si se cambia de orden las comprobaciones:

```
var elObjeto = [];
if(elObjeto instanceof Object) {
  alert("Es un objeto");
}
else if(elObjeto instanceof Array) {
  alert("Es un array");
}
```

En este caso, la salida del programa es el mensaje `"Es un objeto"`. El motivo es que, a pesar de que JavaScript no soporta el concepto de herencia en los objetos definidos a medida, los objetos nativos `Function` y `Array` sí que heredan del objeto `Object`. Así, la comprobación `elObjeto instanceof Object` devuelve un valor `true`, por ser `Array` una clase que hereda de `Object`.

Ejercicio 6

Sobrescribir el objeto `Object` para que incluya un método llamado `implementa()` y que indique si el objeto posee el método cuyo nombre se le pasa como parámetro.

Capítulo 4. DOM (Document Object Model)

4.1. Introducción a DOM

Cuando se definió el lenguaje XML, surgió la necesidad de procesar y manipular el contenido de los archivos XML mediante los lenguajes de programación tradicionales. XML es un lenguaje sencillo de escribir pero complejo para procesar y manipular de forma eficiente. Por este motivo, surgieron algunas técnicas entre las que se encuentra DOM.

DOM o *Document Object Model* es un conjunto de utilidades específicamente diseñadas para manipular documentos XML. Por extensión, DOM también se puede utilizar para manipular documentos XHTML y HTML. Técnicamente, DOM es una API de funciones que se pueden utilizar para manipular las páginas XHTML de forma rápida y eficiente.

Antes de poder utilizar sus funciones, DOM transforma internamente el archivo XML original en una estructura más fácil de manejar formada por una jerarquía de nodos. De esta forma, DOM transforma el código XML en una serie de nodos interconectados en forma de *árbol*.

El árbol generado no sólo representa los contenidos del archivo original (mediante los nodos del árbol) sino que también representa sus relaciones (mediante las ramas del árbol que conectan los nodos).

Aunque en ocasiones DOM se asocia con la programación web y con JavaScript, la API de DOM es independiente de cualquier lenguaje de programación. De hecho, DOM está disponible en la mayoría de lenguajes de programación comúnmente empleados.

Si se considera la siguiente página HTML 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=iso-8859-1" />
    <title>Página sencilla</title>
  </head>

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

Antes de poder utilizar las funciones de DOM, los navegadores convierten automáticamente la página HTML anterior en la siguiente estructura de árbol de nodos:

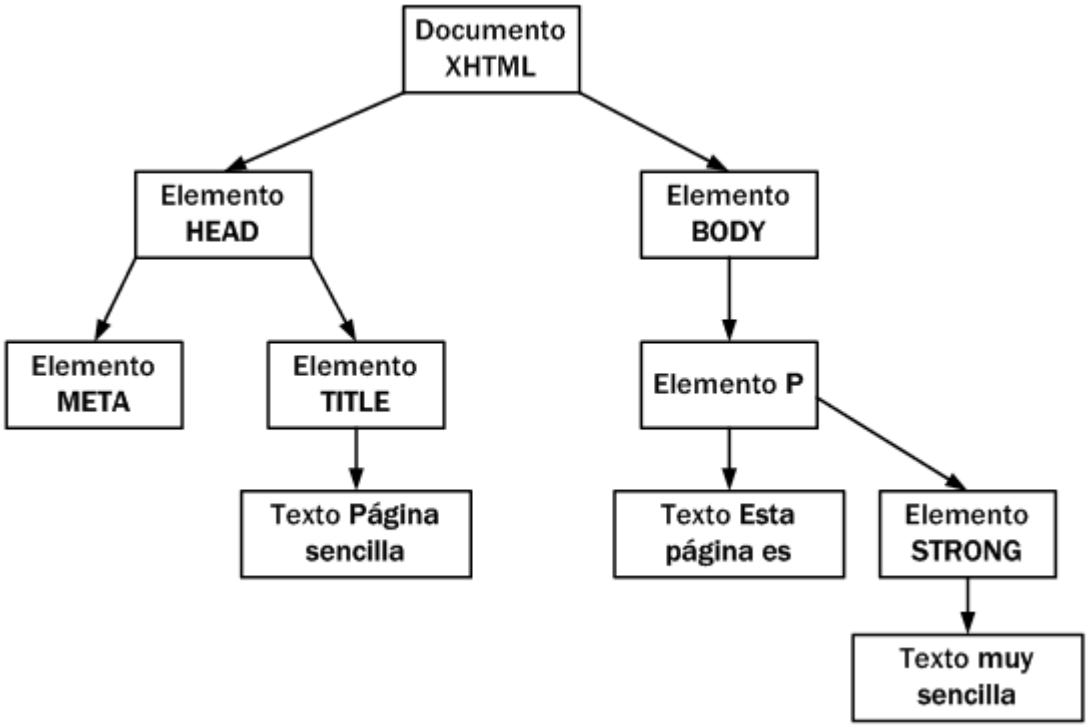

Figura 4.1. Representación en forma de árbol de la página HTML de ejemplo

La página HTML se ha transformado en una jerarquía de nodos, en la que el nodo raíz es un nodo de tipo *documento HTML*. A partir de este nodo, existen dos nodos en el mismo nivel formados por las etiquetas `<head>` y `<body>`. De cada uno de los anteriores surge otro nodo (`<title>` y `<p>` respectivamente). Por último, de cada nodo anterior surge otro nodo de tipo *texto*.

Antes de poder utilizar la API de DOM, se construye de forma automática el árbol para poder ejecutar de forma eficiente todas esas funciones. De este modo, para utilizar DOM es imprescindible que la página web se haya cargado por completo, ya que de otro modo no existe el árbol de nodos y las funciones DOM no pueden funcionar correctamente.

La ventaja de emplear DOM es que permite a los programadores disponer de un control muy preciso sobre la estructura del documento HTML o XML que están manipulando. Las funciones que proporciona DOM permiten añadir, eliminar, modificar y reemplazar cualquier nodo de cualquier documento de forma sencilla.

La primera especificación de DOM (*DOM Level 1*) se definió en 1998 y permitió homogeneizar la implementación del DHTML o *HTML dinámico* en los diferentes navegadores, ya que permitía modificar el contenido de las páginas web sin necesidad de recargar la página entera.

4.2. Tipos de nodos

Los documentos XML y HTML tratados por DOM se convierten en una jerarquía de nodos. Los nodos que representan los documentos pueden ser de diferentes tipos. A continuación se detallan los tipos más importantes:

- `Document`: es el nodo raíz de todos los documentos HTML y XML. Todos los demás nodos derivan de él.

- `DocumentType`: es el nodo que contiene la representación del DTD empleado en la página (indicado mediante el `DOCTYPE`).
- `Element`: representa el contenido definido por un par de etiquetas de apertura y cierre (`<etiqueta>...</etiqueta>`) o de una etiqueta *abreviada* que se abre y se cierra a la vez (`<etiqueta/>`). Es el único nodo que puede tener tanto nodos hijos como atributos.
- `Attr`: representa el par nombre-de-atributo/valor.
- `Text`: almacena el contenido del texto que se encuentra entre una etiqueta de apertura y una de cierre. También almacena el contenido de una sección de tipo CDATA.
- `CDataSection`: es el nodo que representa una sección de tipo `<![CDATA[ ]]>`.
- `Comment`: representa un comentario de XML.

Se han definido otros tipos de nodos pero que no son empleados habitualmente: `DocumentFragment`, `Entity`, `EntityReference`, `ProcessingInstruction` y `Notation`.

El siguiente ejemplo de documento sencillo de XML muestra algunos de los nodos más habituales:

```
<?xml version="1.0"?>
<clientes>
  <!-- El primer cliente -->
  <cliente>
    <nombre>Empresa SA</nombre>
    <sector>Tecnologia</sector>
    <notas><![CDATA[
    Llamar la proxima semana
    ]]></notas>
  </cliente>
</clientes>
```

Su representación como árbol de nodos DOM es la siguiente:

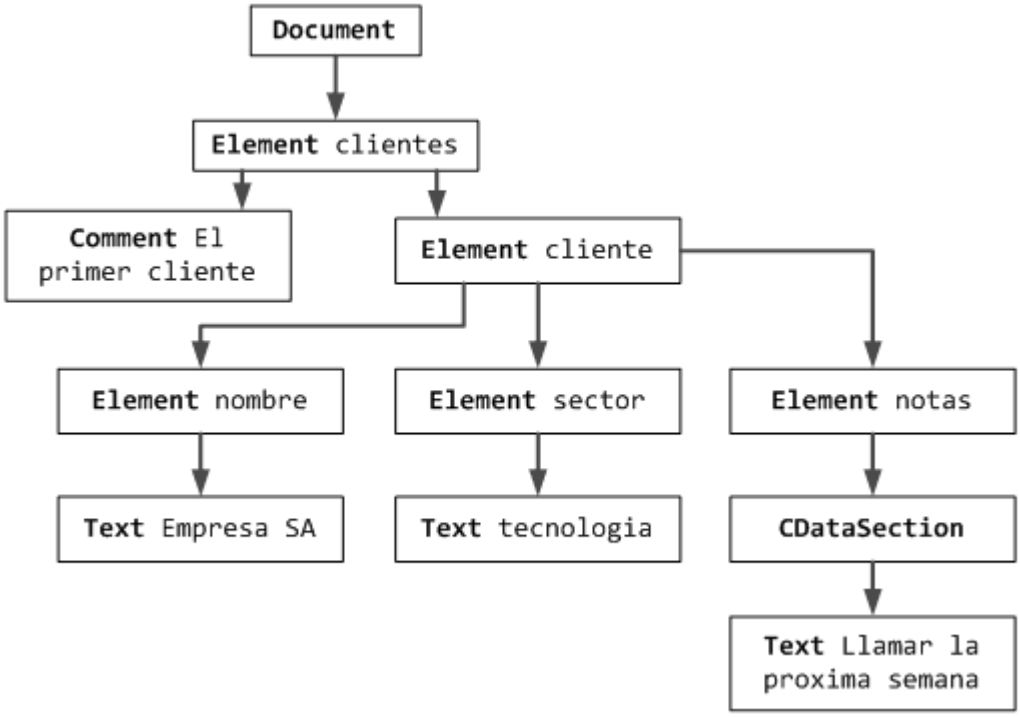

Figura 4.2. Representación en forma de árbol del archivo XML de ejemplo

El nodo raíz siempre es el nodo de tipo `Document`, del que derivan todos los demás nodos del documento. Este nodo es común para todas las páginas HTML y todos los documentos XML. A continuación se incluye la etiqueta `<clientes>...</clientes>`. Como se trata de una etiqueta, DOM la transforma en un nodo de tipo `Element`. Además, como la etiqueta encierra a todos los demás elementos de la página, el nodo `Clientes` de tipo `Element` deriva directamente de `Document` y todos los demás nodos del documento derivan de ese nodo.

El comentario es el primer texto que se incluye dentro de la etiqueta `<clientes>`, por lo que se transforma en el primer subnodo del nodo `clientes`. Al ser un comentario de XML, se trata de un nodo de tipo `Comment`.

Al mismo nivel que el comentario, se encuentra la etiqueta `<cliente>` que define las características del primer cliente y forma el segundo subnodo del nodo `clientes`. Todas las demás etiquetas del documento XML se encuentran encerradas por la etiqueta `<cliente>...</cliente>`, por lo que todos los nodos restantes derivarán del nodo `cliente`.

Cada etiqueta simple de tipo `<etiqueta>texto</etiqueta>` se transforma en un par de nodos: el primero de tipo `Element` (que contiene la etiqueta en sí) y el segundo, un nodo hijo de tipo `Text` que contiene el contenido definido entre la etiqueta de apertura y la de cierre.

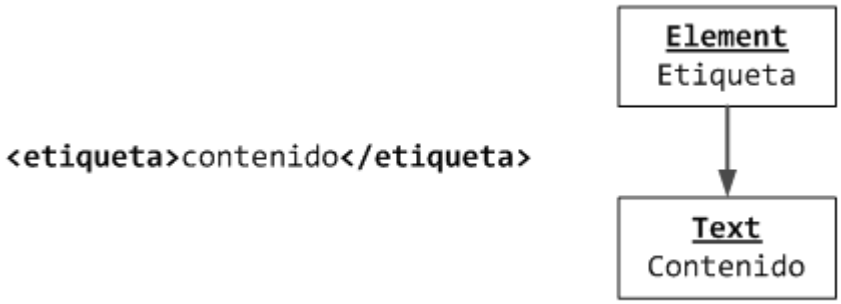

Figura 4.3. Nodos generados por una etiqueta HTML

La etiqueta `<notas>` se transforma en tres nodos, ya que contiene una sección de tipo `CData`, que a su vez se transforma en un nodo del que deriva el contenido propio de la sección `CData`.

Un buen método para comprobar la transformación que sufren las páginas web y visualizar la jerarquía de nodos creada por DOM es utilizar la utilidad *"Inspector DOM"* (o *"DOM Inspector"*) del navegador Mozilla Firefox.

La utilidad se puede encontrar en el menú *Herramientas* y además de mostrar visualmente la jerarquía de nodos, permite acceder fácilmente a toda la información de cada nodo y muestra en la página web el contenido al que hace referencia el nodo actual.

4.3. La interfaz Node

Una vez que DOM ha creado de forma automática el árbol completo de nodos de la página, ya es posible utilizar sus funciones para obtener información sobre los nodos o manipular su contenido. JavaScript crea el objeto `Node` para definir las propiedades y métodos necesarios para procesar y manipular los documentos.

En primer lugar, el objeto `Node` define las siguientes constantes para la identificación de los distintos tipos de nodos:

- `Node.ELEMENT_NODE = 1`
- `Node.ATTRIBUTE_NODE = 2`
- `Node.TEXT_NODE = 3`
- `Node.CDATA_SECTION_NODE = 4`
- `Node.ENTITY_REFERENCE_NODE = 5`
- `Node.ENTITY_NODE = 6`
- `Node.PROCESSING_INSTRUCTION_NODE = 7`
- `Node.COMMENT_NODE = 8`
- `Node.DOCUMENT_NODE = 9`
- `Node.DOCUMENT_TYPE_NODE = 10`
- `Node.DOCUMENT_FRAGMENT_NODE = 11`

- `Node.NOTATION_NODE = 12`

Además de estas constantes, `Node` proporciona las siguientes propiedades y métodos:

Propiedad/Método	Valor devuelto	Descripción
`nodeName`	`String`	El nombre del nodo (no está definido para algunos tipos de nodo)
`nodeValue`	`String`	El valor del nodo (no está definido para algunos tipos de nodo)
`nodeType`	`Number`	Una de las 12 constantes definidas anteriormente
`ownerDocument`	`Document`	Referencia del documento al que pertenece el nodo
`firstChild`	`Node`	Referencia del primer nodo de la lista `childNodes`
`lastChild`	`Node`	Referencia del último nodo de la lista `childNodes`
`childNodes`	`NodeList`	Lista de todos los nodos hijo del nodo actual
`previousSibling`	`Node`	Referencia del nodo hermano anterior o `null` si este nodo es el primer hermano
`nextSibling`	`Node`	Referencia del nodo hermano siguiente o `null` si este nodo es el último hermano
`hasChildNodes()`	`Bolean`	Devuelve `true` si el nodo actual tiene uno o más nodos hijo
`attributes`	`NamedNodeMap`	Se emplea con nodos de tipo `Element`. Contiene objetos de tipo `Attr` que definen todos los atributos del elemento
`appendChild(nodo)`	`Node`	Añade un nuevo nodo al final de la lista `childNodes`
`removeChild(nodo)`	`Node`	Elimina un nodo de la lista `childNodes`
`replaceChild(nuevoNodo, anteriorNodo)`	`Node`	Reemplaza el nodo `anteriorNodo` por el nodo `nuevoNodo`
`insertBefore(nuevoNodo, anteriorNodo)`	`Node`	Inserta el nodo `nuevoNodo` antes que la posición del nodo `anteriorNodo` dentro de la lista `childNodes`

Los métodos y propiedades incluidas en la tabla anterior son específicos de XML, aunque pueden aplicarse a todos los lenguajes basados en XML, como por ejemplo XHTML. Para las páginas creadas con HTML, los navegadores *hacen como si* HTML estuviera basado en XML y lo tratan de la misma forma. No obstante, se han definido algunas extensiones y particularidades específicas para XHTML y HTML.

4.4. HTML y DOM

Desafortunadamente, las posibilidades teóricas de DOM son mucho más avanzadas de las que se pueden utilizar en la práctica para desarrollar aplicaciones web. El motivo es que el uso de DOM siempre está limitado por las posibilidades que ofrece cada navegador. Mientras que algunos navegadores como Firefox y Safari implementan DOM de nivel 1 y 2 (y parte del 3), otros navegadores como Internet Explorer (versión 7 y anteriores) ni siquiera son capaces de ofrecer una implementación completa de DOM nivel 1.

Cuando se utiliza DOM en páginas HTML, el nodo raíz de todos los demás se define en el objeto `HTMLDocument`. Además, se crean objetos de tipo `HTMLElement` por cada nodo de tipo `Element` del árbol DOM. Como se verá en el siguiente capítulo, el objeto `document` es parte del BOM (*Browser Object Model*), aunque también se considera que es equivalente del objeto `Document` del DOM de los documentos XML. Por este motivo, el objeto `document` también hace referencia al nodo raíz de todas las páginas HTML.

4.4.1. Acceso relativo a los nodos

A continuación se muestra la página HTML básica que se va a emplear en todos los siguientes ejemplos:

```
<html>
<head>
  <title>Aprendiendo DOM</title>
</head>
<body>
  <p>Aprendiendo DOM</p>
  <p>DOM es sencillo de aprender</p>
  <p>Ademas, DOM es muy potente</p>
</body>
</html>
```

La operación básica consiste en obtener el objeto que representa el elemento raíz de la página:

```
var objeto_html = document.documentElement;
```

Después de ejecutar la instrucción anterior, la variable `objeto_html` contiene un objeto de tipo `HTMLElement` y que representa el elemento `<html>` de la página web. Según el árbol de nodos DOM, desde el nodo `<html>` derivan dos nodos del mismo nivel jerárquico: `<head>` y `<body>`.

Utilizando los métodos proporcionados por DOM, es sencillo obtener los elementos `<head>` y `<body>`. En primer lugar, los dos nodos se pueden obtener como el primer y el último nodo hijo del elemento `<html>`:

```
var objeto_head = objeto_html.firstChild;
var objeto_body = objeto_html.lastChild;
```

Otra forma directa de obtener los dos nodos consiste en utilizar la propiedad `childNodes` del elemento `<html>`:

```
var objeto_head = objeto_html.childNodes[0];
var objeto_body = objeto_html.childNodes[1];
```

Si se desconoce el número de nodos hijo que dispone un nodo, se puede emplear la propiedad `length` de `childNodes`:

```
var numeroDescendientes = objeto_html.childNodes.length;
```

Además, el DOM de HTML permite acceder directamente al elemento `<body>` utilizando el atajo `document.body`:

```
var objeto_body = document.body;
```

Además de las propiedades anteriores, existen otras propiedades como `previousSibling` y `parentNode` que se pueden utilizar para acceder a un nodo a partir de otro. Utilizando estas propiedades, se pueden comprobar las siguientes igualdades:

```
objeto_head.parentNode == objeto_html
objeto_body.parentNode == objeto_html
objeto_body.previousSibling == objeto_head
objeto_head.nextSibling == objeto_body
objeto_head.ownerDocument == document
```

4.4.2. Tipos de nodos

Una operación común en muchas aplicaciones consiste en comprobar el tipo de nodo, que se obtiene de forma directa mediante la propiedad `nodeType`:

```
alert(document.nodeType); // 9
alert(document.documentElement.nodeType); // 1
```

En el primer caso, el valor `9` es igual al definido en la constante `Node.DOCUMENT_NODE`. En el segundo ejemplo, el valor `1` coincide con la constante `Node.ELEMENT_NODE`. Afortunadamente, no es necesario memorizar los valores numéricos de los tipos de nodos, ya que se pueden emplear las constantes predefinidas:

```
alert(document.nodeType == Node.DOCUMENT_NODE); // true
alert(document.documentElement.nodeType == Node.ELEMENT_NODE); // true
```

El único navegador que no soporta las constantes predefinidas es Internet Explorer 7 y sus versiones anteriores, por lo que si se quieren utilizar es necesario definirlas de forma explícita:

```
if(typeof Node == "undefined") {
  var Node = {
    ELEMENT_NODE: 1,
    ATTRIBUTE_NODE: 2,
    TEXT_NODE: 3,
    CDATA_SECTION_NODE: 4,
    ENTITY_REFERENCE_NODE: 5,
    ENTITY_NODE: 6,
    PROCESSING_INSTRUCTION_NODE: 7,
    COMMENT_NODE: 8,
    DOCUMENT_NODE: 9,
    DOCUMENT_TYPE_NODE: 10,
    DOCUMENT_FRAGMENT_NODE: 11,
    NOTATION_NODE: 12
  };
}
```

El código anterior comprueba si el navegador en el que se está ejecutando tiene definido el objeto `Node`. Si este objeto no está definido, se trata del navegador Internet Explorer 7 o alguna versión anterior, por lo que se crea un nuevo objeto llamado `Node` y se le incluyen como propiedades todas las constantes definidas por DOM.

4.4.3. Atributos

Además del tipo de etiqueta HTML y su contenido de texto, DOM permite el acceso directo a todos los atributos de cada etiqueta. Para ello, los nodos de tipo `Element` contienen la propiedad `attributes`, que permite acceder a todos los atributos de cada elemento. Aunque técnicamente la propiedad `attributes` es de tipo `NamedNodeMap`, sus elementos se pueden acceder como si fuera un array. DOM proporciona diversos métodos para tratar con los atributos:

- `getNamedItem(nombre)`, devuelve el nodo cuya propiedad `nodeName` contenga el valor `nombre`.
- `removeNamedItem(nombre)`, elimina el nodo cuya propiedad `nodeName` coincida con el valor `nombre`.
- `setNamedItem(nodo)`, añade el nodo a la lista `attributes`, indexándolo según su propiedad `nodeName`.
- `item(posicion)`, devuelve el nodo que se encuentra en la posición indicada por el valor numérico `posicion`.

Los métodos anteriores devuelven un nodo de tipo `Attr`, por lo que no devuelven directamente el valor del atributo. Utilizando estos métodos, es posible procesar y modificar fácilmente los atributos de los elementos HTML:

```
<p id="introduccion" style="color: blue">Párrafo de prueba</p>

var p = document.getElementById("introduccion");
var elId = p.attributes.getNamedItem("id").nodeValue; // elId = "introduccion"
var elId = p.attributes.item(0).nodeValue;            // elId = "introduccion"
p.attributes.getNamedItem("id").nodeValue = "preintroduccion";

var atributo = document.createAttribute("lang");
atributo.nodeValue = "es";
p.attributes.setNamedItem(atributo);
```

Afortunadamente, DOM proporciona otros métodos que permiten el acceso y la modificación de los atributos de forma más directa:

- `getAttribute(nombre)`, es equivalente a `attributes.getNamedItem(nombre)`.
- `setAttribute(nombre, valor)` equivalente a `attributes.getNamedItem(nombre).value = valor`.
- `removeAttribute(nombre)`, equivalente a `attributes.removeNamedItem(nombre)`.

De esta forma, el ejemplo anterior se puede reescribir utilizando los nuevos métodos:

```
<p id="introduccion" style="color: blue">Párrafo de prueba</p>

var p = document.getElementById("introduccion");
var elId = p.getAttribute("id"); // elId = "introduccion"
p.setAttribute("id", "preintroduccion");
```

4.4.4. Acceso directo a los nodos

Los métodos presentados hasta el momento permiten acceder a cualquier nodo del árbol de nodos DOM y a todos sus atributos. Sin embargo, las funciones que proporciona DOM para acceder a un nodo a través de sus padres obligan a 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.

Cuando se trabaja con una página web real, el árbol DOM tiene miles de nodos de todos los tipos. Por este motivo, no es eficiente acceder a un nodo descendiendo a través de todos los ascendentes de ese nodo.

Para solucionar este problema, DOM proporciona una serie de métodos para acceder de forma directa a los nodos deseados. Los métodos disponibles son `getElementsByTagName()`, `getElementsByName()` y `getElementById()`.

4.4.4.1. Función getElementsByTagName()

La función `getElementsByTagName()` 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 devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. En realidad, el valor devuelto no es de tipo array *normal*, sino que es un objeto de tipo `NodeList`. De este modo, el primer párrafo de la página se puede obtener de la siguiente manera:

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

De la misma forma, se pueden recorrer todos los párrafos de la página recorriendo el array de nodos devuelto por la función:

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

4.4.4.2. Función getElementsByName()

La función `getElementsByName()` obtiene todos los elementos de la página XHTML cuyo atributo `name` coincida con el parámetro que se le pasa a la función.

En el siguiente ejemplo, se obtiene directamente el único párrafo de la página que tiene 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 incluyen, 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.

Internet Explorer 7 y sus versiones anteriores no implementan de forma correcta esta función, ya que también devuelven los elementos cuyo atributo `id` sea igual al parámetro de la función.

4.4.4.3. Función getElementById()

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

Ejercicio 7

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

Ejercicio 8

A partir de la página web proporcionada y utilizando las funciones DOM:

1. Se debe modificar el protocolo de todas las direcciones de los enlaces. De esta forma, si un enlace apuntaba a http://prueba, ahora debe apuntar a https://prueba
2. Los párrafos de la página cuyo atributo `class` es igual a `"importante"` deben modificarlo por `"resaltado"`. El resto de párrafos deben incluir un atributo `class` igual a `"normal"`.

3. A los enlaces de la página cuyo atributo `class` sea igual a `"importante"`, se les añade un atributo `"name"` con un valor generado automáticamente y que sea igual a `"importante"+i`, donde `i` es un valor numérico cuyo valor inicial es `0` para el primer enlace.

4.4.5. Crear, modificar y eliminar nodos

Hasta ahora, todos los métodos DOM presentados se limitan a acceder a los nodos y sus propiedades. A continuación, se muestran los métodos necesarios para la creación, modificación y eliminación de nodos dentro del árbol de nodos DOM.

Los métodos DOM disponibles para la creación de nuevos nodos son los siguientes:

Método	**Descripción**
`createAttribute(nombre)`	Crea un nodo de tipo atributo con el nombre indicado
`createCDataSection(texto)`	Crea una sección `CDATA` con un nodo hijo de tipo texto que contiene el valor indicado
`createComment(texto)`	Crea un nodo de tipo comentario que contiene el valor indicado
`createDocumentFragment()`	Crea un nodo de tipo `DocumentFragment`
`createElement(nombre_etiqueta)`	Crea un elemento del tipo indicado en el parámetro `nombre_etiqueta`
`createEntityReference(nombre)`	Crea un nodo de tipo `EntityReference`
`createProcessingInstruction(objetivo, datos)`	Crea un nodo de tipo `ProcessingInstruction`
`createTextNode(texto)`	Crea un nodo de tipo texto con el valor indicado como parámetro

Internet Explorer no soporta los métodos `createCDataSection`, `createEntityReference` y `createProcessingInstruction`. Los métodos más empleados son `createElement`, `createTextNode` y `createAttribute`.

A continuación se muestra un ejemplo sencillo en el que se crea un nuevo nodo (una nueva etiqueta HTML) y se añade dinámicamente a la siguiente página HTML:

```
<html>
  <head><title>Ejemplo de creación de nodos</title></head>
  <body></body>
</html>
```

La página HTML original no tiene contenidos, pero mediante DOM se añade dinámicamente el siguiente párrafo de texto:

```
<p>Este párrafo no existía en la página HTML original</p>
```

Añadir el párrafo anterior a la página requiere los siguientes pasos:

1. Crear un nodo de tipo elemento
2. Crear un nodo de tipo texto

3. Asociar el nodo de texto al elemento
4. Añadir el nuevo nodo de tipo elemento a la página original

En primer lugar, se crea un nuevo nodo de tipo elemento:

```
var p = document.createElement("p");
```

A continuación se crea un nodo de texto que almacena el contenido de texto del párrafo:

```
var texto = document.createTextNode("Este párrafo no existía en la página HTML
original");
```

En tercer lugar, se asocia el elemento creado y su contenido de texto (los nodos de tipo `Text` son hijos de los nodos de tipo `Element`):

```
p.appendChild(texto);
```

El método `appendChild()` está definido para todos los diferentes tipos de nodos y se encarga de añadir un nodo al final de la lista `childNodes` de otro nodo.

Por último, se añade el nodo creado al árbol de nodos DOM que representa a la página. Utilizando el método `appendChild()`, se añade el nodo como hijo del nodo que representa al elemento `<body>` de la página:

```
document.body.appendChild(p);
```

La página HTML que resulta después de la ejecución del código JavaScript se muestra a continuación:

```
<html>
  <head><title>Ejemplo de creación de nodos</title></head>
  <body>
    <p>Este párrafo no existía en la página HTML original</p>
  </body>
</html>
```

Una vez más, es importante recordar que las modificaciones en el árbol de nodos DOM sólo se pueden realizar cuando **toda** la página web se ha cargado en el navegador. El motivo es que los navegadores construyen el árbol de nodos DOM una vez que se ha cargado completamente la página web. Cuando una página no ha terminado de cargarse, su árbol no está construido y por tanto no se pueden utilizar las funciones DOM.

Si una página realiza modificaciones automáticas (sin intervención del usuario) es importante utilizar el evento `onload()` para llamar a las funciones de JavaScript, tal y como se verá más adelante en el capítulo de los eventos.

Por otra parte, también se pueden utilizar funciones DOM para eliminar cualquier nodo existente originalmente en la página y cualquier nodo creado mediante los métodos DOM:

Si se considera la siguiente página HTML con un párrafo de texto:

```
<html>
  <head><title>Ejemplo de eliminación de nodos</title></head>
  <body>
    <p>Este parrafo va a ser eliminado dinamicamente</p>
```

```
  </body>
</html>
```

En primer lugar, se obtiene la referencia al nodo que se va a eliminar:

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

Para eliminar cualquier nodo, se emplea la función `removeChild()`, que toma como argumento la referencia al nodo que se quiere eliminar. La función `removeChild()` se debe invocar sobre el nodo padre del nodo que se va a eliminar. En este caso, el *padre* del primer párrafo de la página es el nodo `<body>`:

```
var p = document.getElementsByTagName("p")[0];
document.body.removeChild(p);
```

La página HTML que resulta después de la ejecución del código JavaScript anterior se muestra a continuación:

```
<html>
  <head><title>Ejemplo de eliminación de nodos</title></head>
  <body></body>
</html>
```

Cuando la página está formada por miles de nodos, puede ser costoso acceder hasta el nodo padre del nodo que se quiere eliminar. En estos casos, se puede utilizar la propiedad `parentNode`, que siempre hace referencia al nodo padre de un nodo.

De esta forma, el ejemplo anterior se puede rehacer para eliminar el nodo haciendo uso de la propiedad `parentNode`:

```
var p = document.getElementsByTagName("p")[0];
p.parentNode.removeChild(p);
```

Además de crear y eliminar nodos, las funciones DOM también permiten reemplazar un nodo por otro. Utilizando la misma página HTML de ejemplo, se va a sustituir el párrafo original por otro párrafo con un contenido diferente. La página original contiene el siguiente párrafo:

```
<html>
  <head><title>Ejemplo de sustitución de nodos</title></head>
  <body>
    <p>Este parrafo va a ser sustituido dinamicamente</p>
  </body>
</html>
```

Utilizando las funciones DOM de JavaScript, se crea el nuevo párrafo que se va a mostrar en la página, se obtiene la referencia al nodo original y se emplea la función `replaceChild()` para intercambiar un nodo por otro:

```
var nuevoP = document.createElement("p");
var texto = document.createTextNode("Este parrafo se ha creado dinámicamente y
sustituye al parrafo original");
nuevoP.appendChild(texto);

var anteriorP = document.body.getElementsByTagName("p")[0];
anteriorP.parentNode.replaceChild(nuevoP, anteriorP);
```

Después de ejecutar las instrucciones anteriores, la página HTML resultante es:

```
<html>
  <head><title>Ejemplo de sustitución de nodos</title></head>
  <body>
    <p>Este parrafo se ha creado dinámicamente y sustituye al parrafo original</p>
  </body>
</html>
```

Además de crear, eliminar y sustituir nodos, las funciones DOM permiten insertar nuevos nodos antes o después de otros nodos ya existentes. Si se quiere insertar un nodo después de otro, se emplea la función `appendChild()`. Página HTML original:

```
<html>
  <head><title>Ejemplo de inserción de nodos</title></head>
  <body>
    <p>Primer parrafo</p>
  </body>
</html>
```

El siguiente código JavaScript crea un nuevo párrafo y lo inserta después de cualquier otro nodo de la página HTML:

```
var nuevoP = document.createElement("p");
var texto = document.createTextNode("Segundo parrafo");
nuevoP.appendChild(texto);

document.body.appendChild(nuevoP);
```

Después de ejecutar el código JavaScript anterior, el código HTML resultante es:

```
<html>
  <head><title>Ejemplo de inserción de nodos</title></head>
  <body>
    <p>Primer parrafo</p>
    <p>Segundo parrafo</p>
  </body>
</html>
```

Si se quiere insetar el nuevo párrafo delante del párrafo existente, se puede utilizar la función `insertBefore()`. Página HTML original:

```
<html>
  <head><title>Ejemplo de inserción de nodos</title></head>
  <body>
    <p>Primer parrafo</p>
  </body>
</html>
```

Código JavaScript necesario para insertar un nuevo párrafo delante del párrafo existente:

```
var nuevoP = document.createElement("p");
var texto = document.createTextNode("Segundo parrafo, antes del primero");
nuevoP.appendChild(texto);

var anteriorP = document.getElementsByTagName("p")[0];
document.body.insertBefore(nuevoP, anteriorP);
```

Después de ejecutar el código JavaScript anterior, el código HTML resultante es:

```
<html>
  <head><title>Ejemplo de inserción de nodos</title></head>
  <body>
    <p>Segundo parrafo, antes del primero</p>
    <p>Primer parrafo</p>
  </body>
</html>
```

Ejercicio 9

A partir de la página HTML que se proporciona, completar el código JavaScript definido para realizar la siguiente aplicación sencilla:

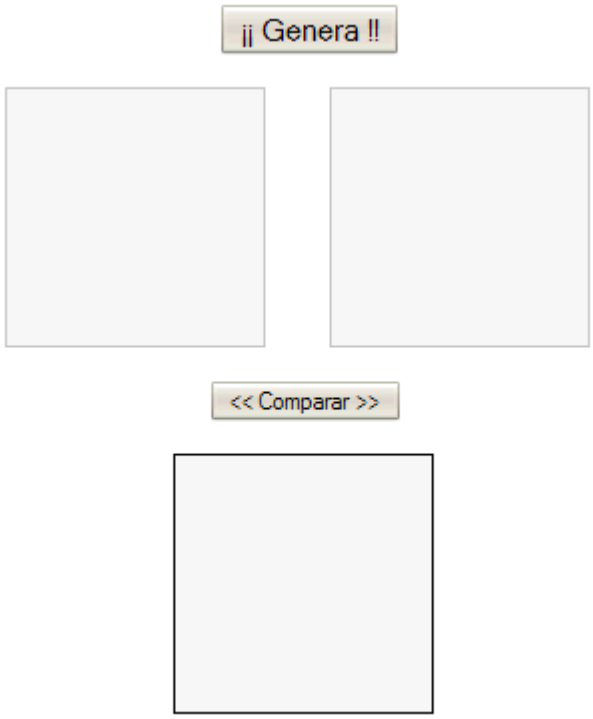

Figura 4.4. Inicio de la aplicación

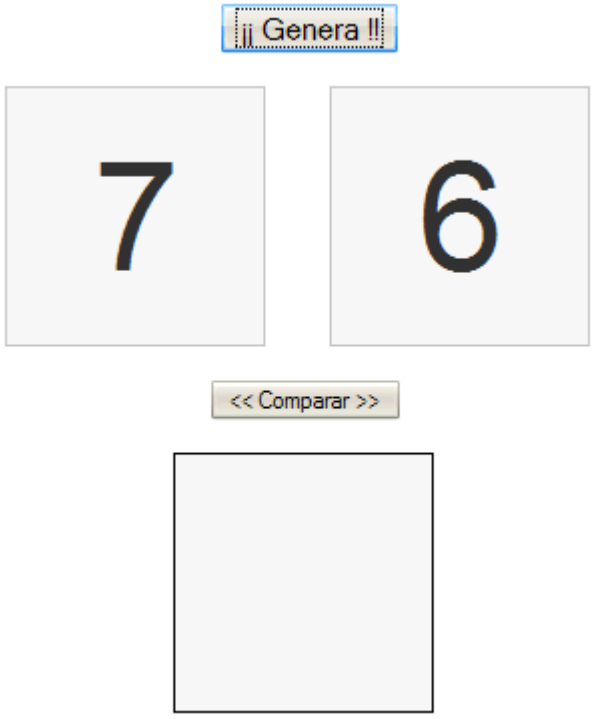

Figura 4.5. Generación de números aleatorios

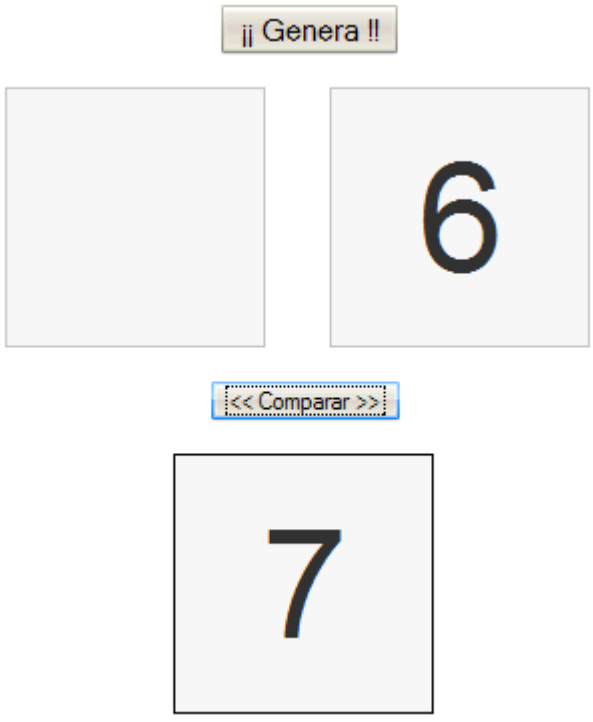

Figura 4.6. Resultado de la comparación

Inicialmente, la aplicación cuenta con tres cajas vacías y dos botones. Al presionar el botón de "Genera", se crean dos números aleatorios. Cada número aleatorio se guarda en un elemento <p>, que a su vez se guarda en una de las dos cajas superiores.

Una vez generados los números, se presiona el botón "Comparar", que compara el valor de los dos párrafos anteriores y determina cual es el mayor. El párrafo con el número más grande, se mueve a la última caja que se utiliza para almacenar el resultado de la operación.

La página que se proporciona contiene todo el código HTML y CSS necesario. Además, incluye todo el código JavaScript relativo a la pulsación de los botones y que se estudiará con detalle en el siguiente capítulo.

En el ejercicio se deben utilizar entre otras, las funciones `getElementById()`, `createElement()`, `createTextNode()`, `appendChild()` y `replaceChild()`.

4.4.6. Atributos HTML y propiedades CSS en DOM

Los métodos presentados anteriormente para el acceso a los atributos de los elementos, son genéricos de XML. La versión de DOM específica para HTML incluye algunas propiedades y métodos aún más directos y sencillos para el acceso a los atributos de los elementos HTML y a sus propiedades CSS.

La principal ventaja del DOM para HTML es que todos los atributos de todos los elementos HTML se transforman en propiedades de los nodos. De esta forma, es posible acceder de forma directa a cualquier atributo de HTML. Si se considera el siguiente elemento `<img>` de HTML con sus tres atributos:

```
<img id="logo" src="logo.gif" border="0" />
```

Empleando los métodos tradicionales de DOM, se puede acceder y manipular cada atributo:

```
var laImagen = document.getElementById("logo");

// acceder a los atributos
var archivo = laImagen.getAttribute("src");
var borde = laImagen.getAttribute("border");

// modificar los atributos
laImagen.setAttribute("src", "nuevo_logo.gif");
laImagen.setAttribute("border", "1");
```

La ventaja de la especificación de DOM para HTML es que permite acceder y modificar todos los atributos de los elementos de forma directa:

```
var laImagen = document.getElementById("logo");

// acceder a los atributos
var archivo = laImagen.src;
var borde = laImagen.border;

// modificar los atributos
laImagen.src = "nuevo_logo.gif";
laImagen.border = "1";
```

Las ventajas de utilizar esta forma de acceder y modificar los atributos de los elementos es que el código resultante es más sencillo y conciso. Por otra parte, algunas versiones de Internet Explorer no implementan correctamente el método `setAttribute()`, lo que provoca que, en ocasiones, los cambios realizados no se reflejan en la página HTML.

La única excepción que existe en esta forma de obtener el valor de los atributos HTML es el atributo `class`. Como la palabra `class` está reservada por JavaScript para su uso futuro, no es

posible utilizarla para acceder al valor del atributo `class` de HTML. La solución consiste en acceder a ese atributo mediante el nombre alternativo `className`:

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

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

El acceso a las propiedades CSS no es tan directo y sencillo como el acceso a los atributos HTML. En primer lugar, los estilos CSS se pueden aplicar de varias formas diferentes sobre un mismo elemento HTML. Si se establecen las propiedades CSS mediante el atributo `style` de HTML:

```
<p id="parrafo" style="color: #C00">...</p>
```

El acceso a las propiedades CSS establecidas mediante el atributo `style` se realiza a través de la propiedad `style` del nodo que representa a ese elemento:

```
var parrafo = document.getElementById("parrafo");
var color = parrafo.style.color;
```

Para acceder al valor de una propiedad CSS, se obtiene la referencia del nodo, se accede a su propiedad `style` y a continuación se indica el nombre de la propiedad CSS cuyo valor se quiere obtener. Aunque parece lógico que en el ejemplo anterior el valor obtenido sea `#C00`, en realidad cada navegador obtiene el mismo valor de formas diferentes:

- Firefox y Safari muestran el valor `rgb(204, 0, 0)`
- Internet Explorer muestra el valor `#c00`
- Opera muestra el valor `#cc0000`

En el caso de las propiedades CSS con nombre compuesto (`font-weight`, `border-top-style`, `list-style-type`, etc.), para acceder a su valor es necesario modificar su nombre original eliminando los guiones medios y escribiendo en mayúsculas la primera letra de cada palabra que no sea la primera:

```
var parrafo = document.getElementById("parrafo");
var negrita = parrafo.style.fontWeight;
```

El nombre original de la propiedad CSS es `font-weight`. Para obtener su valor mediante JavaScript, se elimina el guión medio (`fontweight`) y se pasa a mayúsculas la primera letra de cada palabra que no sea la primera (`fontWeight`).

Si el nombre de la propiedad CSS está formado por tres palabras, se realiza la misma transformación. De esta forma, la propiedad `border-top-style` se accede en DOM mediante el nombre `borderTopStyle`.

Además de obtener el valor de las propiedades CSS, también es posible modificar su valor mediante JavaScript. Una vez obtenida la referencia del nodo, se puede modificar el valor de cualquier propiedad CSS accediendo a ella mediante la propiedad `style`:

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

```
var parrafo = document.getElementById("parrafo");
parrafo.style.margin = "10px";   // añade un margen de 10px al párrafo
parrafo.style.color = "#CCC";    // modifica el color de la letra del párrafo
```

Todos los ejemplos anteriores hacen uso de la propiedad `style` para acceder o establecer el valor de las propiedades CSS de los elementos. Sin embargo, esta propiedad sólo permite acceder al valor de las propiedades CSS establecidas directamente sobre el elemento HTML. En otras palabras, la propiedad `style` del nodo sólo contiene el valor de las propiedades CSS establecidas mediante el atributo `style` de HTML.

Por otra parte, los estilos CSS normalmente se aplican mediante reglas CSS incluidas en archivos externos. Si se utiliza la propiedad `style` de DOM para acceder al valor de una propiedad CSS establecida mediante una regla externa, el navegador no obtiene el valor correcto:

```
// Código HTML
<p id="parrafo">...</p>

// Regla CSS
#parrafo { color: #008000; }

// Código JavaScript
var parrafo = document.getElementById("parrafo");
var color = parrafo.style.color;  // color no almacena ningún valor
```

Para obtener el valor de las propiedades CSS independientemente de cómo se hayan aplicado, es necesario utilizar otras propiedades de JavaScript. Si se utiliza un navegador de la familia Internet Explorer, se hace uso de la propiedad `currentStyle`. Si se utiliza cualquier otro navegador, se puede emplear la función `getComputedStyle()`.

```
// Código HTML
<p id="parrafo">...</p>

// Regla CSS
#parrafo { color: #008000; }

// Código JavaScript para Internet Explorer
var parrafo = document.getElementById("parrafo");
var color = parrafo.currentStyle['color'];

// Código JavaScript para otros navegadores
var parrafo = document.getElementById("parrafo");
var color = document.defaultView.getComputedStyle(parrafo,
'').getPropertyValue('color');
```

La propiedad `currentStyle` requiere el nombre de las propiedades CSS según el formato de JavaScript (sin guiones medios), mientras que la función `getPropertyValue()` exige el uso del nombre original de la propiedad CSS. Por este motivo, la creación de aplicaciones compatibles con todos los navegadores se puede complicar en exceso.

A continuación se muestra una función compatible con todos los navegadores, creada por el programador Robert Nyman y publicada en su blog personal (http://www.robertnyman.com/2006/04/24/get-the-rendered-style-of-an-element/) :

```
function getStyle(elemento, propiedadCss) {
  var valor = "";
  if(document.defaultView && document.defaultView.getComputedStyle){
    valor = document.defaultView.getComputedStyle(elemento,
'').getPropertyValue(propiedadCss);
  }
  else if(elemento.currentStyle) {
    propiedadCss = propiedadCss.replace(/\-(\w)/g, function (strMatch, p1) {
      return p1.toUpperCase();
    });
    valor = elemento.currentStyle[propiedadCss];
  }
  return valor;
}
```

Utilizando la función anterior, es posible rehacer el ejemplo para que funcione correctamente en cualquier navegador:

```
// Código HTML
<p id="parrafo">...</p>

// Regla CSS
#parrafo { color: #008000; }

// Código JavaScript para cualquier navegador
var parrafo = document.getElementById("parrafo");
var color = getStyle(parrafo, 'color');
```

4.4.7. Tablas HTML en DOM

Las tablas son elementos muy comunes en las páginas HTML, por lo que DOM proporciona métodos específicos para trabajar con ellas. Si se utilizan los métodos tradicionales, crear una tabla es una tarea tediosa, por la gran cantidad de nodos de tipo elemento y de tipo texto que se deben crear.

Afortunadamente, la versión de DOM para HTML incluye varias propiedades y métodos para crear tablas, filas y columnas de forma sencilla.

Propiedades y métodos de `<table>`:

Propiedad/Método	**Descripción**
`rows`	Devuelve un array con las filas de la tabla
`tBodies`	Devuelve un array con todos los `<tbody>` de la tabla
`insertRow(posicion)`	Inserta una nueva fila en la posición indicada dentro del array de filas de la tabla
`deleteRow(posicion)`	Elimina la fila de la posición indicada

Propiedades y métodos de `<tbody>`:

Propiedad/Método	**Descripción**
`rows`	Devuelve un array con las filas del `<tbody>` seleccionado

`insertRow(posicion)`	Inserta una nueva fila en la posición indicada dentro del array de filas del `<tbody>`
`deleteRow(posicion)`	Elimina la fila de la posición indicada

Propiedades y métodos de `<tr>`:

Propiedad/Método	**Descripción**
`cells`	Devuelve un array con las columnas de la fila seleccionada
`insertCell(posicion)`	Inserta una nueva columna en la posición indicada dentro del array de columnas de la fila
`deleteCell(posicion)`	Elimina la columna de la posición indicada

Si se considera la siguiente tabla XHTML:

```
<table summary="Descripción de la tabla y su contenido">
<caption>Título de la tabla</caption>
<thead>
  <tr>
    <th scope="col"></th>
    <th scope="col">Cabecera columna 1</th>
    <th scope="col">Cabecera columna 2</th>
  </tr>
</thead>

<tfoot>
  <tr>
    <th scope="col"></th>
    <th scope="col">Cabecera columna 1</th>
    <th scope="col">Cabecera columna 2</th>
  </tr>
</tfoot>

<tbody>
  <tr>
    <th scope="row">Cabecera fila 1</th>
    <td>Celda 1 - 1</td>
    <td>Celda 1 - 2</td>
  </tr>
  <tr>
    <th scope="row">Cabecera fila 2</th>
    <td>Celda 2 - 1</td>
    <td>Celda 2 - 2</td>
  </tr>
</tbody>

</table>
```

A continuación se muestran algunos ejemplos de manipulación de tablas XHTML mediante las propiedades y funciones específicas de DOM.

Obtener el número total de filas de la tabla:

```
var tabla = document.getElementById('miTabla');
var numFilas = tabla.rows.length;
```

Obtener el número total de cuerpos de la tabla (secciones <tbody>):

```
var tabla = document.getElementById('miTabla');
var numCuerpos = tabla.tBodies.length;
```

Obtener el número de filas del primer cuerpo (sección <tbody>) de la tabla:

```
var tabla = document.getElementById('miTabla');
var numFilasCuerpo = tabla.tBodies[0].rows.length;
```

Borrar la primera fila de la tabla y la primera fila del cuerpo (sección <tbody>):

```
var tabla = document.getElementById('miTabla');
tabla.deleteRow(0);
tabla.tBodies[0].deleteRow(0);
```

Borrar la primera columna de la primera fila del cuerpo (sección <tbody>):

```
var tabla = document.getElementById('miTabla');
tabla.tBodies[0].rows[0].deleteCell(0);
```

Obtener el número de columnas de la primera parte del cuerpo (sección <tbody>):

```
var tabla = document.getElementById('miTabla');
var numColumnas = tabla.rows[0].cells.length;
```

Obtener el texto de la primera columna de la primera fila del cuerpo (sección <tbody>):

```
var tabla = document.getElementById('miTabla');
var texto = tabla.tBodies[0].rows[0].cells[0].innerHTML;
```

Recorrer todas las filas y todas las columnas de la tabla:

```
var tabla = document.getElementById('miTabla');
var filas = tabla.rows;
for(var i=0; i<filas.length; i++) {
  var fila = filas[i];
  var columnas = fila.cells;
  for(var j=0; j<columnas.length; j++) {
    var columna = columnas[j];
    // ...
  }
}
```

Insertar una tercera fila al cuerpo (sección <tbody>) de la tabla:

```
var tabla = document.getElementById('miTabla');

// Insertar la tercera fila
tabla.tBodies[0].insertRow(2);

// Crear la columna de tipo <th> que hace de cabecera de fila
var cabecera = document.createElement("th");
cabecera.setAttribute('scope', 'row');
cabecera.innerHTML = 'Cabecera fila 3'
tabla.tBodies[0].rows[2].appendChild(cabecera);
```

```
// Crear las dos columnas de datos y añadirlas a la nueva fila
tabla.tBodies[0].rows[2].insertCell(1);
tabla.tBodies[0].rows[2].cells[1].innerHTML = 'Celda 3 - 1';
tabla.tBodies[0].rows[2].insertCell(2);
tabla.tBodies[0].rows[2].cells[2].innerHTML = 'Celda 3 - 2';
```

Por último, se muestra de forma resumida el código JavaScript necesario para crear la tabla XHTML del ejemplo anterior:

```
// Crear <table> y sus dos atributos
var tabla = document.createElement('table');
tabla.setAttribute('id', 'otraTabla');
tabla.setAttribute('summary', 'Descripción de la tabla y su contenido');

// Crear <caption> y añadirlo a la <table>
var caption = document.createElement('caption');
var titulo = document.createTextNode('Título de la tabla');
caption.appendChild(titulo);
tabla.appendChild(caption);

// Crear sección <thead>
var thead = document.createElement('thead');
tabla.appendChild(thead);

// Añadir una fila a la sección <thead>
thead.insertRow(0);

// Añadir las tres columnas de la fila de <thead>
var cabecera = document.createElement('th');
cabecera.innerHTML = '';
thead.rows[0].appendChild(cabecera);

cabecera = document.createElement('th');
cabecera.setAttribute('scope', 'col');
cabecera.innerHTML = 'Cabecera columna 1';
tabla.rows[0].appendChild(cabecera);

cabecera = document.createElement('th');
cabecera.setAttribute('scope', 'col');
cabecera.innerHTML = 'Cabecera columna 2';
tabla.rows[0].appendChild(cabecera);

// La sección <tfoot> se crearía de forma similar a <thead>

// Crear sección <tbody>
var tbody = document.createElement('tbody');
tabla.appendChild(tbody);

// Añadir una fila a la sección <tbody>
tbody.insertRow(0);

cabecera = document.createElement("th");
cabecera.setAttribute('scope', 'row');
cabecera.innerHTML = 'Cabecera fila 1'
```

```
tabla.tBodies[0].rows[0].appendChild(cabecera);

tbody.rows[0].insertCell(1);
tbody.rows[0].cells[1].innerHTML = 'Celda 1 - 1';
// También se podría hacer:
// tbody.rows[0].cells[0].appendChild(document.createTextNode('Celda 1 - 1'));

tbody.rows[0].insertCell(2);
tbody.rows[0].cells[2].innerHTML = 'Celda 1 - 2';

// El resto de filas del <tbody> se crearía de la misma forma

// Añadir la tabla creada al final de la página
document.body.appendChild(tabla);
```

Capítulo 5. BOM (Browser Object Model)

5.1. Introducción a BOM

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:

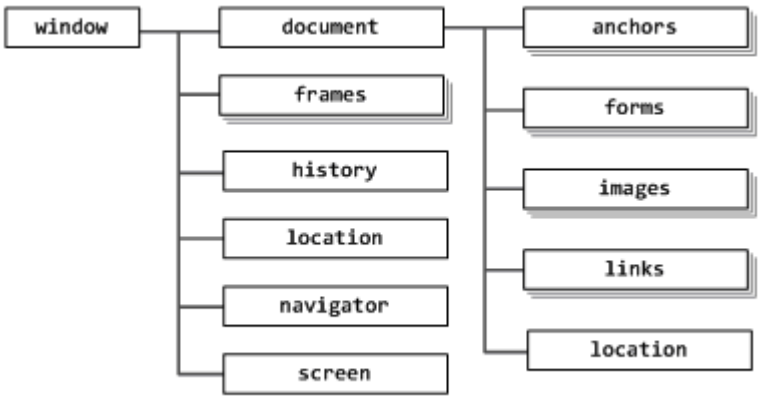

Figura 5.1. Jerarquía de objetos que forman el BOM

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`.

5.2. El 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.

Si una página emplea frames, cada uno de ellos se almacena en el array `frames`, que puede ser accedido numéricamente (`window.frames[0]`) o, si se ha indicado un nombre al frame, mediante su nombre (`window.frames["nombre del frame"]`).

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.frames[0] == frames[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);
```

Las funciones de *timeout* son imprescindibles para crear aplicaciones profesionales, ya que permiten, por ejemplo, que un *script* no espere infinito tiempo para obtener el resultado de una función.

En este caso, la estrategia consiste en establecer una cuenta atrás antes de llamar a la función. Si la función se ejecuta correctamente, en cuanto finalice su ejecución se elimina la cuenta atrás y continúa la ejecución normal del script. Si por cualquier motivo la función no se ejecuta correctamente, la cuenta atrás se cumple y la aplicación puede informar al usuario o reintentar la ejecución de la función.

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);
```

5.3. El objeto document

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:

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

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:

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

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`.

5.4. El objeto location

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:

Propiedad	Descripción

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

`host`	El nombre del servidor
	`http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion`
	`host` = `www.ejemplo.com`

`hostname`	La mayoría de las veces coincide con `host`, aunque en ocasiones, se eliminan las `www` del principio
	`http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion`
	`hostname` = `www.ejemplo.com`

`href`	La URL completa de la página actual `http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion` URL = `http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion`

`pathname`	Todo el contenido que se encuentra después del host
	`http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion`
	`pathname` = `/ruta1/ruta2/pagina.html`

`port`	Si se especifica en la URL, el puerto accedido
	`http://www.ejemplo.com:8080/ruta1/ruta2/pagina.html#seccion`
	`port` = `8080`
	La mayoría de URL no proporcionan un puerto, por lo que su contenido es vacío
	`http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion`
	`port` = (vacío)

`protocol`	El protocolo empleado por la URL, es decir, todo lo que se encuentra antes de las dos barras inclinadas `//`
	`http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion`
	`protocol` = `http:`

`search`	Todo el contenido que se encuentra tras el símbolo `?`, es decir, la consulta o *"query string"*
	`http://www.ejemplo.com/pagina.php?variable1=valor1&variable2=valor2`
	`search` = `?variable1=valor1&variable2=valor2`

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 */
```

5.5. El objeto navigator

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:

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

systemLanguage	(Sólo Internet Explorer) Cadena que representa el idioma del sistema operativo
userAgent	Cadena que representa la cadena que el navegador emplea para identificarse en los servidores
userLanguage	(Sólo Explorer) Cadena que representa el idioma del sistema operativo
userProfile	(Sólo Explorer) Objeto que permite acceder al perfil del usuario

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.

5.6. El objeto screen

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. Los diseñadores de páginas web necesitan conocer las resoluciones más utilizadas por los usuarios para adaptar sus diseños a esas resoluciones.

Las siguientes propiedades están disponibles en el objeto screen:

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

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);
```

Capítulo 6. Eventos

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.

6.1. Modelo básico de eventos

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.

6.1.1. Tipos de eventos

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:

Evento	Descripción	Elementos para los que está definido

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

Algunos de los eventos anteriores (onclick, onkeydown, onkeypress, onreset y onsubmit) permiten evitar el comportamiento por defecto del evento si se devuelve el valor false, tal y como se verá más adelante.

Por otra parte, las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Si se pulsa por ejemplo sobre un botón de tipo submit se desencadenan los eventos onmousedown, onmouseup, onclick y onsubmit.

6.1.2. Manejadores de eventos

En la programación orientada a eventos, las aplicaciones esperan a que se produzcan los eventos. Una vez que se produce un evento, la aplicación responde ejecutando cierto código especialmente preparado. Este tipo de código se denomina *"manejadores de eventos"* (del inglés *"event handlers"*) y las funciones externas que se definen para responder a los eventos se suelen denominar *"funciones manejadoras"*.

En las siguientes secciones se presentan las tres formas más utilizadas para indicar el código que se ejecuta cuando se produce un evento.

6.1.2.1. Manejadores como atributos XHTML

La forma más sencilla de incluir un manejador de evento es mediante un atributo de XHTML. El siguiente ejemplo muestra un mensaje cuando el usuario pincha en el botón:

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

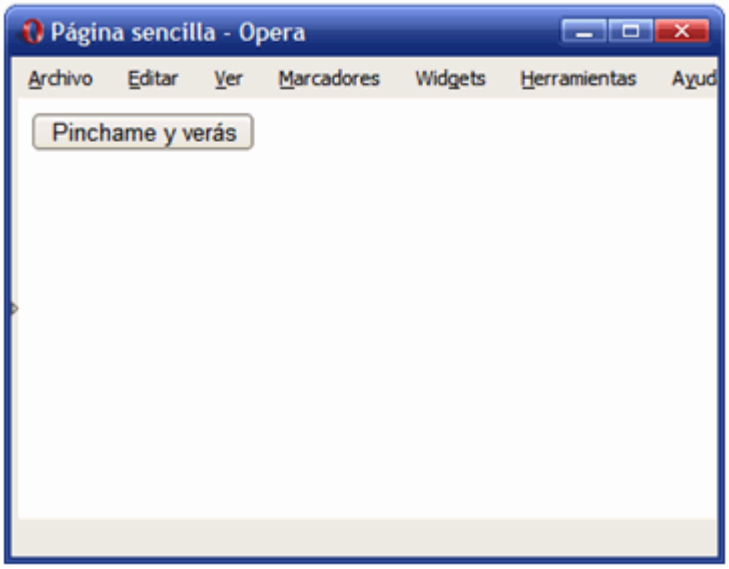

Figura 6.1. Página HTML

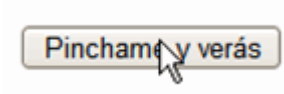

Figura 6.2. El usuario pincha con el ratón sobre el botón que se muestra

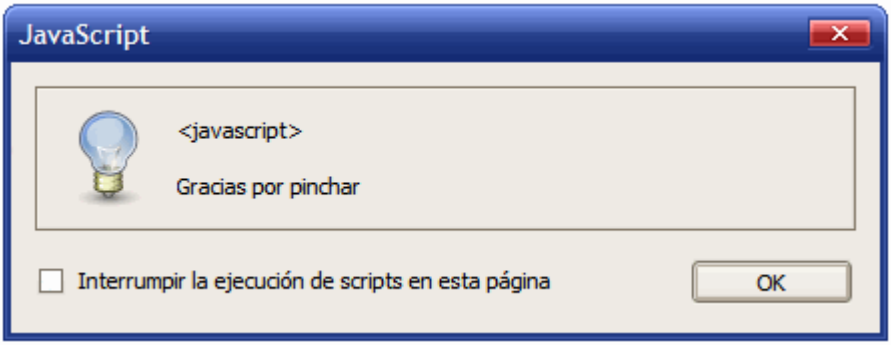

Figura 6.3. Después de pinchar con el ratón, se muestra un mensaje en una nueva ventana

El método consiste en incluir un atributo XHTML con el mismo nombre del evento que se quiere procesar. En este caso, como se quiere mostrar un mensaje cuando se pincha con el ratón sobre un botón, el evento es `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, ya que solamente se trata de mostrar un mensaje mediante la función `alert()`.

En este otro ejemplo, se muestra un mensaje cuando la página se ha cargado completamente:

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

El evento `onload` es uno de los más utilizados porque, como se vio en el capítulo de DOM, las funciones de acceso y manipulación de los nodos del árbol DOM solamente están disponibles cuando la página se carga por completo.

6.1.2.2. Manejadores de eventos y variable this

En los eventos de JavaScript, se puede utilizar la palabra reservada `this` para referirse al elemento XHTML sobre el que se está ejecutando el evento. Esta técnica es útil para ejemplos como el siguiente, en el que se modifican las propiedades del elemento que provoca el evento.

En el siguiente ejemplo, se muestra un borde destacado en el elemento `<div>` cuando el usuario pasa el ratón por encima. Cuando el ratón sale del `<div>`, se vuelve a mostrar el borde original:

```
<div id="elemento" style="padding: .2em; width: 150px; height: 60px; border: thin solid
silver" onmouseover = "document.getElementById('elemento').style.borderColor = 'black'"
onmouseout = "document.getElementById('elemento').style.borderColor = 'silver'">
  Sección de contenidos...
</div>
```

El evento `onmouseover` se activa cuando el ratón pasa por encima del elemento, en este caso el `<div>`. Para cambiar el color del borde, se utiliza la propiedad `border-color` de CSS. Por lo tanto, en primer lugar se obtiene la referencia del elemento mediante `document.getElementById('elemento')`. A continuación, se utiliza la propiedad `style` para acceder a las propiedades CSS y se modifica el nombre de `border-color` por el de `borderColor`. Para volver al color original cuando el ratón sale del elemento, se realiza la misma operación sobre el evento `onmouseout`.

El uso de la variable `this` puede simplificar todos estos pasos, ya que dentro de un manejador de eventos, la variable `this` equivale al elemento que ha provocado el evento. Así, la variable `this` es igual al `<div>` de la página y por tanto `this.style.borderColor` permite cambiar de forma directa el color del borde del `<div>`:

```
<div style="padding: .2em; width: 150px; height: 60px; border: thin solid silver"
onmouseover="this.style.borderColor='black'"
onmouseout="this.style.borderColor='silver'">
  Sección de contenidos...
</div>
```

Haciendo uso de la variable `this`, el código es mucho más sencillo de escribir, leer y mantener.

6.1.2.3. Manejadores de eventos como funciones externas

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`.

6.1.2.4. Manejadores de eventos semánticos

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() {
  ...
}
```

Ejercicio 10

A partir de la página web proporcionada y utilizando manejadores de eventos semánticos, 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

Cuando la sección se oculte, debe cambiar el mensaje del enlace asociado (pista: propiedad `innerHTML`).

6.2. El flujo de eventos

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.

6.2.1. Event bubbling

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:

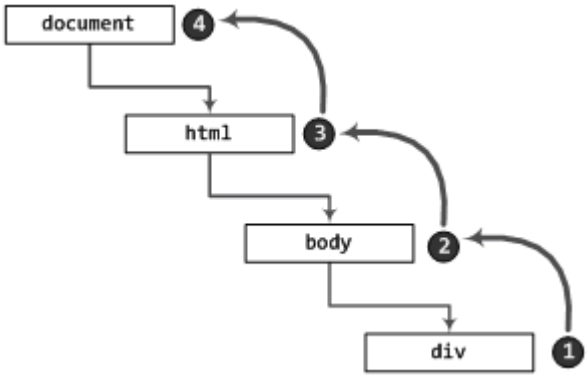

Figura 6.4. Esquema del funcionamiento del "event bubbling"

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:

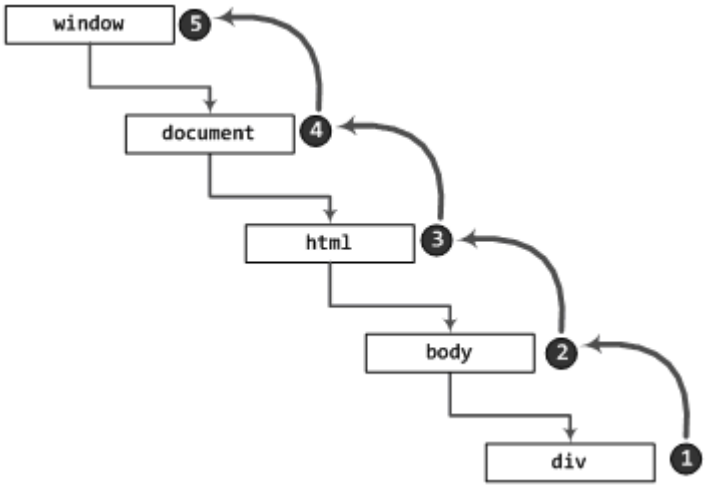

Figura 6.5. Esquema del funcionamiento del "event bubbling" en los navegadores de Mozilla (por ejemplo, Firefox)

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.

6.2.2. Event capturing

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.

6.2.3. Eventos DOM

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:

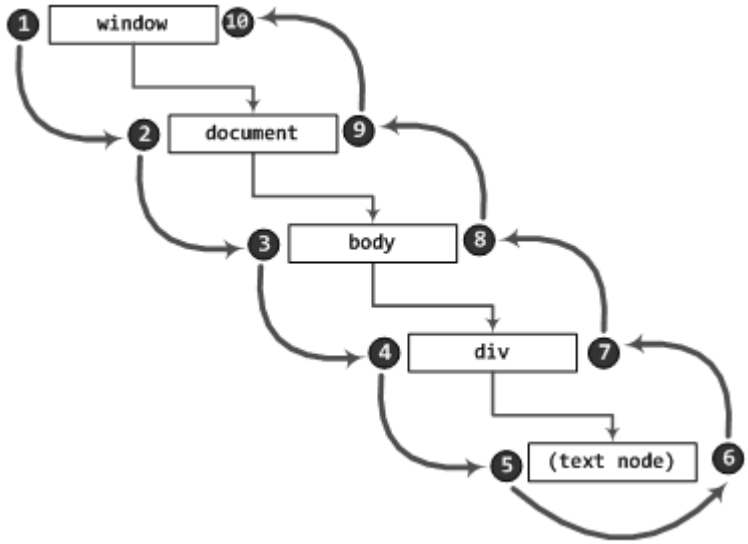

Figura 6.6. Esquema del flujo de eventos del modelo DOM

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.

6.3. Handlers y listeners

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.

6.3.1. Manejadores de eventos de Internet Explorer

La familia de navegadores de Internet Explorer emplean los métodos `attachEvent()` y `detachEvent()` para asociar y desasociar manejadores de eventos. Todos los elementos y el objeto `window` disponen de estos métodos.

Los métodos requieren dos parámetros: el nombre del evento que se quiere manejar y una referencia a la función encargada de procesar el evento. El nombre del evento se debe indicar con el prefijo `on` incluido, como muestra el siguiente ejemplo:

```
function muestraMensaje() {
  alert("Has pulsado el ratón");
}
var elDiv = document.getElementById("div_principal");
elDiv.attachEvent("onclick", muestraMensaje);

// Más adelante se decide desasociar la función del evento
elDiv.detachEvent("onclick", muestraMensaje);
```

Asociar más de una función al mismo evento es igual de sencillo:

```
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.attachEvent("onclick", muestraMensaje);
elDiv.attachEvent("onclick", muestraOtroMensaje);
```

Si el usuario pincha sobre el `<div>`, se muestran los dos mensajes de aviso que se han asignado al evento.

Se pueden mezclar incluso diferentes técnicas para asociar múltiples manejadores de eventos a un mismo evento de un elemento. El siguiente ejemplo utiliza la asignación semántica de manejadores para asignar el primer manejador y la función `attachEvent()` para asignar el segundo manejador. El resultado final es igual al del ejemplo anterior:

```
var elDiv = document.getElementById("div_principal");
elDiv.onclick = muestraMensaje;
elDiv.attachEvent("onclick", muestraOtroMensaje);
```

Sin embargo, no es posible asignar múltiples eventos mediante las propiedades de DOM:

```
var elDiv = document.getElementById("div_principal");
elDiv.onclick = muestraMensaje;
elDiv.onclick = muestraOtroMensaje;
```

6.3.2. Manejadores de eventos de DOM

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 *capture*. 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.

6.4. El objeto event

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 automáticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento.

Internet Explorer permite el acceso al objeto `event` a través del objeto `window`:

```
elDiv.onclick = function() {
  var elEvento = window.event;
}
```

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(elEvento) {
  ...
}
```

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.

6.4.1. Propiedades y métodos

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.

6.4.1.1. Propiedades definidas por Internet Explorer

La siguiente tabla recoge las propiedades definidas para el objeto `event` en los navegadores de la familia Internet Explorer:

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

shiftKey	Boolean	Devuelve true si se ha pulsado la tecla SHIFT y false en otro caso
srcElement	Element	El elemento que origina el evento
toElement	Element	El elemento al que entra el ratón (para ciertos eventos de ratón)
type	Cadena de texto	El nombre del evento
x	Número entero	Coordenada X de la posición del ratón respecto del elemento padre del elemento que origina el evento
y	Número entero	Coordenada Y de la posición del ratón respecto del elemento padre del elemento que origina el evento

Todas las propiedades salvo repeat son de lectura/escritura y por tanto, su valor se puede leer y/o establecer.

6.4.1.2. Propiedades definidas por DOM

La siguiente tabla recoge las propiedades definidas para el objeto event en los navegadores que siguen los estándares:

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

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

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`.

6.4.2. Similitudes y diferencias entre navegadores

6.4.2.1. Similitudes

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`.

6.4.2.2. Diferencias

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:

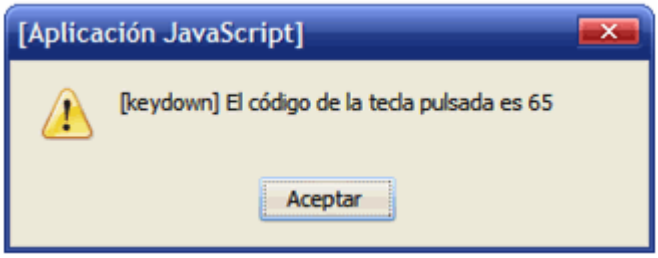

Figura 6.7. Mensaje mostrado en el navegador Firefox

Figura 6.8. Mensaje mostrado en el navegador Internet Explorer

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:

```
// Internet Explorer
elEvento.returnValue = false;

// 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:

```
// Internet Explorer
elEvento.cancelBubble = true;

// 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`.

6.5. Tipos de eventos

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".

6.5.1. Eventos de ratón

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:

Evento	Descripción
`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.

6.5.1.1. Propiedades

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`.

6.5.2. Eventos de teclado

Los eventos que se incluyen en esta clasificación son los siguientes:

Evento	Descripción
`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

6.5.2.1. Propiedades

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.

6.5.3. Eventos HTML

Los eventos HTML definidos se recogen en la siguiente tabla:

Evento	Descripción
`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 `<object>` 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.

6.5.4. Eventos DOM

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:

Evento	**Descripción**
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

6.6. Solución cross browser

Las diferencias existentes entre los navegadores disponibles en la actualidad complican en exceso el desarrollo de aplicaciones compatibles con todos los navegadores, llamadas aplicaciones *"cross browser"* en inglés.

Por ese motivo, se va a diseñar una utilidad que permit unificar la asociación/desasociación de manejadores de eventos, la obtención del objeto event y todas sus propiedades. La utilidad que se muestra se ha obtenido del excelente libro *"Professional JavaScript for Web Developers"*, escrito por Nicholas C. Zakas y publicado por la editorial Wrox.

6.6.1. Asignación de manejadores de eventos

En primer lugar, se crea el objeto que va a englobar todas las propiedades y métodos relacionados con los eventos:

```
var EventUtil = new Object();
```

El primer método relacionado con los eventos que es necesario estandarizar es el de la asignación y eliminación de manejadores de eventos:

```
EventUtil.addEventHandler = function(elemento, tipoEvento, funcion) {
  if(elemento.addEventListener) { // navegadores DOM
    elemento.addEventListener(tipoEvento, funcion, false);
  }
  else if(elemento.attachEvent) { // Internet Explorer
    elemento.attachEvent("on"+tipoEvento, funcion);
  }
  else { // resto de navegadores
    elemento["on"+tipoEvento] = funcion;
  }
};

EventUtil.removeEventHandler = function(elemento, tipoEvento, funcion) {
  if(elemento.removeEventListener) { // navegadores DOM
    elemento.removeEventListener(tipoEvento, funcion, false);
  }
  else if(elemento.detachEvent) { // Internet Explorer
    elemento.detachEvent("on"+tipoEvento, funcion);
  }
  else { // resto de navegadores
    elemento["on"+tipoEvento] = null;
  }
};
```

6.6.2. Obtención del objeto Event

Para obtener el objeto `event`, se crea un nuevo método en la utilidad llamado `getEvent()`:

```
EventUtil.getEvent = function() {
  if(window.event) { // Internet Explorer
    return this.formatEvent(window.event);
  }
  else { // navegadores DOM
    return EventUtil.getEvent.caller.arguments[0];
  }
};
```

El método `getEvent()` es un método que no acepta parámetros y que devuelve el objeto `event` convenientemente adaptado para permitir un comportamiento homogéneo entre diferentes navegadores.

En el caso de Internet Explorer, el objeto `event` se obtiene directamente a partir del objeto `window`. Sin embargo, antes de devolver el objeto, se modifica añadiendo las propiedades que no dispone en comparación con el objeto `event` de los navegadores DOM.

En el caso de los navegadores DOM, el objeto `event` se obtiene como el primer argumento de la función que actúa como manejador del evento. Como ya se vio en el capítulo de JavaScript básico, la propiedad `caller` de una función siempre almacena una referencia a la función que la invocó.

Así, si en el interior de un manejador de eventos se hace la llamada al método `EventUtil.getEvent()`, la propiedad `caller` será el propio manejador de eventos y su primer argumento será el objeto `event`. Parece muy abstracto, pero si se piensa detenidamente se comprende fácilmente la solución tan concisa y elegante que se ha utilizado.

6.6.3. Estandarización del objeto Event

El objeto `event` presenta unas propiedades y métodos muy diferentes en función del tipo de navegador en el que se ejecuta la aplicación JavaScript. Para estandarizar el objeto `event`, se crea un método que añade al objeto `event` de Internet Explorer todas las propiedades que le faltan.

El código completo de este método se muestra a continuación:

```
EventUtil.formatEvent = function(elEvento) {
  // Detectar si el navegador actual es Internet Explorer
  var esIE = navigator.userAgent.toLowerCase().indexOf('msie')!=-1;
  if(esIE) {
    elEvento.charCode = (elEvento.type=="keypress") ? elEvento.keyCode : 0;
    elEvento.eventPhase = 2;
    elEvento.isChar = (elEvento.charCode > 0);
    elEvento.pageX = elEvento.clientX + document.body.scrollLeft;
    elEvento.pageY = elEvento.clientY + document.body.scrollTop;
    elEvento.preventDefault = function() {
      this.returnValue = false;
    };
    if(elEvento.type == "mouseout") {
      elEvento.relatedTarget = elEvento.toElement;
    }
    else if(elEvento.type == "mouseover") {
      elEvento.relatedTarget = elEvento.fromElement
    }
    elEvento.stopPropagation = function() {
      this.cancelBuble = true;
    };
    elEvento.target = elEvento.srcElement;
    elEvento.time = (new Date).getTime();
  }
  return elEvento;
}
```

Capítulo 7. Primeros pasos con AJAX

7.1. Breve historia de AJAX

La historia de AJAX está íntimamente relacionada con un objeto de programación llamado `XMLHttpRequest`. El origen de este objeto se remonta al año 2000, con productos como Exchange 2000, Internet Explorer 5 y Outlook Web Access.

Todo comenzó en 1998, cuando **Alex Hopmann** y su equipo se encontraban desarrollando la entonces futura versión de Exchange 2000. El punto débil del servidor de correo electrónico era su cliente vía web, llamado OWA (*Outlook Web Access*).

Durante el desarrollo de OWA, se evaluaron dos opciones: un cliente formado sólo por páginas HTML estáticas que se recargaban constantemente y un cliente realizado completamente con HTML dinámico o DHTML. Alex Hopmann pudo ver las dos opciones y se decantó por la basada en DHTML. Sin embargo, para ser realmente útil a esta última le faltaba un componente esencial: "algo" que evitara tener que enviar continuamente los formularios con datos al servidor.

Motivado por las posibilidades futuras de OWA, Alex creó en un solo fin de semana la primera versión de lo que denominó XMLHTTP. La primera demostración de las posibilidades de la nueva tecnología fue un éxito, pero faltaba lo más difícil: incluir esa tecnología en el navegador Internet Explorer.

Si el navegador no incluía XMLHTTP de forma nativa, el éxito del OWA se habría reducido enormemente. El mayor problema es que faltaban pocas semanas para que se lanzara la última beta de Internet Explorer 5 previa a su lanzamiento final. Gracias a sus contactos en la empresa, Alex consiguió que su tecnología se incluyera en la librería MSXML que incluye Internet Explorer.

De hecho, el nombre del objeto (`XMLHTTP`) se eligió para tener una buena excusa que justificara su inclusión en la librería XML de Internet Explorer, ya que este objeto está mucho más relacionado con HTTP que con XML.

7.2. La primera aplicación

7.2.1. Código fuente

La aplicación AJAX completa más sencilla consiste en una adaptación del clásico *"Hola Mundo"*. En este caso, una aplicación JavaScript descarga un archivo del servidor y muestra su contenido sin necesidad de recargar la página.

Código fuente completo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
```

```
<title>Hola Mundo con AJAX</title>

<script type="text/javascript">
function descargaArchivo() {
  // Obtener la instancia del objeto XMLHttpRequest
  if(window.XMLHttpRequest) {
    peticion_http = new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
  }

  // Preparar la funcion de respuesta
  peticion_http.onreadystatechange = muestraContenido;

  // Realizar peticion HTTP
  peticion_http.open('GET', 'http://localhost/holamundo.txt', true);
  peticion_http.send(null);

  function muestraContenido() {
    if(peticion_http.readyState == 4) {
      if(peticion_http.status == 200) {
        alert(peticion_http.responseText);
      }
    }
  }
}

window.onload = descargaArchivo;
</script>

</head>
<body></body>
</html>
```

En el ejemplo anterior, cuando se carga la página se ejecuta el método JavaScript que muestra el contenido de un archivo llamado `holamundo.txt` que se encuentra en el servidor. La clave del código anterior es que la petición HTTP y la descarga de los contenidos del archivo se realizan sin necesidad de recargar la página.

7.2.2. Análisis detallado

La aplicación AJAX del ejemplo anterior se compone de cuatro grandes bloques: instanciar el objeto `XMLHttpRequest`, preparar la función de respuesta, realizar la petición al servidor y ejecutar la función de respuesta.

Todas las aplicaciones realizadas con técnicas de AJAX deben instanciar en primer lugar el objeto `XMLHttpRequest`, que es el objeto clave que permite realizar comunicaciones con el servidor en segundo plano, sin necesidad de recargar las páginas.

La implementación del objeto `XMLHttpRequest` depende de cada navegador, por lo que es necesario emplear una discriminación sencilla en función del navegador en el que se está ejecutando el código:

```
if(window.XMLHttpRequest) {  // Navegadores que siguen los estándares
  peticion_http = new XMLHttpRequest();
}
else if(window.ActiveXObject) {  // Navegadores obsoletos
  peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Los navegadores que siguen los estándares (Firefox, Safari, Opera, Internet Explorer 7 y 8) implementan el objeto `XMLHttpRequest` de forma nativa, por lo que se puede obtener a través del objeto `window`. Los navegadores obsoletos (Internet Explorer 6 y anteriores) implementan el objeto `XMLHttpRequest` como un objeto de tipo ActiveX.

Una vez obtenida la instancia del objeto `XMLHttpRequest`, se prepara la función que se encarga de procesar la respuesta del servidor. La propiedad `onreadystatechange` del objeto `XMLHttpRequest` permite indicar esta función directamente incluyendo su código mediante una función anónima o indicando una referencia a una función independiente. En el ejemplo anterior se indica directamente el nombre de la función:

```
peticion_http.onreadystatechange = muestraContenido;
```

El código anterior indica que cuando la aplicación reciba la respuesta del servidor, se debe ejecutar la función `muestraContenido()`. Como es habitual, la referencia a la función se indica mediante su nombre sin paréntesis, ya que de otro modo se estaría ejecutando la función y almacenando el valor devuelto en la propiedad `onreadystatechange`.

Después de preparar la aplicación para la respuesta del servidor, se realiza la petición HTTP al servidor:

```
peticion_http.open('GET', 'http://localhost/prueba.txt', true);
peticion_http.send(null);
```

Las instrucciones anteriores realizan el tipo de petición más sencillo que se puede enviar al servidor. En concreto, se trata de una petición de tipo `GET` simple que no envía ningún parámetro al servidor. La petición HTTP se crea mediante el método `open()`, en el que se incluye el tipo de petición (`GET`), la URL solicitada (`http://localhost/prueba.txt`) y un tercer parámetro que vale `true`.

Una vez creada la petición HTTP, se envía al servidor mediante el método `send()`. Este método incluye un parámetro que en el ejemplo anterior vale `null`. Más adelante se ven en detalle todos los métodos y propiedades que permiten hacer las peticiones al servidor.

Por último, cuando se recibe la respuesta del servidor, la aplicación ejecuta de forma automática la función establecida anteriormente.

```
function muestraContenido() {
  if(peticion_http.readyState == 4) {
    if(peticion_http.status == 200) {
      alert(peticion_http.responseText);
    }
  }
}
```

La función `muestraContenido()` comprueba en primer lugar que se ha recibido la respuesta del servidor (mediante el valor de la propiedad `readyState`). Si se ha recibido alguna respuesta, se comprueba que sea válida y correcta (comprobando si el código de estado HTTP devuelto es igual a `200`). Una vez realizadas las comprobaciones, simplemente se muestra por pantalla el contenido de la respuesta del servidor (en este caso, el contenido del archivo solicitado) mediante la propiedad `responseText`.

7.2.3. Refactorizando la primera aplicación

La primera aplicación AJAX mostrada anteriormente presenta algunas carencias importantes. A continuación, se refactoriza su código ampliándolo y mejorándolo para que se adapte mejor a otras situaciones. En primer lugar, se definen unas variables que se utilizan en la función que procesa la respuesta del servidor:

```
var READY_STATE_UNINITIALIZED = 0;
var READY_STATE_LOADING = 1;
var READY_STATE_LOADED = 2;
var READY_STATE_INTERACTIVE = 3;
var READY_STATE_COMPLETE = 4;
```

Como se verá más adelante, la respuesta del servidor sólo puede corresponder a alguno de los cinco estados definidos por las variables anteriores. De esta forma, el código puede utilizar el nombre de cada estado en vez de su valor numérico, por lo que se facilita la lectura y el mantenimiento de las aplicaciones.

Además, la variable que almacena la instancia del objeto `XMLHttpRequest` se va a transformar en una variable global, de forma que todas las funciones que hacen uso de ese objeto tengan acceso directo al mismo:

```
var peticion_http;
```

A continuación, se crea una función genérica de carga de contenidos mediante AJAX:

```
function cargaContenido(url, metodo, funcion) {
  peticion_http = inicializa_xhr();

  if(peticion_http) {
    peticion_http.onreadystatechange = funcion;
    peticion_http.open(metodo, url, true);
    peticion_http.send(null);
  }
}
```

La función definida admite tres parámetros: la URL del contenido que se va a cargar, el método utilizado para realizar la petición HTTP y una referencia a la función que procesa la respuesta del servidor.

En primer lugar, la función `cargaContenido()` inicializa el objeto `XMLHttpRequest` (llamado `xhr` de forma abreviada). Una vez inicializado, se emplea el objeto `peticion_http` para establecer la función que procesa la respuesta del servidor. Por último, la función `cargaContenido()` realiza la petición al servidor empleando la URL y el método HTTP indicados como parámetros.

La función `inicializa_xhr()` se emplea para encapsular la creación del objeto `XMLHttpRequest`:

```
function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}
```

La función `muestraContenido()` también se refactoriza para emplear las variables globales definidas:

```
function muestraContenido() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      alert(peticion_http.responseText);
    }
  }
}
```

Por último, la función `descargaArchivo()` simplemente realiza una llamada a la función `cargaContenido()` con los parámetros adecuados:

```
function descargaArchivo() {
  cargaContenido("http://localhost/holamundo.txt", "GET", muestraContenido);
}
```

A continuación se muestra el código completo de la refactorización de la primera aplicación:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Hola Mundo con AJAX, version 2</title>

<script type="text/javascript" language="javascript">

var READY_STATE_UNINITIALIZED=0;
var READY_STATE_LOADING=1;
var READY_STATE_LOADED=2;
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;

var peticion_http;

function cargaContenido(url, metodo, funcion) {
  peticion_http = inicializa_xhr();

  if(peticion_http) {
    peticion_http.onreadystatechange = funcion;
    peticion_http.open(metodo, url, true);
    peticion_http.send(null);
  }
}
```

```
function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function muestraContenido() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      alert(peticion_http.responseText);
    }
  }
}

function descargaArchivo() {
  cargaContenido("http://localhost/holamundo.txt", "GET", muestraContenido);
}

window.onload = descargaArchivo;

</script>

</head>
<body></body>
</html>
```

Ejercicio 11

A partir de la página web proporcionada, añadir el código JavaScript necesario para que:

1. Al cargar la página, el cuadro de texto debe mostrar por defecto la URL de la propia página.
2. Al pulsar el botón "Mostrar Contenidos", se debe descargar mediante peticiones AJAX el contenido correspondiente a la URL introducida por el usuario. El contenido de la respuesta recibida del servidor se debe mostrar en la zona de "Contenidos del archivo".
3. En la zona "Estados de la petición" se debe mostrar en todo momento el estado en el que se encuentra la petición (No inicializada, cargando, completada, etc.)
4. Mostrar el contenido de todas las cabeceras de la respuesta del servidor en la zona "Cabeceras HTTP de la respuesta del servidor".
5. Mostrar el código y texto de estado de la respuesta del servidor en la zona "Código de estado".

7.3. Métodos y propiedades del objeto XMLHttpRequest

El objeto `XMLHttpRequest` posee muchas otras propiedades y métodos diferentes a las manejadas por la primera aplicación de AJAX. A continuación se incluye la lista completa de todas las propiedades y métodos del objeto y todos los valores numéricos de sus propiedades.

Las propiedades definidas para el objeto `XMLHttpRequest` son:

Propiedad	Descripción
readyState	Valor numérico (entero) que almacena el estado de la petición
responseText	El contenido de la respuesta del servidor en forma de cadena de texto
responseXML	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM
status	El código de estado HTTP devuelto por el servidor (200 para una respuesta correcta, 404 para "No encontrado", 500 para un error de servidor, etc.)
statusText	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.

Los valores definidos para la propiedad readyState son los siguientes:

Valor	Descripción
0	No inicializado (objeto creado, pero no se ha invocado el método open)
1	Cargando (objeto creado, pero no se ha invocado el método send)
2	Cargado (se ha invocado el método send, pero el servidor aún no ha respondido)
3	Interactivo (se han recibido algunos datos, aunque no se puede emplear la propiedad responseText)
4	Completo (se han recibido todos los datos de la respuesta del servidor)

Los métodos disponibles para el objeto XMLHttpRequest son los siguientes:

Método	Descripción
abort()	Detiene la petición actual
getAllResponseHeaders()	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor
getResponseHeader("cabecera")	Devuelve una cadena de texto con el contenido de la cabecera solicitada
onreadystatechange	Responsable de manejar los eventos que se producen. Se invoca cada vez que se produce un cambio en el estado de la petición HTTP. Normalmente es una referencia a una función JavaScript
open("metodo", "url")	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino (puede indicarse de forma absoluta o relativa)
send(contenido)	Realiza la petición HTTP al servidor
setRequestHeader("cabecera", "valor")	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar el método open() antes que setRequestHeader()

El método open() requiere dos parámetros (método HTTP y URL) y acepta de forma opcional otros tres parámetros. Definición formal del método open():

```
open(string metodo, string URL [,boolean asincrono, string usuario, string password]);
```

Por defecto, las peticiones realizadas son asíncronas. Si se indica un valor `false` al tercer parámetro, la petición se realiza de forma síncrona, esto es, se detiene la ejecución de la aplicación hasta que se recibe de forma completa la respuesta del servidor.

No obstante, las peticiones síncronas son justamente contrarias a la filosofía de AJAX. El motivo es que una petición síncrona *congela* el navegador y no permite al usuario realizar ninguna acción hasta que no se haya recibido la respuesta completa del servidor. La sensación que provoca es que el navegador se ha *colgado* por lo que no se recomienda el uso de peticiones síncronas salvo que sea imprescindible.

Los últimos dos parámetros opcionales permiten indicar un nombre de usuario y una contraseña válidos para acceder al recurso solicitado.

Por otra parte, el método `send()` requiere de un parámetro que indica la información que se va a enviar al servidor junto con la petición HTTP. Si no se envían datos, se debe indicar un valor igual a `null`. En otro caso, se puede indicar como parámetro una cadena de texto, un array de bytes o un objeto XML DOM.

7.4. Utilidades y objetos para AJAX

Una de las operaciones más habituales en las aplicaciones AJAX es la de obtener el contenido de un archivo o recurso del servidor. Por tanto, se va a construir un objeto que permita realizar la carga de datos del servidor simplemente indicando el recurso solicitado y la función encargada de procesar la respuesta:

```
var cargador = new net.CargadorContenidos("pagina.html", procesaRespuesta);
```

La lógica común de AJAX se encapsula en un objeto de forma que sea fácilmente reutilizable. Aplicando los conceptos de objetos de JavaScript, funciones constructoras y el uso de prototype, es posible realizar de forma sencilla el objeto cargador de contenidos.

El siguiente código ha sido adaptado del excelente libro "Ajax in Action", escrito por Dave Crane, Eric Pascarello y Darren James y publicado por la editorial Manning.

```
var net = new Object();

net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;

// Constructor
net.CargadorContenidos = function(url, funcion, funcionError) {
  this.url = url;
  this.req = null;
  this.onload = funcion;
  this.onerror = (funcionError) ? funcionError : this.defaultError;
  this.cargaContenidoXML(url);
}

net.CargadorContenidos.prototype = {
```

```
  cargaContenidoXML: function(url) {
    if(window.XMLHttpRequest) {
      this.req = new XMLHttpRequest();
    }
    else if(window.ActiveXObject) {
      this.req = new ActiveXObject("Microsoft.XMLHTTP");
    }

    if(this.req) {
      try {
        var loader = this;
        this.req.onreadystatechange = function() {
          loader.onReadyState.call(loader);
        }
        this.req.open('GET', url, true);
        this.req.send(null);
      } catch(err) {
        this.onerror.call(this);
      }
    }
  },

  onReadyState: function() {
    var req = this.req;
    var ready = req.readyState;
    if(ready == net.READY_STATE_COMPLETE) {
      var httpStatus = req.status;
      if(httpStatus == 200 || httpStatus == 0) {
        this.onload.call(this);
      }
      else {
        this.onerror.call(this);
      }
    }
  },

  defaultError: function() {
    alert("Se ha producido un error al obtener los datos"
      + "\n\nreadyState:" + this.req.readyState
      + "\nstatus: " + this.req.status
      + "\nheaders: " + this.req.getAllResponseHeaders());
  }
}
```

Una vez definido el objeto `net` con su método `CargadorContenidos()`, ya es posible utilizarlo en las funciones que se encargan de mostrar el contenido del archivo del servidor:

```
function muestraContenido() {
  alert(this.req.responseText);
}

function cargaContenidos() {
  var cargador = new net.CargadorContenidos("http://localhost/holamundo.txt",
muestraContenido);
}
```

```
window.onload = cargaContenidos;
```

En el ejemplo anterior, la aplicación muestra un mensaje con los contenidos de la URL indicada:

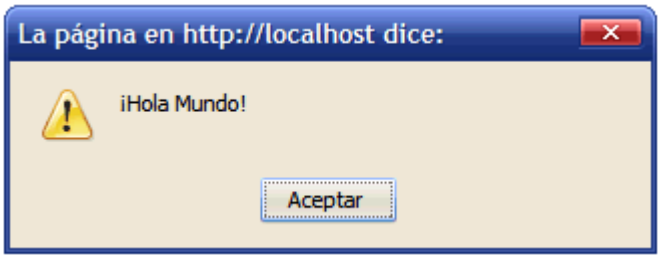

Figura 7.1. Mensaje mostrado cuando el resultado es exitoso

Por otra parte, si la URL que se quiere cargar no es válida o el servidor no responde, la aplicación muestra el siguiente mensaje de error:

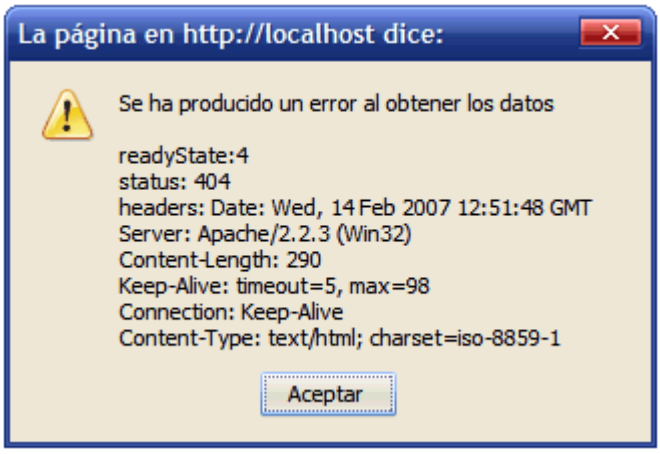

Figura 7.2. Mensaje mostrado cuando el resultado es erróneo

El código del cargador de contenidos hace un uso intensivo de objetos, JSON, funciones anónimas y uso del objeto `this`. Seguidamente, se detalla el funcionamiento de cada una de sus partes.

El primer elemento importante del código fuente es la definición del objeto `net`.

```
var net = new Object();
```

Se trata de una variable global que encapsula todas las propiedades y métodos relativos a las operaciones relacionadas con las comunicaciones por red. De cierto modo, esta variable global simula el funcionamiento de los *namespaces* ya que evita la colisión entre nombres de propiedades y métodos diferentes.

Después de definir las constantes empleadas por el objeto `XMLHttpRequest`, se define el constructor del objeto `CargadorContenidos`:

```
net.CargadorContenidos = function(url, funcion, funcionError) {
  this.url = url;
  this.req = null;
  this.onload = funcion;
  this.onerror = (funcionError) ? funcionError : this.defaultError;
  this.cargaContenidoXML(url);
}
```

Aunque el constructor define tres parámetros diferentes, en realidad solamente los dos primeros son obligatorios. De esta forma, se inicializa el valor de algunas variables del objeto, se comprueba si se ha definido la función que se emplea en caso de error (si no se ha definido, se emplea una función genérica definida más adelante) y se invoca el método responsable de cargar el recurso solicitado (`cargaContenidoXML`).

```
net.CargadorContenidos.prototype = {
  cargaContenidoXML:function(url) {
    ...
  },
  onReadyState:function() {
    ...
  },
  defaultError:function() {
    ...
  }
}
```

Los métodos empleados por el objeto `net.cargaContenidos` se definen mediante su prototipo. En este caso, se definen tres métodos diferentes: `cargaContenidoXML()` para cargar recursos de servidor, `onReadyState()` que es la función que se invoca cuando se recibe la respuesta del servidor y `defaultError()` que es la función que se emplea cuando no se ha definido de forma explícita una función responsable de manejar los posibles errores que se produzcan en la petición HTTP.

La función `defaultError()` muestra un mensaje de aviso del error producido y además muestra el valor de algunas de las propiedades de la petición HTTP:

```
defaultError:function() {
  alert("Se ha producido un error al obtener los datos"
    + "\n\nreadyState:" + this.req.readyState
    + "\nstatus: " + this.req.status
    + "\nheaders: " + this.req.getAllResponseHeaders());
}
```

En este caso, el objeto `this` se resuelve al objeto `net.cargaContenidos`, ya que es el objeto que contiene la función anónima que se está ejecutando.

Por otra parte, la función `onReadyState` es la encargada de gestionar la respuesta del servidor:

```
onReadyState: function() {
  var req = this.req;
  var ready = req.readyState;
  if(ready == net.READY_STATE_COMPLETE) {
    var httpStatus = req.status;
    if(httpStatus == 200 || httpStatus == 0) {
      this.onload.call(this);
    } else {
      this.onerror.call(this);
    }
  }
}
```

Tras comprobar que la respuesta del servidor está disponible y es correcta, se realiza la llamada a la función que realmente procesa la respuesta del servidor de acuerdo a las necesidades de la aplicación.

```
this.onload.call(this);
```

El objeto `this` se resuelve como `net.CargadorContenidos`, ya que es el objeto que contiene la función que se está ejecutando. Por tanto, `this.onload` es la referencia a la función que se ha definido como responsable de procesar la respuesta del servidor (se trata de una referencia a una función externa).

Normalmente, la función externa encargada de procesar la respuesta del servidor, requerirá acceder al objeto `XMLHttpRequest` que almacena la petición realizada al servidor. En otro caso, la función externa no será capaz de acceder al contenido devuelto por el servidor.

Como ya se vio en los capítulos anteriores, el método `call()` es uno de los métodos definidos para el objeto `Function()`, y por tanto disponible para todas las funciones de JavaScript. Empleando el método `call()` es posible obligar a una función a ejecutarse sobre un objeto concreto. En otras palabras, empleando el método `call()` sobre una función, es posible que dentro de esa función el objeto `this` se resuelva como el objeto pasado como parámetro en el método `call()`.

Así, la instrucción `this.onload.call(this);` se interpreta de la siguiente forma:

- El objeto `this` que se pasa como parámetro de `call()` se resuelve como el objeto `net.CargadorContenidos`.
- El objeto `this.onload` almacena una referencia a la función externa que se va a emplear para procesar la respuesta.
- El método `this.onload.call()` ejecuta la función cuya referencia se almacena en `this.onload`.
- La instrucción `this.onload.call(this);` permite ejecutar la función externa con el objeto `net.CargadorContenidos` accesible en el interior de la función mediante el objeto `this`.

Por último, el método `cargaContenidoXML` se encarga de enviar la petición HTTP y realizar la llamada a la función que procesa la respuesta:

```
cargaContenidoXML:function(url) {
  if(window.XMLHttpRequest) {
    this.req = new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    this.req = new ActiveXObject("Microsoft.XMLHTTP");
  }
  if(this.req) {
    try {
      var loader=this;
      this.req.onreadystatechange = function() {
        loader.onReadyState.call(loader);
      }
```

```
      this.req.open('GET', url, true);
      this.req.send(null);
    } catch(err) {
      this.onerror.call(this);
    }
  }
}
```

En primer lugar, se obtiene una instancia del objeto `XMLHttpRequest` en función del tipo de navegador. Si se ha obtenido correctamente la instancia, se ejecutan las instrucciones más importantes del método `cargaContenidoXML`:

```
var loader = this;
this.req.onreadystatechange = function() {
  loader.onReadyState.call(loader);
}
this.req.open('GET', url, true);
this.req.send(null);
```

A continuación, se almacena la instancia del objeto actual (`this`) en la nueva variable `loader`. Una vez almacenada la instancia del objeto `net.cargadorContenidos`, se define la función encargada de procesar la respuesta del servidor. En la siguiente función anónima:

```
this.req.onreadystatechange = function() { ... }
```

En el interior de esa función, el objeto `this` no se resuelve en el objeto `net.CargadorContenidos`, por lo que no se puede emplear la siguiente instrucción:

```
this.req.onreadystatechange = function() {
  this.onReadyState.call(loader);
}
```

Sin embargo, desde el interior de esa función anónima si es posible acceder a las variables definidas en la función exterior que la engloba. Así, desde el interior de la función anónima sí que es posible acceder a la instancia del objeto `net.CargadorContenidos` que se almacenó anteriormente.

En el código anterior, no es obligatorio emplear la llamada al método `call()`. Se podría haber definido de la siguiente forma:

```
var loader=this;
this.req.onreadystatechange = function() {
  // loader.onReadyState.call(loader);
  loader.onReadyState();
}
```

En el interior de la función `onReadyState`, el objeto `this` se resuelve como `net.ContentLoader`, ya que se trata de un método definido en el prototipo del propio objeto.

Ejercicio 12

La página HTML proporcionada incluye una zona llamada *ticker* en la que se deben mostrar noticias generadas por el servidor. Añadir el código JavaScript necesario para:

1. De forma periódica cada cierto tiempo (por ejemplo cada segundo) se realiza una petición al servidor mediante AJAX y se muestra el contenido de la respuesta en la zona reservada para las noticias.
2. Además del contenido enviado por el servidor, se debe mostrar la hora en la que se ha recibido la respuesta.
3. Cuando se pulse el botón "Detener", la aplicación detiene las peticiones periódicas al servidor. Si se vuelve a pulsar sobre ese botón, se reanudan las peticiones periódicas.
4. Añadir la lógica de los botones "Anterior" y "Siguiente", que detienen las peticiones al servidor y permiten mostrar los contenidos anteriores o posteriores al que se muestra en ese momento.
5. Cuando se recibe una respuesta del servidor, se resalta visualmente la zona llamada *ticker*.
6. Modificar la aplicación para que se reutilice continuamente el mismo objeto `XMLHttpRequest` para hacer las diferentes peticiones.

7.5. Interacción con el servidor

7.5.1. Envío de parámetros con la petición HTTP

Hasta ahora, el objeto `XMLHttpRequest` se ha empleado para realizar peticiones HTTP sencillas. Sin embargo, las posibilidades que ofrece el objeto `XMLHttpRequest` son muy superiores, ya que también permite el envío de parámetros junto con la petición HTTP.

El objeto `XMLHttpRequest` puede enviar parámetros tanto con el método GET como con el método POST de HTTP. En ambos casos, los parámetros se envían como una serie de pares clave/valor concatenados por símbolos &. El siguiente ejemplo muestra una URL que envía parámetros al servidor mediante el método GET:

```
http://localhost/aplicacion?parametro1=valor1&parametro2=valor2&parametro3=valor3
```

La principal diferencia entre ambos métodos es que mediante el método POST los parámetros se envían en el cuerpo de la petición y mediante el método GET los parámetros se concatenan a la URL accedida. El método GET se utiliza cuando se accede a un recurso que depende de la información proporcionada por el usuario. El método POST se utiliza en operaciones que crean, borran o actualizan información.

Técnicamente, el método GET tiene un límite en la cantidad de datos que se pueden enviar. Si se intentan enviar más de 512 bytes mediante el método GET, el servidor devuelve un error con código `414` y mensaje `Request-URI Too Long` (*"La URI de la petición es demasiado larga"*).

Cuando se utiliza un elemento `<form>` de HTML, al pulsar sobre el botón de envío del formulario, se crea automáticamente la cadena de texto que contiene todos los parámetros que se envían al servidor. Sin embargo, el objeto `XMLHttpRequest` no dispone de esa posibilidad y la cadena que contiene los parámetros se debe construir manualmente.

A continuación se incluye un ejemplo del funcionamiento del envío de parámetros al servidor. Se trata de un formulario con tres campos de texto que se validan en el servidor mediante AJAX. El código HTML también incluye un elemento `<div>` vacío que se utiliza para mostrar la respuesta del servidor:

```
<form>
  <label for="fecha_nacimiento">Fecha de nacimiento:</label>
  <input type="text" id="fecha_nacimiento" name="fecha_nacimiento" /><br/>

  <label for="codigo_postal">Codigo postal:</label>
  <input type="text" id="codigo_postal" name="codigo_postal" /><br/>

  <label for="telefono">Telefono:</label>
  <input type="text" id="telefono" name="telefono" /><br/>

  <input type="button" value="Validar datos" />
</form>

<div id="respuesta"></div>
```

El código anterior produce la siguiente página:

Figura 7.3. Formulario de ejemplo

El código JavaScript necesario para realizar la validación de los datos en el servidor se muestra a continuación:

```
var READY_STATE_COMPLETE=4;
var peticion_http = null;

function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function crea_query_string() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
  var telefono = document.getElementById("telefono");

  return "fecha_nacimiento=" + encodeURIComponent(fecha.value) +
         "&codigo_postal=" + encodeURIComponent(cp.value) +
         "&telefono=" + encodeURIComponent(telefono.value) +
         "&nocache=" + Math.random();
}

function valida() {
  peticion_http = inicializa_xhr();
  if(peticion_http) {
```

```
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/validaDatos.php", true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    var query_string = crea_query_string();
    peticion_http.send(query_string);
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      document.getElementById("respuesta").innerHTML = peticion_http.responseText;
    }
  }
}
```

La clave del ejemplo anterior se encuentra en estas dos líneas de código:

```
peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
peticion_http.send(query_string);
```

En primer lugar, si no se establece la cabecera `Content-Type` correcta, el servidor descarta todos los datos enviados mediante el método POST. De esta forma, al programa que se ejecuta en el servidor no le llega ningún parámetro. Así, para enviar parámetros mediante el método POST, es obligatorio incluir la cabecera `Content-Type` mediante la siguiente instrucción:

```
peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

Por otra parte, el método `send()` es el que se encarga de enviar los parámetros al servidor. En todos los ejemplos anteriores se utilizaba la instrucción `send(null)` para indicar que no se envían parámetros al servidor. Sin embargo, en este caso la petición si que va a enviar los parámetros.

Como ya se ha comentado, los parámetros se envían en forma de cadena de texto con las variables y sus valores concatenados mediante el símbolo & (esta cadena normalmente se conoce como *"query string"*). La cadena con los parámetros se construye manualmente, para lo cual se utiliza la función `crea_query_string()`:

```
function crea_query_string() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
  var telefono = document.getElementById("telefono");

  return "fecha_nacimiento=" + encodeURIComponent(fecha.value) +
         "&codigo_postal=" + encodeURIComponent(cp.value) +
         "&telefono=" + encodeURIComponent(telefono.value) +
         "&nocache=" + Math.random();
}
```

La función anterior obtiene el valor de todos los campos del formulario y los concatena junto con el nombre de cada parámetro para formar la cadena de texto que se envía al servidor. El uso de la función `encodeURIComponent()` es imprescindible para evitar problemas con algunos caracteres especiales.

La función `encodeURIComponent()` reemplaza todos los caracteres que no se pueden utilizar de forma directa en las URL por su representación hexadecimal. Las letras, números y los caracteres `- _ . ! ~ * ' ( )` no se modifican, pero todos los demás caracteres se sustituyen por su equivalente hexadecimal.

Las sustituciones más conocidas son las de los espacios en blanco por `%20`, y la del símbolo `&` por `%26`. Sin embargo, como se muestra en el siguiente ejemplo, también se sustituyen todos los acentos y cualquier otro carácter que no se puede incluir directamente en una URL:

```
var cadena = "cadena de texto";
var cadena_segura = encodeURIComponent(cadena);
// cadena_segura = "cadena%20de%20texto";

var cadena = "otra cadena & caracteres problemáticos / : =";
var cadena_segura = encodeURIComponent(cadena);
// cadena_segura =
"otra%20cadena%20%26%20caracteres%20problem%C3%A1ticos%20%2F%20%3A%20%3D";
```

JavaScript incluye una función contraria llamada `decodeURIComponent()` y que realiza la transformación inversa. Además, también existen las funciones `encodeURI()` y `decodeURI()` que codifican/decodifican una URL completa. La principal diferencia entre `encodeURIComponent()` y `encodeURI()` es que esta última no codifica los caracteres `; / ? : @ & = + $ , #`:

```
var cadena = "http://www.ejemplo.com/ruta1/index.php?parametro=valor con ñ y &";
var cadena_segura = encodeURIComponent(cadena);
// cadena_segura =
"http%3A%2F%2Fwww.ejemplo.com%2Fruta1%2Findex.php%3Fparametro%3Dvalor%20con%20%C3%B1%20y%20%26";

var cadena_segura = encodeURI(cadena);
// cadena_segura = "http://www.ejemplo.com/ruta1/
index.php?parametro=valor%20con%20%C3%B1%20y%20";
```

Por último, la función `crea_query_string()` añade al final de la cadena un parámetro llamado `nocache` y que contiene un número aleatorio (creado mediante el método `Math.random()`). Añadir un parámetro aleatorio adicional a las peticiones GET y POST es una de las estrategias más utilizadas para evitar problemas con la caché de los navegadores. Como cada petición varía al menos en el valor de uno de los parámetros, el navegador está obligado siempre a realizar la petición directamente al servidor y no utilizar su cache. A continuación se muestra un ejemplo de la *query string* creada por la función definida:

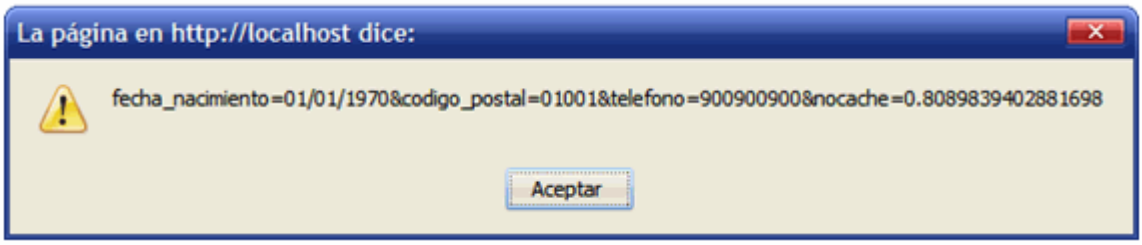

Figura 7.4. Query String creada para el formulario de ejemplo

En este ejemplo sencillo, el servidor simplemente devuelve el resultado de una supuesta validación de los datos enviados mediante AJAX:

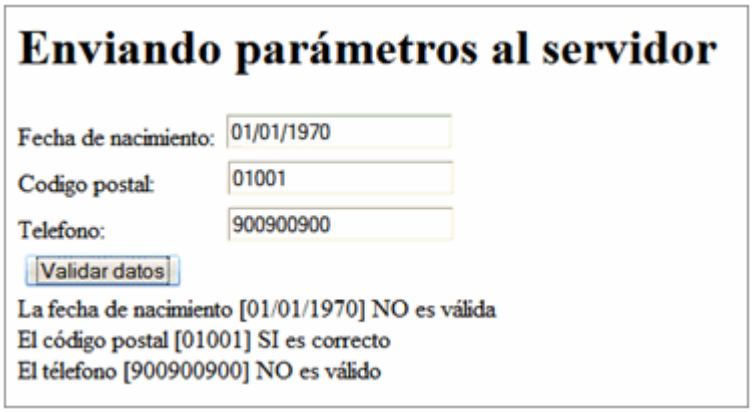

Figura 7.5. Mostrando el resultado devuelto por el servidor

En las aplicaciones reales, las validaciones de datos mediante AJAX sólo se utilizan en el caso de validaciones complejas que no se pueden realizar mediante el uso de código JavaScript básico. En general, las validaciones complejas requieren el uso de bases de datos: comprobar que un nombre de usuario no esté previamente registrado, comprobar que la localidad se corresponde con el código postal indicado, etc.

Ejercicio 13

Un ejemplo de validación compleja es la que consiste en comprobar si un nombre de usuario escogido está libre o ya lo utiliza otro usuario. Como es una validación que requiere el uso de una base de datos muy grande, no se puede realizar en el navegador del cliente. Utilizando las técnicas mostradas anteriormente y la página web que se proporciona:

1. Crear un script que compruebe con AJAX y la ayuda del servidor si el nombre escogido por el usuario está libre o no.
2. El script del servidor se llama `compruebaDisponibilidad.php` y el parámetro que contiene el nombre se llama `login`.
3. La respuesta del servidor es `"si"` o `"no"`, en función de si el nombre de usuario está libre y se puede utilizar o ya ha sido ocupado por otro usuario.
4. A partir de la respuesta del servidor, mostrar un mensaje al usuario indicando el resultado de la comprobación.

7.5.2. Refactorizando la utilidad net.CargadorContenidos

La utilidad diseñada anteriormente para la carga de contenidos y recursos almacenados en el servidor, solamente está preparada para realizar peticiones HTTP sencillas mediante GET. A continuación se refactoriza esa utilidad para que permita las peticiones POST y el envío de parámetros al servidor.

El primer cambio necesario es el de adaptar el constructor para que se puedan especificar los nuevos parámetros:

```
net.CargadorContenidos = function(url, funcion, funcionError, metodo, parametros,
contentType) {
```

Se han añadido tres nuevos parámetros: el método HTTP empleado, los parámetros que se envían al servidor junto con la petición y el valor de la cabecera `content-type`.

A continuación, se sustituye la instrucción `this.req.open('GET', url, true);` por esta otra:

```
this.req.open(metodo, url, true);
```

El siguiente paso es añadir (si así se indica) la cabecera `Content-Type` de la petición:

```
if(contentType) {
  this.req.setRequestHeader("Content-Type", contentType);
}
```

Por último, se sustituye la instrucción `this.req.send(null);` por esta otra:

```
this.req.send(parametros);
```

Así, el código completo de la solución refactorizada es el siguiente:

```
var net = new Object();

net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;

// Constructor
net.CargadorContenidos = function(url, funcion, funcionError, metodo, parametros,
contentType) {
  this.url = url;
  this.req = null;
  this.onload = funcion;
  this.onerror = (funcionError) ? funcionError : this.defaultError;
  this.cargaContenidoXML(url, metodo, parametros, contentType);
}

net.CargadorContenidos.prototype = {
  cargaContenidoXML: function(url, metodo, parametros, contentType) {
    if(window.XMLHttpRequest) {
      this.req = new XMLHttpRequest();
    }
    else if(window.ActiveXObject) {
      this.req = new ActiveXObject("Microsoft.XMLHTTP");
    }

    if(this.req) {
      try {
        var loader = this;
        this.req.onreadystatechange = function() {
          loader.onReadyState.call(loader);
        }
        this.req.open(metodo, url, true);
        if(contentType) {
          this.req.setRequestHeader("Content-Type", contentType);
        }
        this.req.send(parametros);
        } catch(err) {
          this.onerror.call(this);
        }
```

```
    }
  },

  onReadyState: function() {
    var req = this.req;
    var ready = req.readyState;
    if(ready == net.READY_STATE_COMPLETE) {
      var httpStatus = req.status;
      if(httpStatus == 200 || httpStatus == 0) {
        this.onload.call(this);
      }
      else {
        this.onerror.call(this);
      }
    }
  },

  defaultError: function() {
    alert("Se ha producido un error al obtener los datos"
      + "\n\nreadyState:" + this.req.readyState
      + "\nstatus: " + this.req.status
      + "\nheaders: " + this.req.getAllResponseHeaders());
  }
}
```

7.6. Aplicaciones complejas

7.6.1. Envío de parámetros mediante XML

La flexibilidad del objeto `XMLHttpRequest` permite el envío de los parámetros por otros medios alternativos a la tradicional *query string*. De esta forma, si la aplicación del servidor así lo requeire, es posible realizar una petición al servidor enviando los parámetros en formato XML.

A continuación se modifica el ejemplo anterior para enviar los datos del usuario en forma de documento XML. En primer lugar, se modifica la llamada a la función que construye la *query string*:

```
function valida() {
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/validaDatos.php", true);
    var parametros_xml = crea_xml();
    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send(parametros_xml);
  }
}
```

Seguidamente, se crea la función `crea_xml()` que se encarga de construir el documento XML que contiene los parámetros enviados al servidor:

```
function crea_xml() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
```

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

  var xml = "<parametros>";
  xml = xml + "<fecha_nacimiento>" + fecha.value + "<\/fecha_nacimiento>";
  xml = xml + "<codigo_postal>" + cp.value + "<\/codigo_postal>";
  xml = xml + "<telefono>" + telefono.value + "<\/telefono>";
  xml = xml + "<\/parametros>";
  return xml;
}
```

El código de la función anterior emplea el carácter \ en el cierre de todas las etiquetas XML. El motivo es que las etiquetas de cierre XML y HTML (al contrario que las etiquetas de apertura) se interpretan en el mismo lugar en el que se encuentran, por lo que si no se incluyen esos caracteres \ el código no validaría siguiendo el estándar XHTML de forma estricta.

El método `send()` del objeto `XMLHttpRequest` permite el envío de una cadena de texto y de un documento XML. Sin embargo, en el ejemplo anterior se ha optado por una solución intermedia: una cadena de texto que representa un documento XML. El motivo es que no existe a día de hoy un método robusto y que se pueda emplear en la mayoría de navegadores para la creación de documentos XML completos.

7.6.2. Procesando respuestas XML

Además del envío de parámetros en formato XML, el objeto `XMLHttpRequest` también permite la recepción de respuestas de servidor en formato XML. Una vez obtenida la respuesta del servidor mediante la propiedad `petición_http.responseXML`, es posible procesarla empleando los métodos DOM de manejo de documentos XML/HTML.

En este caso, se modifica la respuesta del servidor para que no sea un texto sencillo, sino que la respuesta esté definida mediante un documento XML:

```
<respuesta>
  <mensaje>...</mensaje>
  <parametros>
    <telefono>...</telefono>
    <codigo_postal>...</codigo_postal>
    <fecha_nacimiento>...</fecha_nacimiento>
  </parametros>
</respuesta>
```

La respuesta del servidor incluye un mensaje sobre el éxito o fracaso de la operación de validación de los parámetros y además incluye la lista completa de parámetros enviados al servidor.

La función encargada de procesar la respuesta del servidor se debe modificar por completo para tratar el nuevo tipo de respuesta recibida:

```
function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      var documento_xml = peticion_http.responseXML;
      var root = documento_xml.getElementsByTagName("respuesta")[0];
```

```
      var mensajes = root.getElementsByTagName("mensaje")[0];
      var mensaje = mensajes.firstChild.nodeValue;

      var parametros = root.getElementsByTagName("parametros")[0];

      var telefono =
parametros.getElementsByTagName("telefono")[0].firstChild.nodeValue;
      var fecha_nacimiento =
parametros.getElementsByTagName("fecha_nacimiento")[0].firstChild.nodeValue;
      var codigo_postal =
parametros.getElementsByTagName("codigo_postal")[0].firstChild.nodeValue;

      document.getElementById("respuesta").innerHTML = mensaje + "<br/>" + "Fecha
nacimiento = " + fecha_nacimiento + "<br/>" + "Codigo postal = " + codigo_postal +
"<br/>" + "Telefono = " + telefono;
    }
  }
}
```

El primer cambio importante es el de obtener el contenido de la respuesta del servidor. Hasta ahora, siempre se utilizaba la propiedad `responseText`, que devuelve el texto simple que incluye la respuesta del servidor. Cuando se procesan respuestas en formato XML, se debe utilizar la propiedad `responseXML`.

El valor devuelto por `responseXML` es un documento XML que contiene la respuesta del servidor. Como se trata de un documento XML, es posible utilizar con sus contenidos todas las funciones DOM que se vieron en el capítulo correspondiente a DOM.

Aunque el manejo de repuestas XML es mucho más pesado y requiere el uso de numerosas funciones DOM, su utilización se hace imprescindible para procesar respuestas muy complejas o respuestas recibidas por otros sistemas que exportan sus respuestas internas a un formato estándar XML.

El mecanismo para obtener los datos varía mucho según cada documento XML, pero en general, se trata de obtener el valor almacenado en algunos elementos XML que a su vez pueden ser descendientes de otros elementos. Para obtener el primer elemento que se corresponde con una etiqueta XML, se utiliza la siguiente instrucción:

```
var elemento = root.getElementsByTagName("nombre_etiqueta")[0];
```

En este caso, se busca la primera etiqueta `<nombre_etiqueta>` que se encuentra dentro del elemento `root` (en este caso se trata de la raíz del documento XML). Para ello, se buscan todas las etiquetas `<nombre_etiqueta>` del documento y se obtiene la primera mediante `[0]`, que corresponde al primer elemento del array de elementos.

Una vez obtenido el elemento, para obtener su valor se debe acceder a su primer nodo hijo (que es el nodo de tipo texto que almacena el valor) y obtener la propiedad `nodeValue`, que es la propiedad que guarda el texto correspondiente al valor de la etiqueta:

```
var valor = elemento.firstChild.nodeValue;
```

Normalmente, las dos instrucciones anteriores se unen en una sola instrucción:

```
var tfno = parametros.getElementsByTagName("telefono")[0].firstChild.nodeValue;
```

Ejercicio 14

Normalmente, cuando se valida la disponibilidad de un nombre de usuario, se muestra una lista de valores alternativos en el caso de que el nombre elegido no esté disponible. Modificar el ejercicio de comprobación de disponibilidad de los nombres para que permita mostrar una serie de valores alternativos devueltos por el servidor.

El script del servidor se llama `compruebaDisponibilidadXML.php` y el parámetro que contiene el nombre se llama `login`. La respuesta del servidor es un documento XML con la siguiente estructura:

Si el nombre de usuario está libre:

```
<respuesta>
  <disponible>si</disponible>
</respuesta>
```

Si el nombre de usuario está ocupado:

```
<respuesta>
  <disponible>no</disponible>
  <alternativas>
    <login>...</login>
    <login>...</login>
    ...
    <login>...</login>
  </alternativas>
</respuesta>
```

Los nombres de usuario alternativos se deben mostrar en forma de lista de elementos (`<ul></ul>`).

Modificar la lista anterior para que muestre enlaces para cada uno de los nombres alternativos. Al pinchar sobre el enlace de un nombre alternativo, se copia en el cuadro de texto del login del usuario.

7.6.3. Parámetros y respuestas JSON

Aunque el formato XML está soportado por casi todos los lenguajes de programación, por muchas aplicaciones y es una tecnología madura y probada, en algunas ocasiones es más útil intercambiar información con el servidor en formato JSON.

JSON es un formato mucho más compacto y ligero que XML. Además, es mucho más fácil de procesar en el navegador del usuario. Afortunadamente, cada vez existen más utilidades para procesar y generar el formato JSON en los diferentes lenguajes de programación del servidor (PHP, Java, C#, etc.)

El ejemplo mostrado anteriormente para procesar las respuestas XML del servidor se puede reescribir utilizando respuestas JSON. En este caso, la respuesta que genera el servidor es mucho más concisa:

```
{
mensaje: "...",
```

```
parametros: {telefono: "...", codigo_postal: "...", fecha_nacimiento: "..." }
}
```

Considerando el nuevo formato de la respuesta, es necesario modificar la función que se encarga de procesar la respuesta del servidor:

```
function procesaRespuesta() {
  if(http_request.readyState == READY_STATE_COMPLETE) {
  if(http_request.status == 200) {
    var respuesta_json = http_request.responseText;
    var objeto_json = eval("("+respuesta_json+")");

    var mensaje = objeto_json.mensaje;

    var telefono = objeto_json.parametros.telefono;
    var fecha_nacimiento = objeto_json.parametros.fecha_nacimiento;
    var codigo_postal = objeto_json.parametros.codigo_postal;

    document.getElementById("respuesta").innerHTML = mensaje + "<br>" + "Fecha
nacimiento = " + fecha_nacimiento + "<br>" + "Codigo postal = " + codigo_postal +
"<br>" + "Telefono = " + telefono;
    }
  }
}
```

La respuesta JSON del servidor se obtiene mediante la propiedad `responseText`:

```
var respuesta_json = http_request.responseText;
```

Sin embargo, esta propiedad solamente devuelve la respuesta del servidor en forma de cadena de texto. Para trabajar con el código JSON devuelto, se debe transformar esa cadena de texto en un objeto JSON. La forma más sencilla de realizar esa conversión es mediante la función `eval()`, en la que deben añadirse paréntesis al principio y al final para realizar la evaluación de forma correcta:

```
var objeto_json = eval("("+respuesta_json+")");
```

Una vez realizada la transformación, el objeto JSON ya permite acceder a sus métodos y propiedades mediante la notación de puntos tradicional. Comparado con las respuestas XML, este procedimiento permite acceder a la información devuelta por el servidor de forma mucho más simple:

```
// Con JSON
var fecha_nacimiento = objeto_json.parametros.fecha_nacimiento;

// Con XML
var parametros = root.getElementsByTagName("parametros")[0];
var fecha_nacimiento =
parametros.getElementsByTagName("fecha_nacimiento")[0].firstChild.nodeValue;
```

También es posible el envío de los parámetros en formato JSON. Sin embargo, no es una tarea tan sencilla como la creación de un documento XML. Así, se han diseñado utilidades específicas para la transformación de objetos JavaScript a cadenas de texto que representan el objeto en formato JSON. Esta librería se puede descargar desde el sitio web www.json.org.

Para emplearla, se añade la referencia en el código de la página:

```
<script type="text/javascript" src="json.js"></script>
```

Una vez referenciada la librería, se emplea el método `stringify` para realizar la transformación:

```
var objeto_json = JSON.stringify(objeto);
```

Además de las librerías para JavaScript, están disponibles otras librerías para muchos otros lenguajes de programación habituales. Empleando la librería desarrollada para Java, es posible procesar la petición JSON realizada por un cliente:

```
import org.json.JSONObject;
...
String cadena_json = "{propiedad: valor, codigo_postal: otro_valor}";
JSONObject objeto_json = new JSONObject(cadena_json);
String codigo_postal = objeto_json.getString("codigo_postal");
```

Ejercicio 15

Rehacer el ejercicio 14 para procesar respuestas del servidor en formato JSON. Los cambios producidos son:

1) El script del servidor se llama `compruebaDisponibilidadJSON.php` y el parámetro que contiene el nombre se llama `login`.

2) La respuesta del servidor es un objeto JSON con la siguiente estructura:

El nombre de usuario está libre:

```
{ disponible: "si" }
```

El nombre de usuario está ocupado:

```
{ disponible: "no", alternativas: ["...", "...", ..., "..."] }
```

7.7. Seguridad

La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución. Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado y prácticamente sin recursos ni permisos para realizar tareas básicas.

Las aplicaciones JavaScript no pueden leer ni escribir ningún archivo del sistema en el que se ejecutan. Tampoco pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript. Además, un script sólo puede cerrar aquellas ventanas de navegador que ha abierto ese script.

La restricción del acceso a diferentes dominios es más restrictiva de lo que en principio puede parecer. El problema es que los navegadores emplean un método demasiado simple para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.

Por ejemplo, si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com/scripts/codigo.js, las funciones y métodos incluidos en ese código no pueden acceder a los recursos contenidos en los siguientes archivos:

- http://www.ejemplo.com:8080/scripts/codigo2.js
- https://www.ejemplo.com/scripts/codigo2.js
- http://192.168.0.1/scripts/codigo2.js
- http://scripts.ejemplo.com/codigo2.js

Afortunadamente, existe una forma de solucionar parcialmente el problema del acceso a recursos no originados exactamente en el mismo dominio. La solución se basa en establecer el valor de la propiedad `document.domain`

Así, si el código alojado en http://www.ejemplo.com/scritps/codigo1.js establece la variable `document.domain = "ejemplo.com";` y por otra parte, el código alojado en http://scripts.ejemplo.com/codigo2.js establece la variable `document.domain = "ejemplo.com";` los recursos de ambos códigos pueden interactuar entre sí.

La propiedad `document.domain` se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación. Evidentemente, los navegadores no permiten establecer cualquier valor en esa propiedad, por lo que sólo se puede indicar un valor que corresponda a una parte del subdominio completo donde se encuentra el script.

Capítulo 8. Técnicas básicas con AJAX

8.1. Listas desplegables encadenadas

8.1.1. Contexto

Algunas aplicaciones web disponen de varias listas desplegables encadenadas. En este tipo de listas, cuando se selecciona un elemento de la primera lista desplegable, se cargan en la segunda lista unos valores que dependen del valor seleccionado en la primera lista.

El mayor inconveniente de este tipo de listas se produce cuando existen un gran número de opciones posibles. Si se considera por ejemplo el caso de una tienda, en la primera lista desplegable se pueden mostrar decenas de productos y en la segunda lista se muestran los diferentes modelos de cada producto y sus precios.

Si todos los elementos de las listas desplegables se almacenan mediante arrays de JavaScript en la propia página, los tiempos de carga se pueden disparar y hacerlo completamente inviable.

Por otra parte, se puede optar por recargar completamente la página cada vez que se selecciona un valor diferente en la primera lista desplegable. Sin embargo, recargar la página entera cada vez que se selecciona un valor, aumenta la carga en el servidor y el tiempo de espera del usuario.

Una posible solución intermedia consiste en actualizar las listas desplegables mediante AJAX. Los valores de la primera lista se incluyen en la página web y cuando se selecciona un valor de esta lista, se realiza una consulta al servidor que devuelve los valores que se deben mostrar en la otra lista desplegable.

8.1.2. Solución propuesta

A continuación se muestra el esquema gráfico del funcionamiento de la solución propuesta:

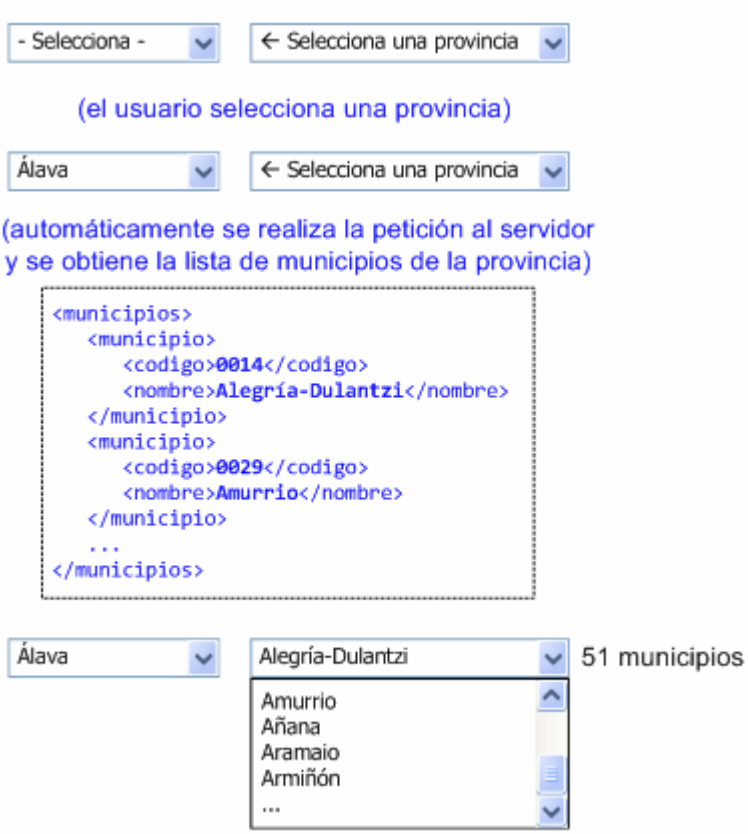

Figura 8.1. Solución propuesta para las listas encadenadas con AJAX

Ejercicio 16

Crear un script que cargue de forma dinámica mediante AJAX la lista de provincias de un país y la lista de los municipios de cada provincia seleccionada.

1) Definir el código HTML de las dos listas desplegables vacías.

2) Cuando se cargue la página, cargar la lista de provincias en la primera lista desplegable. El script del servidor se llama `cargaProvinciasXML.php`. El formato de la respuesta es XML, con la siguiente estructura:

```
<provincias>
  <provincia>
    <codigo>01</codigo>
    <nombre>Álava</nombre>
  </provincia>
  ...
</provincias>
```

Para insertar las opciones en la lista desplegable, se pueden utilizar dos técnicas:

1. Propiedad `innerHTML` de la lista y código HTML de cada etiqueta `<option>`.
2. Crear elementos de tipo opción (`new Option(nombre, valor)`) y añadirlo al array `options[]` de la lista desplegable.

3) Añadir de forma semántica el evento adecuado a la lista de provincias para que cuando se seleccione una provincia, se carguen automáticamente todos sus municipios en la otra lista.

4) Cuando se seleccione una determinada provincia, se carga mediante AJAX la lista completa de municipios en la otra lista desplegable. El script del servidor se llama `cargaMunicipiosXML.php`. El parámetro que se debe enviar al servidor es el código de la provincia y el parámetro se llama `provincia`. El método que espera el servidor es `POST`. El formato de la respuesta es XML, con la siguiente estructura:

```
<municipios>
  <municipio>
    <codigo>0014</codigo>
    <nombre>Alegría-Dulantzi</nombre>
  </municipio>
  ...
</municipios>
```

Ejercicio 17

Modificar el ejercicio anterior para soportar las respuestas del servidor en formato JSON. Los cambios introducidos son los siguientes:

1) El script del servidor utilizado para cargar las provincias se llama `cargaProvinciasJSON.php` y la respuesta del servidor tiene el siguiente formato:

```
{ "01": "Álava/Araba",  "02": "Albacete",  "03": "Alicante/Alacant", ...  }
```

2) El script del servidor utilizado para cargar los municipios se llama `cargaMunicipiosJSON.php` y la respuesta del servidor tiene el siguiente formato:

```
{ "0014": "Alegría-Dulantzi",  "0029": "Amurrio",  ...  }
```

8.2. Teclado virtual

8.2.1. Contexto

Algunas aplicaciones web multi-idioma disponen de la posibilidad de introducir información en muchos idiomas diferentes. El principal problema de estas aplicaciones es que el teclado físico que utiliza el usuario no siempre corresponde al idioma en el que se quieren introducir los contenidos.

La solución habitual de este problema consiste en mostrar un teclado virtual en la pantalla que muestre el valor correcto para cada tecla del idioma seleccionado por el usuario.

8.2.2. Solución propuesta

A continuación se muestra el aspecto gráfico del teclado virtual que se va a construir mediante AJAX:

Figura 8.2. Aspecto final del teclado virtual construido con AJAX

Ejercicio 18

Se propone la construcción de un teclado virtual que permita escribir los contenidos en diversos idiomas y alfabetos. El script hace un uso intensivo de elementos de AJAX como los eventos, DOM, javascript avanzado, JSON y el objeto `XMLHttpRequest`.

Cada uno de los teclados correspondientes a un idioma se carga desde el servidor, para no sobrecargar la aplicación. El teclado de un idioma concreto está formado por varios teclados alternativos o variantes. Así, se encuentra el teclado `normal` para las teclas que se muestran inicialmente, el teclado `caps` con las teclas que se escriben al pulsar sobre la tecla `Bloq. Mayúsculas`, el teclado `shift` que contiene los símbolos que se escriben al pulsar sobre la tecla `Shift` y el teclado `altgr` que contiene los símbolos que se pueden escribir después de pulsar la tecla `Alt Gr`.

Por tanto, cada idioma tiene cuatro teclados diferentes: `normal`, `caps`, `shift` y `altgr`. Inicialmente, el script proporciona el objeto `teclados` con un elemento llamado `es` que contiene los cuatro teclados correspondientes al idioma español.

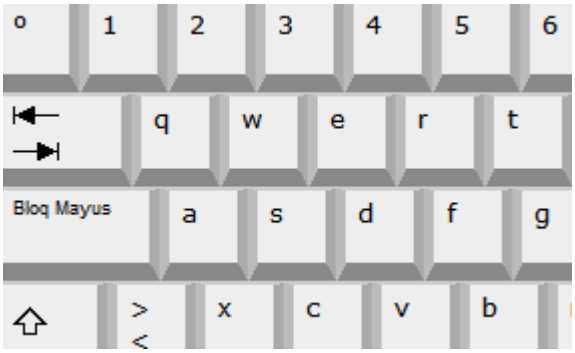

Figura 8.3. Detalle del teclado para el idioma español y la variante "normal"

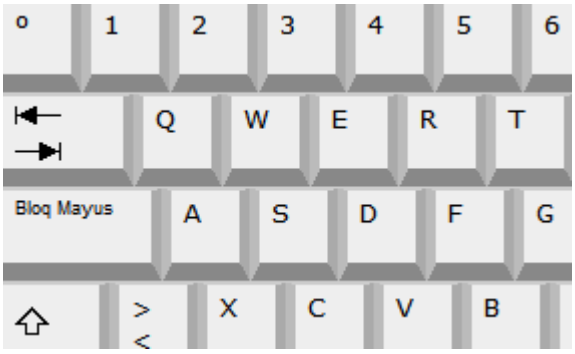

Figura 8.4. Detalle del teclado para el idioma español y la variante "caps"

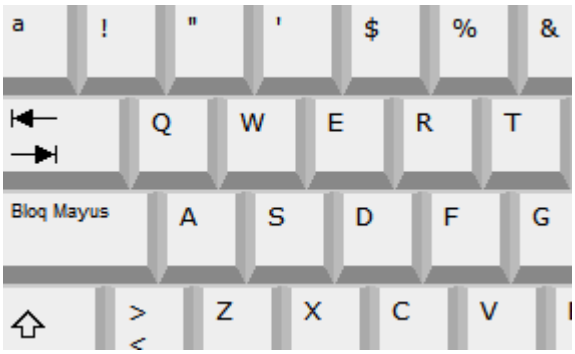

Figura 8.5. Detalle del teclado para el idioma español y la variante "shift"

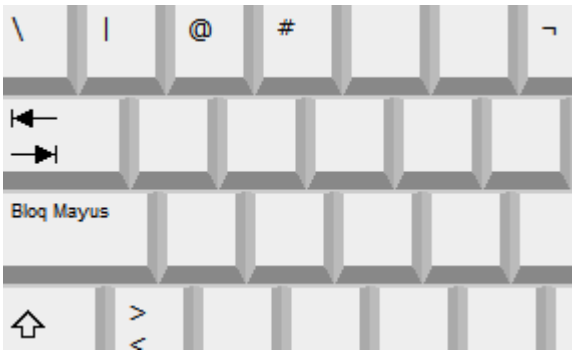

Figura 8.6. Detalle del teclado para el idioma español y la variante "altgr"

1) Crear una función llamada `cargaTeclado()` que muestre en cada tecla virtual el valor de la tecla correspondiente al teclado de un idioma y una variante determinados.

2) Al cargar la página, se debe ejecutar la función `cargaTeclado()`. Previamente, se debe establecer el valor inicial de dos variables globales llamadas `tecladoIdioma` y `tecladoVariante`.

En la misma función `cargaTeclado()`, se debe asignar un evento a cada tecla activa para que al pulsar con el ratón sobre ella, se ejecute la función `pulsaTecla()`.

3) En la función `pulsaTecla()` se obtiene el carácter de la tecla que se ha pulsado correspondiente al teclado activo en ese momento. La tecla se añade a un array global llamado `teclasPulsadas`.

Por último, desde la función `pulsaTecla()` se llama a una función `mostrarContenidos()` que actualiza el texto mostrado en el área reservada para mostrar las teclas pulsadas.

4) Añadir la lógica para tratar las *"teclas especiales"*. Para ello, añadir un evento adecuado que llame a la función `pulsaTeclaEspecial()` cuando el usuario pulse sobre `Enter`, `Tabulador`, `Barra`

`Espaciadora` y `Borrado` (`BackSpace`). En cada caso, se debe añadir al array de teclas pulsadas el carácter correspondiente: `\n`, `\t`, espacio en blanco y el borrado de la última tecla pulsada.

5) Modificar la función `mostrarContenidos()` para que antes de mostrar las teclas que se han pulsado, convierta los caracteres especiales en caracteres correctos para mostrarlos en un elemento HTML: las nuevas líneas (`\n`) se transforman en `<br/>`, los espacios en blanco se transforman en ` ` y el tabulador (`\t`) se transforma en `    `.

6) Cuando se pulsa la tecla `Bloq. Mayús.` o `Shift` o `Alt Gr`, se debe cambiar la variante del teclado actual. Para ello, existen las variantes `caps` para las mayúsculas, `shift` para los símbolos de la tecla `Shift` y `altgr` para los símbolos que aparecen cuando se pulsa la tecla `AltGr`. Añadir a estas teclas especiales el evento adecuado para que se ejecute la función `pulsaTeclaEspecial()` en la que se deben realizar las tareas que correspondan a cada tecla. Además, debería crearse una variable global llamada `estado` que almacene en todo momento el estado de pulsación de estas teclas especiales, ya que el resultado no es el mismo si se pulsa la tecla de mayúsculas estando o no estando pulsada anteriormente.

7) Una vez configurado el script básico del teclado virtual, se van a añadir los elementos relativos a la comunicación con el servidor. En primer lugar, al cargar la página se muestran en una lista desplegable todos los idiomas disponibles. El script del servidor se llama `tecladoVirtual.php` y el envío de parámetros se realiza mediante `POST`. Para cargar los idiomas disponibles, el parámetro que se debe utilizar es `accion` y su valor es `listaIdiomas`. La respuesta del servidor es un objeto JSON con los códigos y nombres de cada idioma, además del código del idioma que se carga al principio:

```
{ idiomas: {es: "Español", de: "Alemán", ru: "Ruso", el: "Griego", ...},
  defecto: "es" }
```

Figura 8.7. Lista desplegable con los idiomas disponibles para el teclado virtual

8) Cuando se cambie el idioma en la lista desplegable, se debe cargar automáticamente el teclado correspondiente a ese idioma. El primer teclado que se muestra al cargar la página es el correspondiente al idioma por defecto indicado por el servidor.

Los teclados de cada idioma con todas sus variantes también se descargan desde el servidor. El script es `tecladoVirtual.php`, en este caso la acción es `cargaTeclado` y se debe pasar otro parámetro llamado `idioma` con el código del idioma cuyo teclado se quiere cargar desde el servidor.

La respuesta del servidor es la siguiente:

```
{ normal: ["&#x00BA;", "&#x0031;", "&#x0032;", "&#x0033;", "&#x0034;", "&#x0035;",
"&#x0036;", ..., "&#x002E;", "&#x002D;"],
 caps: ["&#x00BA;", "&#x0031;", "&#x0032;", "&#x0033;", "&#x0034;", "&#x0035;",
"&#x0036;", ..., "&#x002E;", "&#x002D;"],
shift: ["&#x00AA;", "&#x0021;", "&#x0022;", "&#x0027;", "&#x0024;", "&#x0025;",
"&#x0026;", ..., "&#x003A;", "&#x005F;"],
```

```
altgr: ["&#x005C;", "&#x007C;", "&#x0040;", "&#x0023;",,, "&#x00AC;",,,,,,,,
,,,,,,,,, "&#x005B;","&#x005D;",,,,,,,,,,, "&#x007B;", "&#x007D;",,,,,,,,,,""] }
```

Figura 8.8. Detalle del teclado para el idioma ruso y la variante "normal"

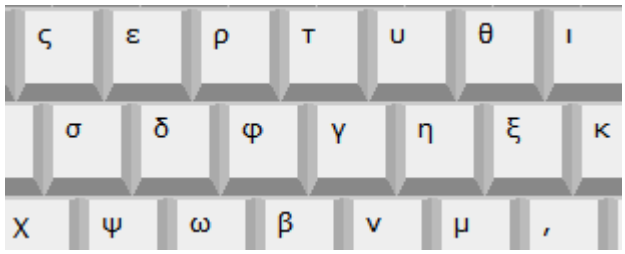

Figura 8.9. Detalle del teclado para el idioma griego y la variante "normal"

Si se utiliza `net.CargadorContenidoscompleto`, puede ser útil emplear el último parámetro que indica si la petición al servidor es síncrona o asíncrona. En este caso, debería ser síncrona, ya que el resto del programa no puede seguir trabajando hasta que se haya cargado completamente el teclado solicitado.

9) Por último, se va a añadir la característica de autoguardado. Para ello, cada 30 segundos se envía el contenido del usuario al servidor para almacenarlo de forma segura. El servidor devuelve el texto que ha guardado y se muestra en la página para comprar el texto del usuario y el texto guardado. El script del servidor se llama `tecladoVirtual.php`, la acción es `guardar` y el parámetro `contenido` es el que indica el contenido creado por el usuario.

10) Se puede añadir una pequeña mejora visual al teclado virtual: existe una clase de CSS llamada `pulsada` y que se puede utilizar para resaltar de forma clara la tecla que se ha pulsado. Utilizar esa clase para iluminar durante un breve espacio de tiempo la tecla pulsada en cada momento.

11) Otras posibles mejoras: funcionamiento del teclado numérico, funcionamiento de los acentos, manejo de los LED del teclado, etc.

8.3. Autocompletar

8.3.1. Contexto

Algunas veces, se presenta al usuario un cuadro de texto en el que tiene que introducir un valor que pertenece a un grupo muy grande de datos. Algunos casos habituales son: una dirección de correo electrónico que pertenezca a la libreta de direcciones del usuario, el nombre válido de un municipio de un país, el nombre de un empleado de una empresa grande, etc.

En la mayoría de casos, utilizar una lista desplegable que muestre todos los valores es completamente inviable, ya que pueden existir miles de posibles valores. Por otra parte, un

cuadro de texto simple resulta de poca utilidad para el usuario. La solución consiste en combinar un cuadro de texto y una lista desplegable mediante AJAX.

Al usuario se le presenta un cuadro de texto simple en el que puede introducir la información. A medida que el usuario escribe en el cuadro de texto, la aplicación solicita al servidor aquellos términos que estén relacionados con lo escrito por el usuario. Cuando la aplicación recibe la respuesta del servidor, la muestra al usuario a modo de ayuda para autocompletar la información.

8.3.2. Solución propuesta

A continuación se muestra la interacción del sistema de autocompletado propuesto:

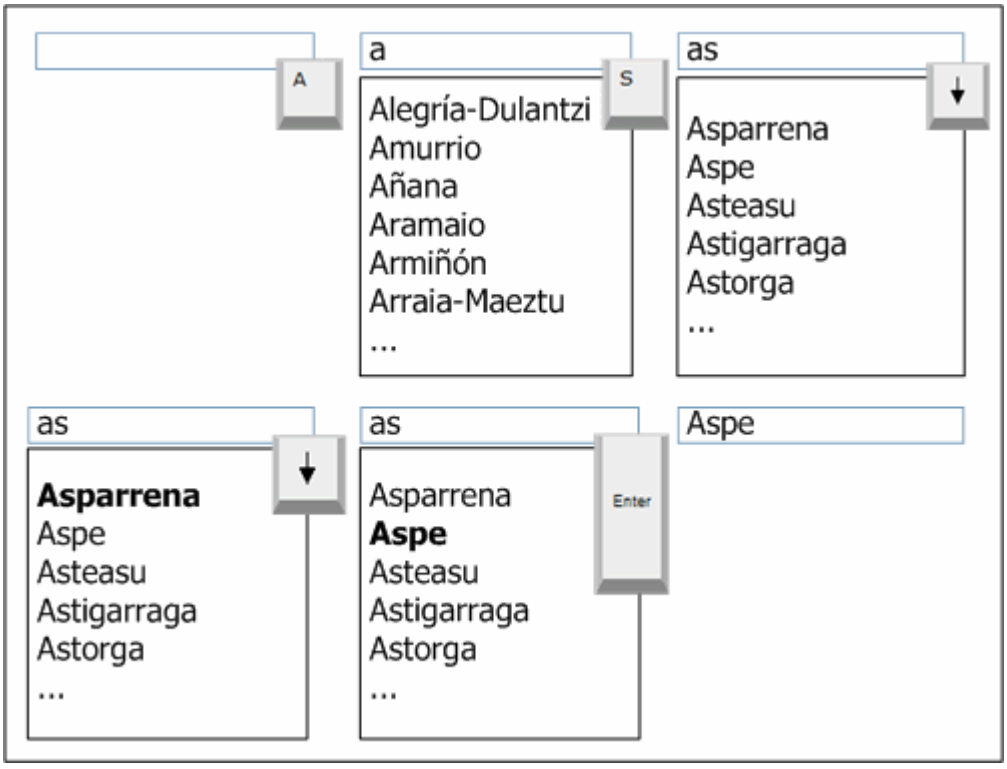

Figura 8.10. Interacción completa del usuario con el sistema de autocompletado

Ejercicio 19

A partir del formulario proporcionado, añadir la opción de autocompletar el nombre del municipio que está escribiendo el usuario. El esquema del funcionamiento propuesto es el siguiente:

1) Al cargar la página, se debe crear un elemento HTML de tipo `<div>` en el que se van a mostrar las sugerencias enviadas por el servidor.

Además, se debe establecer el evento de teclado adecuado en el cuadro de texto y también se debe posicionar el cursor en ese cuadro de texto para poder escribir en el directamente (Pista: `focus()`).

2) Cuando se pulse una tecla sobre el cuadro de texto, se debe ejecutar la función `autocompleta()`. Desde esta función, se debe llamar a la función responsable de obtener la lista de municipios del servidor. El script se llama `autocompletaMunicipios.php`, el parámetro que se envía mediante `POST`, se llama `municipio` y debe contener la cadena de texto escrita por el usuario.

El servidor responde con un array en formato JSON con la lista de municipios cuyo nombre comienza por el texto enviado. Ejemplo de respuesta del servidor:

```
[ "Alegría-Dulantzi", "Amurrio", "Añana", "Aramaio", "Armiñón", ... ]
```

3) Una vez obtenido el array de sugerencias, se debe mostrar en forma de lista de elementos (etiqueta `<ul>` de HTML). Para transformar el array en la lista `<ul>`, modificar el `prototype` del objeto `Array` y añadir una función específica que realice la transformación.

4) Modificar la función `autocompleta()` para tener en consideración 3 teclas especiales: las flechas superior e inferior y la tecla `Enter`. (Pista: propiedad `keyCode`). Cuando se utilizan las flechas del teclado hacia arriba y hacia abajo, se van seleccionando los elementos de la lista. Cuando se pulsa el `Enter`, se selecciona el elemento copiando su valor al cuadro de texto y ocultando la lista de sugerencias. (Pista: variable global `elementoSeleccionado`)

5) Para mejorar el rendimiento de la aplicación, añadir una cache para las sugerencias. Cada vez que se recibe una lista de sugerencias del servidor, se almacena en un objeto que relaciona el texto que ha introducido el usuario y la respuesta del servidor. Ejemplo:

```
{
"a": ["Ababuj", "Abades", "Abadía", "Abadín", "Abadiño", "Abáigar", "Abajas",
"Abaltzisketa", "Abánades", "Abanilla", "Abanto y Ciérvana-Abanto Zierbena", "Abanto",
"Abarán", "Abarca de Campos", "Abárzuza", "Abaurregaina/Abaurrea Alta", "Abaurrepea/
Abaurrea Baja", "Abegondo", "Abejar", "Abejuela", "Abella de la Conca"],

"al": ["Alacón", "Aladrén", "Alaejos", "Alagón", "Alaior", "Alájar", "Alajeró",
"Alameda de Gardón (La)", "Alameda de la Sagra", "Alameda del Valle", "Alameda",
"Alamedilla (La)", "Alamedilla", "Alamillo", "Alaminos", "Alamús (Els)", "Alange",
"Alanís", "Alaquàs", "Alar del Rey", "Alaraz"],
...
}
```

De esta forma, antes de realizar una petición al servidor, se comprueba si ya está disponible una lista de sugerencias para ese texto. Además, cuando se realiza una consulta al servidor, la respuesta se almacena en la cache para su posible reutilización futura.

6) Mostrar un mensaje adecuado cuando el servidor devuelva un array vacío por no haber sugerencias para el texto introducido por el usuario.

Capítulo 9. Técnicas avanzadas con AJAX

9.1. Monitorización de servidores remotos

9.1.1. Contexto

Las aplicaciones JavaScript ejecutadas en los navegadores tienen unas restricciones muy estrictas en cuanto a su seguridad. Además de no poder acceder a recursos locales como archivos y directorios, los scripts solamente pueden realizar conexiones de red con el mismo dominio al que pertenece la aplicación JavaScript.

9.1.2. Solución propuesta

Ejercicio 20

Se propone la realización de una consola básica de monitorización de equipos de red. Los servidores que se van a monitorizar pertenecen a dominios conocidos de Internet y por tanto, externos a la aplicación JavaScript.

En otras palabras, se trata de *"hacer un ping"* a través de la red mediante AJAX para comprobar los equipos que se quieren monitorizar.

1) Cuando se cargue la página, se debe construir *"el mapa de red"* que muestra todos los servidores que se van a monitorizar. Para construir este mapa, se proporciona la página web básica y un objeto llamado `nodos`, que contiene los datos correspondientes a los equipos que se quieren monitorizar.

Para mostrar cada nodo que se van a monitorizar, se crean dinámicamente elementos de tipo `<div>` que muestran el nombre de cada nodo, su URL y una zona de datos en la que se mostrará más adelante cierta información del nodo. A continuación se muestra un ejemplo del posible código XHTML que se puede utilizar:

```
<div id="nodo0" class="normal">
  <strong>Nombre</strong>
  <br/>
  URL
  <span id="datos0"></span>
</div>
```

Figura 9.1. Aspecto inicial de la consola de monitorización mostrando todos los servidores remotos que se van a monitorizar

2) La consola de monitorización se controla mediante dos botones. De esta forma, se deben establecer los eventos adecuados en los botones `Monitorizar` y `Detener`.

3) Al pulsar sobre el botón `Monitorizar` debe comenzar la ejecución periódica (por ejemplo cada 10 segundos) de la función que realiza las conexiones con los servidores y procesa su respuesta. Al pulsar sobre el botón `Detener`, se debe impedir la ejecución periódica de esa función.

4) La función que se ejecuta de forma periódica (por ejemplo cada 10 segundos) debe realizar un *"ping"* a cada uno de los equipos. Hacer un *ping* consiste en intentar establecer una conexión con el servidor utilizando el método `HEAD` de HTTP (en vez de los tradicionales métodos `GET` o `POST`).

Si se intenta acceder a la página principal de un sitio web utilizando el método `HEAD` y la petición es correcta, el servidor devuelve las cabeceras HTTP. De esta forma, para comprobar si el servidor ha respondido de forma correcta, se puede intentar obtener el valor de alguna cabecera enviada por el servidor, como por ejemplo `Date`.

Para realizar las peticiones, se puede utilizar la utilidad `net.CargadorContenidosCompleto` y establecer de forma adecuada las funciones que se encargan de procesar las respuestas correctas y las respuestas erróneas.

5) La función que procesa las respuestas correctamente recibidas, debe obtener el valor de alguna cabecera HTTP como por ejemplo `Date`. Si la respuesta es correcta, mostrar en la zona `<span id="datos"></span>` de cada nodo el valor de la cabecera `Server`, que indica el tipo de servidor web que utiliza el nodo remoto.

6) También es importante mostrar visualmente la monitorización que se está realizando, por lo que se van a modificar las propiedades CSS de cada nodo para indicar el estado en el que se encuentra:

- Cuando se está realizando una consulta al servidor, la propiedad `border` de CSS debe ser igual a `3px solid #000000`.
- Cuando la respuesta recibida es correcta, la clase CSS del `<div>` es `"on"` y el valor de la propiedad `border` es `3px solid #00FF00`.
- Cuando la respuesta es errónea, la clase CSS del `<div>` es `"off"` y el valor de la propiedad `border` es `3px solid #FF0000`.

Figura 9.2. La consola de monitorización muestra visualmente el estado de cada nodo y también muestra parte de la información devuelta por el servidor

Al ejecutar la aplicación en Internet Explorer, se muestra un aviso al usuario sobre si desea dar permiso a la aplicación para realizar conexiones externas:

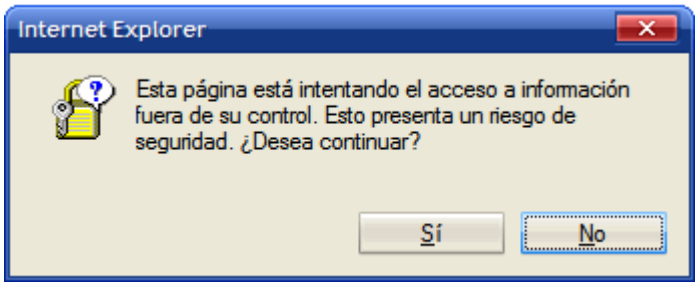

Figura 9.3. Mensaje de aviso que muestra Internet Explorer al intentar establecer conexiones de red con servidores remotos

Para poder ejecutar correctamente la aplicación en los navegadores de tipo Firefox, se debe firmar digitalmente el script y solicitar permiso de forma expresa al usuario. Para más información, se puede consultar el siguiente recurso: http://www.mozilla.org/projects/security/components/signed-scripts.html

9.2. Lector RSS

9.2.1. Contexto

La sindicación de contenidos mediante protocolos como RSS y Atom ha permitido que los sitios web puedan compartir fácilmente sus noticias, artículos, entradas de blogs y muchos otros contenidos digitales. Aunque RSS es un conjunto de formatos y protocolos, en su forma última es un archivo XML que incluye los contenidos que quiere compartir el servidor web. La sencillez del formato y la posibilidad de procesar archivos XML con JavaScript hacen que RSS sea ideal para desarrollar aplicaciones AJAX que traten con sus contenidos.

9.2.2. Solución propuesta

Ejercicio 21

A partir de la página web que se proporciona, completar el script para realizar un lector avanzado de canales RSS.

1) Al cargar la página, se debe añadir un evento en el botón `Mostrar RSS`.

2) El script debe mostrar información sobre las operaciones que realiza. Para ello, cada acción que se ejecute debe mostrar un mensaje en el elemento `<div id="info"></div>` de la página.

3) Al pulsar sobre el botón `Mostrar RSS`, se ejecuta la función `cargarRSS()` y se informa al usuario de que se está buscando el canal RSS.

4) La primera tarea que se ejecuta en la función `cargarRSS()` es la de obtener la URL específica del canal RSS a partir de la URL original de la página HTML. Esta tarea no es sencilla, por lo que es recomendable utilizar el script del servidor llamado `descubreRss.php()`, que acepta un parámetro llamado `url` pasado mediante el método GET y devuelve la URL correspondiente al canal RSS.

5) Una vez obtenida la URL del canal RSS, se descarga su contenido. Para obtener los contenidos del canal RSS, es conveniente utilizar un proxy que permita saltarse la restricción de JavaScript para realizar conexiones de red remotas. El script se llama `proxy.php` y admite dos parámetros GET

llamados `url` (que es la URL que se quiere descargar) y `ct` (`Content-Type` del contenido que se está descargando, que es muy importante cuando se quieren recibir contenidos de tipo XML).

6) Después de descargar el contenido del canal RSS, se debe procesar su contenido para obtener cada uno de sus elementos y almacenarlos en un array global llamado `canal`.

El formato XML resumido de RSS es el siguiente:

```
<?xml version="1.0"?>
<rss version="2.0">
<channel>
  <title>Ejemplo de canal 2.0</title>
  <link>http://www.ejemplo_no_real.com</link>
  <description>Se trata de un ejemplo de canal RSS 2.0, sencillo pero
completo</description>
  <item>
    <title>El primer elemento</title>
    <link>http://www.ejemplo_no_real.com/elementos/001.html</link>
    <description>Esta es la descripción del primer elemento.</description>
    <pubDate>Sun, 18 Feb 2007 15:04:27 GMT</pubDate>
  </item>
  <item>
    <title>El segundo elemento</title>
    <link> http://www.ejemplo_no_real.com/elementos/002.html </link>
    <description> Esta es la descripción del primer elemento.</description>
    <pubDate>Sun, 18 Feb 2007 15:04:27 GMT</pubDate>
  </item>
  ...
  <item>
    <title>El elemento N</title>
    <link> http://www.ejemplo_no_real.com/elementos/00n.html </link>
    <description> Esta es la descripción del elemento N.</description>
    <pubDate>Sun, 18 Feb 2007 15:04:27 GMT</pubDate>
  </item>
</channel>
</rss>
```

El formato del array `elementos` puede ser cualquiera que permita almacenar para cada elemento su titular, descripción, enlace y fecha de publicación.

7) Una vez descargado y procesado el canal RSS, mostrar sus elementos tal y como se indica en la siguiente imagen:

Figura 9.4. Aspecto final del lector RSS construido con AJAX

Al pinchar en cada titular de los que se muestran en la parte izquierda, se carga su contenido completo en la zona central.

9.3. Google Maps

9.3.1. Contexto

Google Maps fue una de las primeras aplicaciones basadas en AJAX de uso masivo por parte de los usuarios. Su gran éxito ha provocado que todos sus rivales hayan copiado el funcionamiento de sus mapas.

Además, Google ofrece de forma gratuita una API con la que poder desarrollar aplicaciones a medida basadas en los mapas de Google, integrar los mapas en otras aplicaciones e incluso hacer *"mash-up"* o mezclas de Google Maps y otras aplicaciones web que también disponen de una API pública.

9.3.2. Solución propuesta

Antes de utilizar la API de los mapas de Google, es necesario obtener una clave personal y única para cada sitio web donde se quiere utilizar. El uso de la API es gratuito para cualquier aplicación que pueda ser accedida libremente por los usuarios. La clave de la API se puede obtener desde: http://www.google.com/apis/maps/

Para usos comerciales de la API también existen servicios de pago que requieren el uso de otras claves.

Las claves se solicitan por cada ruta del servidor. De esta forma, si se solicita una clave para http://www.misitio.com/ruta1, cualquier aplicación o página que se encuentre bajo esa ruta del servidor podrá hacer uso de la API de los mapas con esa clave.

Si no se dispone de un sitio web público, es posible indicar como servidor el valor `http://localhost` para poder hacer pruebas en un servidor local. Para solicitar la clave, es necesario disponer de una cuenta de usuario de Google (se puede utilizar por ejemplo la cuenta de Gmail). La clave generada depende por tanto del dominio de la aplicación y de la cuenta de usuario.

Las claves de la API de Google Maps consisten en una cadena de texto muy larga con un aspecto similar al siguiente: `ABQIAAAA30JtKUU8se-7KKPRGSfCMBT2yXp_ZAY8_ufC3CFXhHIE1NvwkxRZNdns2BwZvEY-V68DvlyUYwi1-Q`.

Una vez obtenida la clave, cualquier página que quiera hacer uso de la API debe enlazar el siguiente archivo de JavaScript:

```
<script src="http://maps.google.com/
maps?file=api&amp;v=2&amp;hl=es&amp;key=ABQIAAAA30JtKUU8se-7KKPRGSfCMBT2yXp_ZAY8_ufC3CFXhHIE1NvwkxRZ
type="text/javascript"></script>
```

Los parámetros necesarios son `file`, que indica el tipo de archivo que se quiere cargar (en este caso la API), `v`, que indica la versión de la API (en este caso `2`), `hl`, que permite indicar el idioma en el que se muestran los mapas (si no se indica este parámetro, los mapas se muestran en inglés) y el parámetro `key`, que contiene la clave que se acaba de obtener.

Una vez obtenida la clave, es muy sencillo crear el primer mapa de Google:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejemplo de uso de Google Maps</title>
<script src="http://maps.google.com/
maps?file=api&amp;v=2&amp;hl=es&amp;key=ABQIAAAA30JtKUU8se-7KKPRGSfCMBT2yXp_ZAY8_ufC3CFXhHIE1NvwkxRZ
type="text/javascript"></script>
<script type="text/javascript">
function load() {
  if (GBrowserIsCompatible()) {
    var latitud = 48.858729;
    var longitud = 2.352448;
    var zoom = 15;
    var mapa = new GMap2(document.getElementById("mapa"));
    mapa.setCenter(new GLatLng(latitud, longitud), zoom);
  }
}
</script>
</head>

<body onload="load()" onunload="GUnload()">
  <div id="mapa" style="width: 500px; height: 400px"></div>
</body>
</html>
```

El anterior código HTML y JavaScript genera el siguiente mapa:

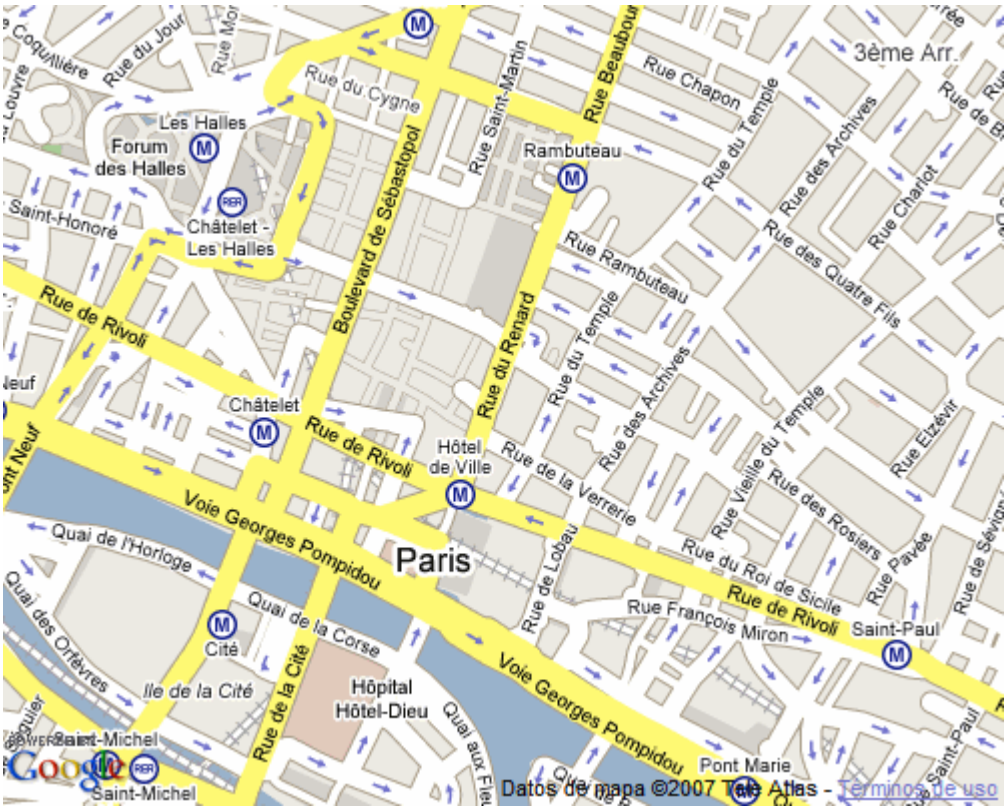

Figura 9.5. Mapa sencillo creado con la API de Google Maps y las coordenadas de longitud y latitud indicadas

En primer lugar, se define un elemento (normalmente un `<div>`) en el que se mostrará el mapa. El tamaño del mapa será igual al tamaño de su elemento contenedor, a no ser que se indique explícitamente otro tamaño:

```
<div id="mapa" style="width: 500px; height: 400px"></div>
```

Una vez definido el contenedor del mapa, se establecen los eventos necesarios en la página que lo contiene:

```
<body onload="load()" onunload="GUnload()">
```

Los ejemplos que se pueden consultar en la documentación de Google utilizan esa asignación directa de eventos en la etiqueta `<body>`. Evidentemente, los eventos también se podrían asignar de forma *semántica* mediante el siguiente código:

```
window.onload = load;
window.onunload = GUnload;
```

La función `load()` es la que se ejecuta al cargar la página y la que establece las opciones con las que se crea el mapa. En primer lugar, se ejecuta la función `GBrowserIsCompatible()`, que indica si el navegador actual del usuario es compatible con el uso de los mapas de Google. En caso contrario, no se ejecutan las instrucciones y no se muestra el mapa:

```
if (GBrowserIsCompatible()) {
  ...
}
```

Para crear un nuevo mapa, se utiliza la clase `GMap2`, que representa un mapa en una página. Las páginas pueden contener más de un mapa, pero cada uno de ellos hace referencia a una instancia diferente de la clase `GMap2`. El único argumento obligatorio para crear el mapa es la referencia al elemento que contendrá el mapa (obtenida mediante su `id`):

```
var mapa = new GMap2(document.getElementById("mapa"));
```

Una vez instanciada la clase `GMap2`, el mapa ya ha sido creado. Sin embargo, el mapa no se muestra correctamente hasta que se indique en que punto geográfico está centrado. El método `setCenter()` permite centrar un mapa y opcionalmente, indicar el nivel de zoom y el tipo de mapa que se muestra:

```
var latitud = 42.845007;
var longitud = -2.673;
var zoom = 15;
mapa.setCenter(new GLatLng(latitud, longitud), zoom);
```

El punto geográfico en el que está centrado el mapa se indica mediante un objeto de tipo `GLatLng()` que toma dos parámetros: el primero es el valor de la latitud del punto geográfico y el otro parámetro indica la longitud de esa posición geográfica. De forma opcional se puede indicar el nivel de zoom del mapa.

La latitud puede tomar un valor entre `+90` (`90` grados al norte del ecuador) y `-90` (`90` grados al sur del ecuador), la longitud puede tomar un valor entre `+180` (`180` grados al este de Greenwitch) y `-180` (`180` grados al oeste de Greenwitch). El nivel de zoom puede variar entre `1` (en el que se ve la Tierra entera) y `18` (en el que se ven los detalles de cada calle). No obstante, se debe tener en cuenta que no todos los puntos geográficos disponen de todos los niveles de zoom.

Después de crear un mapa básico, es muy sencillo añadir los controles para aumentar o disminuir el nivel de zoom y el control que indica el tipo de mapa que se muestra:

```
mapa.addControl(new GSmallMapControl());
mapa.addControl(new GMapTypeControl());
```

Ahora, el mapa permite variar el nivel de zoom y mostrar otro tipo de mapa, como se muestra en la siguiente imagen:

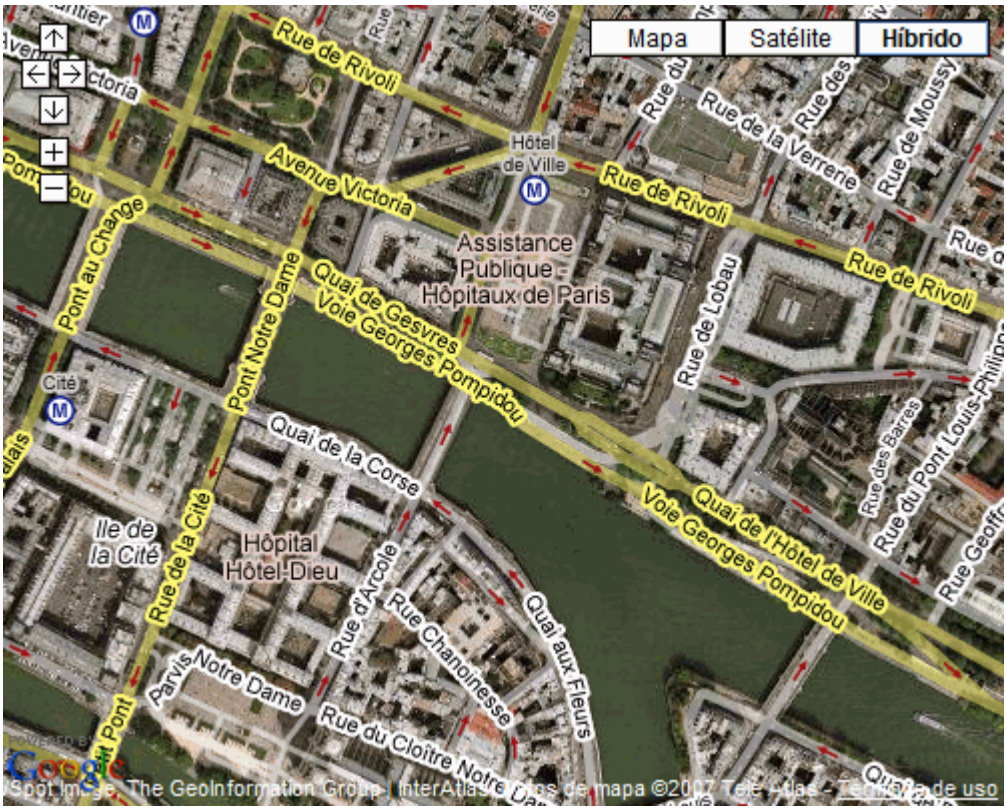

Figura 9.6. Mapa sencillo creado con la API de Google Maps y que permite variar su nivel de zoom y el tipo de mapa que se muestra

Google Maps incluye una documentación muy extensa sobre el uso de la API y todos los métodos disponibles. La documentación inicial con ejemplos básicos se puede consultar en: http://www.google.com/apis/maps/documentation/

La referencia completa de clases, propiedades y métodos de la API está disponible en: http://www.google.com/apis/maps/documentation/reference.html

Los mapas de Google permiten controlar un gran número de eventos, pudiendo asociar funciones o ejecutar directamente código JavaScript cada vez que se produce un evento. El siguiente ejemplo muestra un mensaje de tipo `alert()` con las coordenadas del centro del mapa cada vez que el usuario suelta el mapa después de haberlo movido:

```
GEvent.addListener(mapa, "moveend", function() {
  var centro = mapa.getCenter();
  alert(centro.toString());
});
```

Además del evento `moveend` existen muchos otros, como por ejemplo `move` que permite ejecutar cualquier función de forma repetida a medida que el usuario mueve el mapa. Los mapas también permiten colocar marcadores para mostrar una posición. El siguiente ejemplo hace uso del evento `click` sobre los mapas para mostrar marcadores:

```
GEvent.addListener(mapa, "click", function(marcador, punto) {
  mapa.addOverlay(new GMarker(punto));
});
```

Ahora, cada vez que se pulsa sobre un punto del mapa, se añade un nuevo marcador en ese punto:

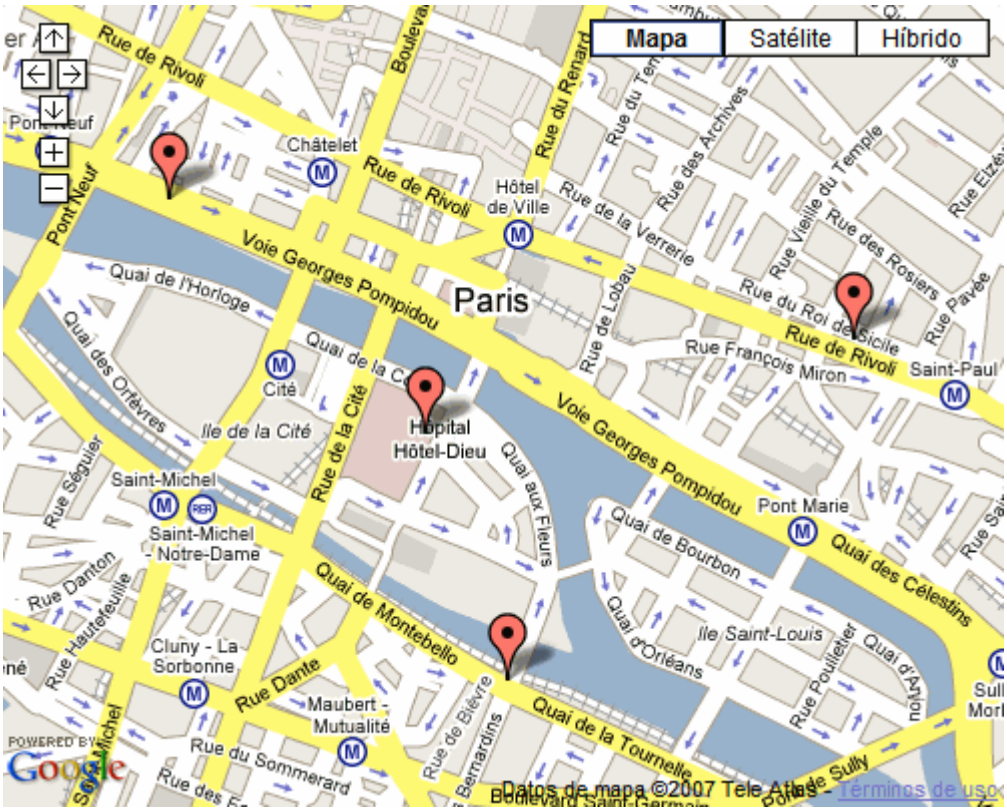

Figura 9.7. Los eventos de la API de los mapas de Google permiten añadir marcadores de posición cada vez que se pincha en un punto del mapa

Modificando ligeramente el ejemplo anterior, es posible borrar un marcador si se pulsa sobre el:

```
GEvent.addListener(mapa, "click", function(marcador, punto) {
  if(marcador) {
    mapa.removeOverlay(marcador);
  } else {
    mapa.addOverlay(new GMarker(punto));
  }
});
```

Modificando de nuevo el código anterior, es posible añadir eventos a los marcadores para que cada vez que se pinche en uno de ellos se muestre un mensaje con sus coordenadas geográficas:

```
GEvent.addListener(mapa, "click", function(marcador, punto) {
  var nuevoMarcador = new GMarker(punto);
  GEvent.addListener(nuevoMarcador, "click", function() {
    this.openInfoWindowHtml("Lat: " + this.getPoint().lat() + "<br/>Lon: " +
this.getPoint().lng());
  });
  mapa.addOverlay(nuevoMarcador);
});
```

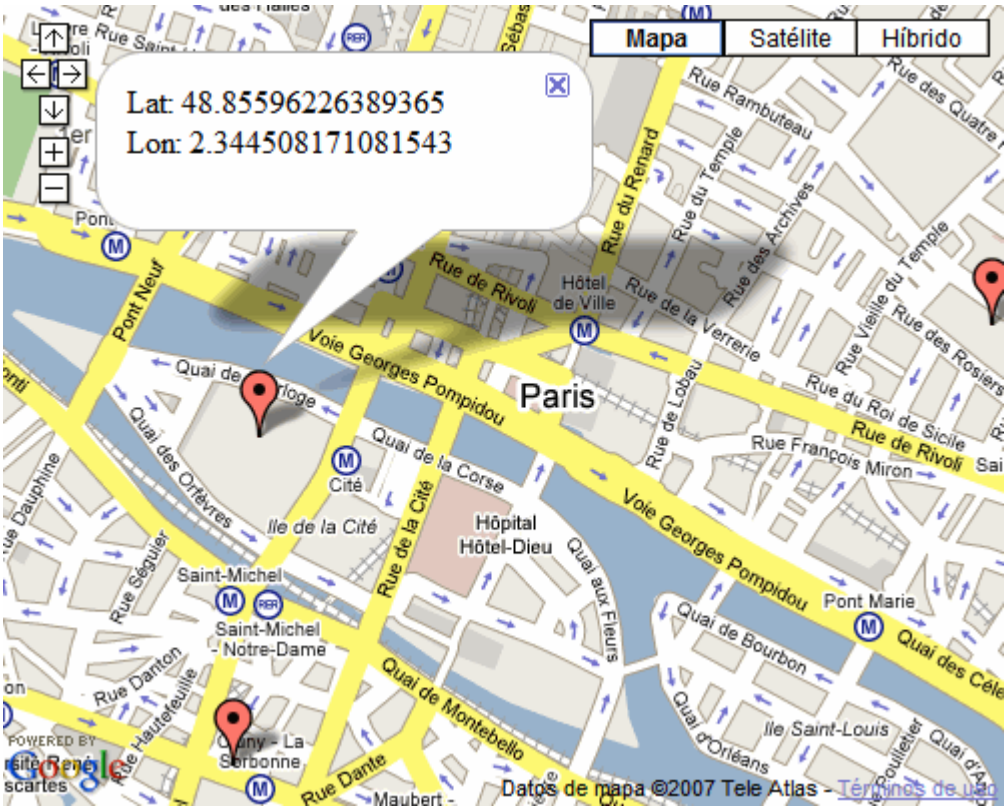

Figura 9.8. Al pulsar sobre cada marcador de posición en el mapa, se muestra un mensaje informando sobre la longitud y latitud de la posición del marcador

A continuación se muestra la explicación detallada del código anterior. En primer lugar se establece un nuevo evento cada vez que se pincha sobre el mapa:

```
GEvent.addListener(mapa, "click", function(marcador, punto) {
```

Cada vez que se pincha, se crea un nuevo marcador con las coordenadas de la posición en la que se ha pinchado:

```
var nuevoMarcador = new GMarker(punto);
```

Antes de mostrar el marcador en el mapa, se añade un evento al propio marcador, que se ejecutará cada vez que se pinche sobre el:

```
GEvent.addListener(nuevoMarcador, "click", function() {
```

El método `openInfoWindowHtml()` permite mostrar una pequeña ventana con un mensaje que puede contener código HTML. Dentro de la función manejadora del evento, la variable `this` se resuelve en el propio marcador, por lo que se puede hacer uso de ella para obtener sus coordenadas y para mostrar el mensaje:

```
this.openInfoWindowHtml("Lat: " + this.getPoint().lat() + "<br/>Lon: " +
this.getPoint().lng());
```

Por último, se añade el marcador al mapa:

```
mapa.addOverlay(nuevoMarcador);
```

Ejercicio 22

Partiendo del mapa básico que se acaba de crear, se pide:

1) En la misma página se debe incluir otro mapa que muestre las antípodas del punto geográfico inicial del primer mapa. Este segundo mapa no debe mostrar ningún control de zoom ni de tipo de mapa.

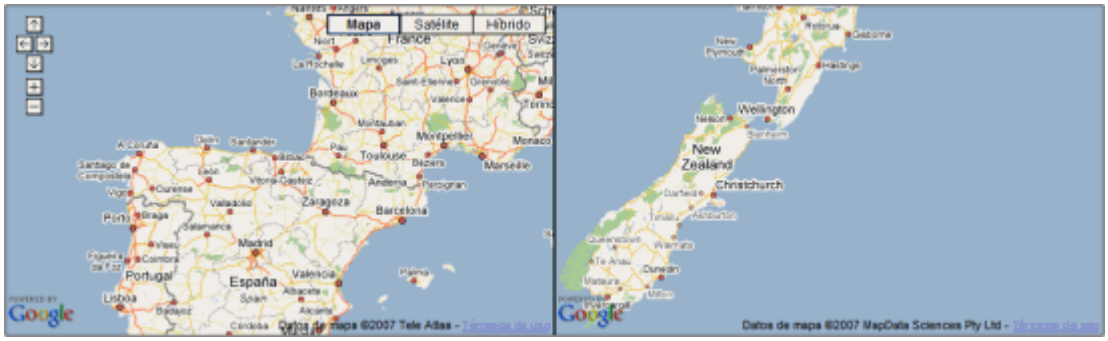

Figura 9.9. Aspecto del mapa principal y del mapa secundario que muestra el lugar geográfico definido como "antípodas" del lugar mostrado en el mapa principal

2) Al mover el primer mapa, el segundo mapa debe mostrar en todo momento las antípodas de ese lugar. Además, el zoom y el tipo de mapa también debe estar sincronizado, de forma que el segundo mapa muestre en todo momento el mismo nivel de zoom y el mismo tipo de mapa que el primero.

3) Cuando se pinche en el primer mapa, se muestra un marcador con su longitud y su latitud. Además, automáticamente se crea un marcador en el segundo mapa indicando el lugar exacto que corresponde a su antípoda.

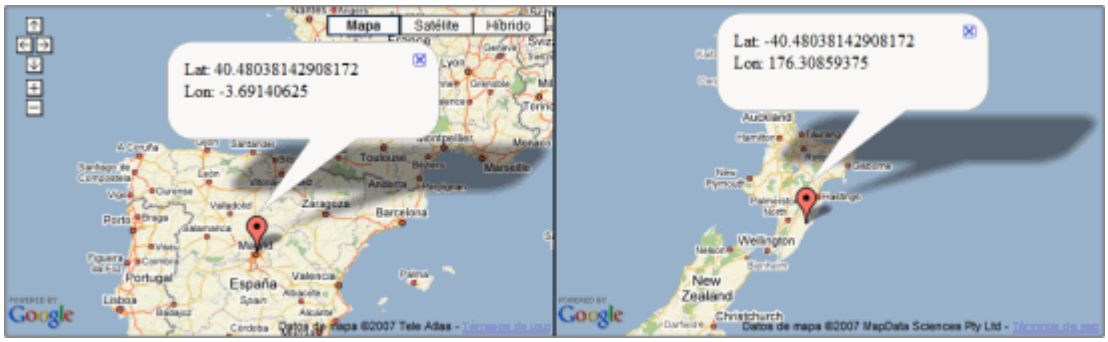

Figura 9.10. Cuando se crea un marcador en el mapa principal, automáticamente debe crearse otro marcador de posición en el lugar geográfico correspondiente a su antípoda

Ejercicio 23

La disponibilidad de una API pública, sencilla, gratuita y muy potente permite integrar los mapas de Google con cualquier otra aplicación abierta que genere información geoespacial.

Un ejemplo típico de esta mezcla de aplicaciones es la información metereológica. Con la ayuda de los mapas de Google, es posible mostrar en cada punto del mapa la previsión metereológica en forma de marcador de posición con un icono personalizado, tal y como se muestra en la siguiente imagen:

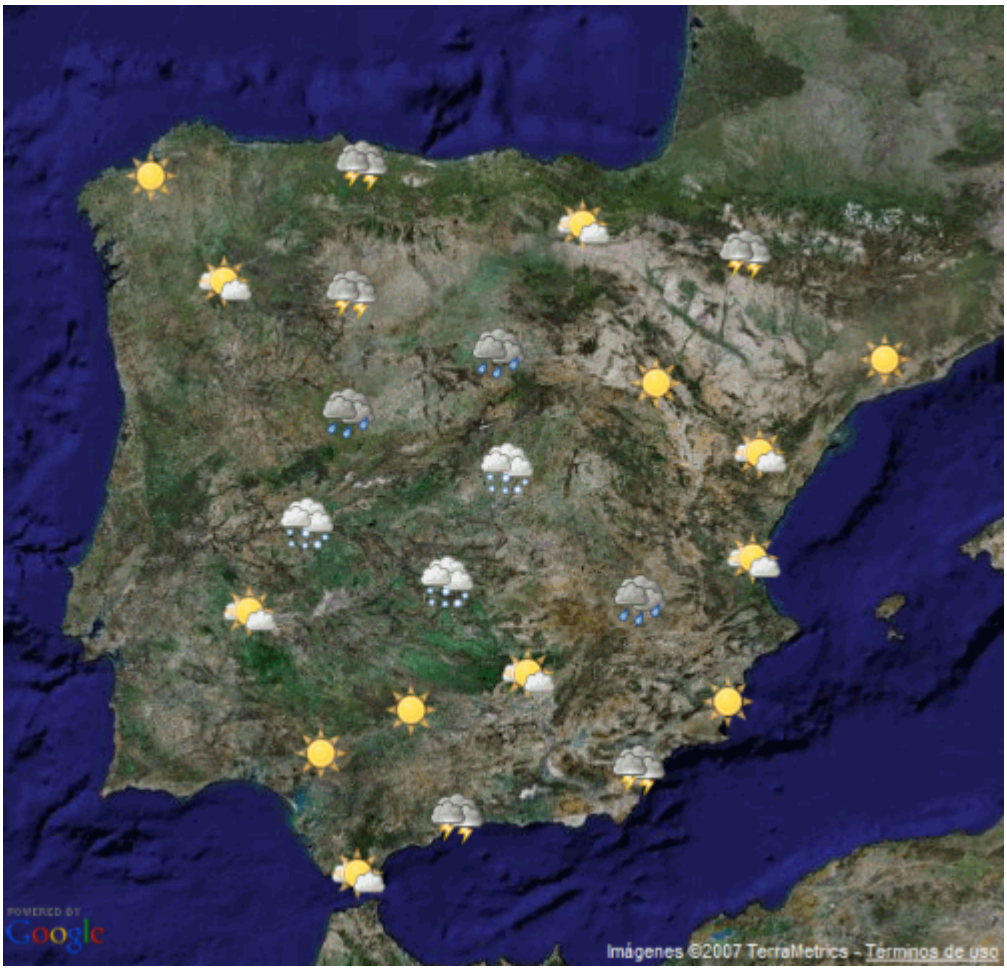

Figura 9.11. Mapa de previsión metereológica construido con los mapas de Google y que utiliza iconos personalizados para mostrar cada marcador de posición

1) Mostrar un mapa de Google de tamaño `600x600` píxel, centrado en `[40.41558722527384, -3.6968994140625]`, que muestre por defecto el mapa de tipo satélite y que no incluya ningún tipo de control.

2) Cada cierto tiempo se debe mostrar la previsión meteorológica descargada desde el servidor. El script del servidor se denomina `previsionMeteorologica.php` y no es necesario pasar ningún parámetro. El servidor devuelve un array en formato JSON con la lista de puntos geográficos junto con su previsión meteorológica:

```
[
{ latlon: [42.779275360242, -2.63671875], prediccion: "tormentas" },
{ latlon: [43.245202722034, -8.32763671875], prediccion: "nieve" },
{ latlon: [42.228517356209, -7.36083984375], prediccion: "lluvia" },
...
{ latlon: [41.54147766679, -3.75732421875], prediccion: "nublado" },
]
```

3) La información meteorológica de cada punto se muestra mediante un marcador personalizado. Pistas: la clase `GMarkerManager` permite gestionar conjuntos grandes de marcadores. La clase `GMarker` permite definir un icono personalizado para el marcador. La clase `GIcon` permite crear nuevos iconos listos para los marcadores. En la carpeta `imagenes` se proporcionan iconos para cada

una de las condiciones meteorológicas: `lluvia.png`, `nieve.png`, `nublado.png`, etc. Las imágenes utilizadas en este ejemplo, pertenecen al excelente conjunto de iconos del proyecto Tango (http://tango.freedesktop.org/Tango_Desktop_Project) .

Capítulo 10. Frameworks y librerías

Las aplicaciones web son cada vez más complejas, ya que incluyen efectos e interacciones que hasta hace poco tiempo eran exclusivas de las aplicaciones de escritorio. Al mismo tiempo, la programación de estas aplicaciones avanzadas se complica por varios motivos.

En primer lugar, las aplicaciones comerciales deben funcionar correctamente e igual de bien en al menos cinco navegadores diferentes: Internet Explorer 6 y 7, Firefox, Opera y Safari. En segundo lugar, el tiempo disponible para el desarrollo de cada nueva característica se reduce cada vez más por la necesidad continua de incluir novedades en las aplicaciones antes de que las incluya la competencia.

Por todo lo anterior, han surgido librerías y *frameworks* específicos para el desarrollo de aplicaciones con JavaScript. Utilizando estas librerías, se reduce el tiempo de desarrollo y se tiene la seguridad de que las aplicaciones funcionan igual de bien en cualquiera de los navegadores más populares.

Aunque se han publicado decenas de librerías y frameworks, a continuación se van a mostrar las dos más populares: Prototype (junto con script.aculo.us) y jQuery.

10.1. El framework Prototype

Prototype (http://www.prototypejs.org/) es un framework que facilita el desarrollo de aplicaciones web con JavaScript y AJAX. Su autor original es **Sam Stephenson**, aunque las últimas versiones incorporan código e ideas de muchos otros programadores. A pesar de que incluye decenas de utilidades, la librería es compacta y está programada de forma muy eficiente.

Prototype se ha convertido en poco tiempo en una referencia básica de AJAX y es la base de muchos otros frameworks y librerías relacionadas como script.aculo.us (http://script.aculo.us/) . Las primeras versiones de Prototype no incluían ningún tipo de documentación, lo que dificultaba su uso y provocaba que la mayoría de usuarios desconocieran su verdadero potencial.

Afortunadamente, las versiones más recientes del framework disponen de una completa documentación de todas las funciones y métodos que componen su API. La documentación incluye la definición completa de cada método, sus atributos y varios ejemplos de uso: http://www.prototypejs.org/api

10.1.1. Funciones y métodos básicos

La primera función que se estudia cuando se está aprendiendo Prototype es tan útil como impronunciable: `$()`. La *"función dólar"* es un atajo mejorado de la función `document.getElementById()`.

Si se le pasa una cadena de texto con el identificador de un elemento, obtiene ese elemento. La función admite uno o más parámetros: si se le pasa un parámetro, devuelve un objeto; si se le pasan varios parámetros, devuelve un array simple con todos los objetos.

```
// Con JavaScript
var elemento = document.getElementById('primero');

// Con Prototype
var elemento = $('primero');

// Con JavaScript
var elemento1 = document.getElementById('primero');
var elemento2 = document.getElementById('segundo');

// Con Prototype
var elementos = $('primero', 'segundo');
```

Otra de las funciones más útiles de Prototype es `$F()`, que es similar a la anterior función, pero se utiliza para obtener directamente el valor de los campos de formulario:

```
<input id="municipio" />

// Con JavaScript
document.getElementById("municipio").value

// Con Prototype
$F("municipio")


<select id="municipio">
  <option>...</option>
</select>

// Con JavaScript
document.getElementById("municipio").options[document.getElementById("municipio").selectedIndex].val

// Con Prototype
$F("municipio")
```

Una de las funciones más espectaculares de Prototype y que no tiene equivalente en JavaScript es `$$()`, que permite seleccionar elementos de la página utilizando selectores de CSS.

```
<div id="principal">
  <p>Primer párrafo</p>
  <p>Segundo párrafo</p>
</div>
<p>Tercer párrafo</p>

var todosParrafos = $$('p');
var parrafosInteriores = $$('#principal p');
```

Prototype incluye una función muy útil llamada `$A()`, para convertir en array *"cualquier cosa que se parezca a un array"*. Algunas funciones de JavaScript, como por ejemplo `getElementsByTagName()` devuelven objetos de tipo `NodeList` o `HTMLCollection`, que no son arrays, aunque pueden recorrerse como tales.

```
<select id="lista">
  <option value="1">Primer valor</option>
  <option value="2">Segundo valor</option>
```

```
  <option value="3">Tercer valor</option>
</select>

// 'lista_nodos' es una variable de tipo NodeList
var lista_nodos = $('lista').getElementsByTagName('option');

// 'nodos' es una variable de tipo array
var nodos = $A(lista_nodos);
// nodos = [objeto_html_opcion1, objeto_html_opcion2, objeto_html_opcion3]
```

Una función similar a `$A()` es `$H()`, que crea *arrays asociativos* (también llamados *"hash"*) a partir del argumento que se le pasa:

```
var usuarios = { usuario1: "password1",
                 usuario2: "password2",
                 usuario3: "password3" };

var hash_usuarios = $H(usuarios);

var logins = hash_usuarios.keys();
// logins = ["usuario1", "usuario2", "usuario3"]

var passwords = hash_usuarios.values();
// passwords = ["password1", "password2", "password3"]

var queryString = hash_usuarios.toQueryString();
// queryString = "usuario1=password1&usuario2=password2&usuario3=password3"

var debug = hash_usuarios.inspect();
// #<Hash:{'usuario1':'password1', 'usuario2':'password2','usuario3':'password3'}>
```

Por último, Prototype incluye la función `$R()` para crear rangos de valores. El rango de valores se crea desde el valor del primer argumento hasta el valor del segundo argumento. El tercer argumento de la función indica si se excluye o no el último valor (por defecto, el tercer argumento vale `false`, que indica que sí se incluye el último valor).

```
var rango = $R(0, 100, false);
// rango = [0, 1, 2, 3, ..., 100]

var rango = $R(0, 100);
// rango = [0, 1, 2, 3, ..., 100]

var rango = $R(0, 100, true);
// rango = [0, 1, 2, 3, ..., 99]

var rango2 = $R(100, 0);
// rango2 = [100]

var rango = $R(0, 100);
var incluido = rango.include(4);
// incluido = true

var rango = $R(0, 100);
var incluido = rango.include(400);
// incluido = false
```

Los rangos que se pueden crear van mucho más allá de simples sucesiones numéricas. La *"inteligencia"* de la función `$R()` permite crear rangos tan avanzados como los siguientes:

```
var rango = $R('a', 'k');
// rango = ['a', 'b', 'c', ..., 'k']

var rango = $R('aa', 'ak');
// rango = ['aa', 'ab', 'ac', ..., 'ak']

var rango = $R('a_a', 'a_k');
// rango = ['a_a', 'a_b', 'a_c', ..., 'a_k']
```

Por último, una función muy útil que se puede utilizar con cadenas de texto, objetos y arrays de cualquier tipo es `inspect()`. Esta función devuelve una cadena de texto que es una representación de los contenidos del objeto. Se trata de una utilidad imprescindible cuando se están depurando las aplicaciones, ya que permite visualizar el contenido de variables complejas.

10.1.2. Funciones para cadenas de texto

El framework Prototype extiende las cadenas de texto de JavaScript añadiéndoles una serie de funciones que pueden resultar muy útiles:

`stripTags()`: Elimina todas las etiquetas HTML y XML de la cadena de texto

`stripScripts()`: Elimina todos los bloques de tipo `<script></script>` de la cadena de texto

`escapeHTML()`: transforma todos los caracteres *problemáticos* en HTML a su respectiva entidad HTML (< se transforma en `&lt;`, & se transforma en `&amp;`, etc.)

```
var cadena = "<p>Prueba de texto & caracteres HTML</p>".escapeHTML();
// cadena = "&lt;p&gt;Prueba de texto &amp; caracteres HTML&lt;/p&gt;"
```

`unescapeHTML()`: función inversa de `escapeHTML()`

```
var cadena = "<p>Prueba de texto & caracteres HTML</p>".unescapeHTML();
// cadena = "Prueba de texto & caracteres HTML"

var cadena = "<p>&ntilde; &aacute; &iquest; &amp;</p>".unescapeHTML();
// cadena = "ñ á ¿ &"
```

`extractScripts()`: devuelve un array con todos los bloques `<script></script>` de la cadena de texto

`evalScripts()`: ejecuta cada uno de los bloques `<script></script>` de la cadena de texto

`toQueryParams()`: convierte una cadena de texto de tipo *query string* en un array asociativo (*hash*) de pares parámetro/valor

```
var cadena = "parametro1=valor1&parametro2=valor2&parametro3=valor3";

var parametros = cadena.toQueryParams();
// $H(parametros).inspect() = #<Hash:{'parametro1':'valor1', 'parametro2':'valor2','
parametro3':'valor3'}>
```

`toArray()`: convierte la cadena de texto en un array que contiene sus letras

`camelize()`: convierte una cadena de texto separada por guiones en una cadena con notación de tipo *CamelCase*

```
var cadena = "el-nombre-de-la-variable".camelize();
// cadena = "elNombreDeLaVariable"
```

`underscore()`: función inversa de `camelize()`, ya que convierte una cadena de texto escrita con notación *CamelCase* en una cadena de texto con las palabras separadas por guiones bajos

```
var cadena = "elNombreDeLaVariable".underscore();
// cadena = "el_nombre_de_la_variable"
```

`dasherize()`: modifica los guiones bajos (_) de una cadena de texto por guiones medios (-)

```
var cadena = "el_nombre_de_la_variable".dasherize();
// cadena = "el-nombre-de-la-variable"
```

Combinando `camelize()`, `underscore()` y `dasherize()`, se puede obtener el nombre DOM de cada propiedad CSS y viceversa:

```
var cadena = 'borderTopStyle'.underscore().dasherize();
// cadena = 'border-top-style'

var cadena = 'border-top-style'.camelize();
// cadena = 'borderTopStyle'
```

10.1.3. Funciones para elementos

Prototype define funciones muy útiles para manipular los elementos incluidos en las páginas HTML. Cualquier elemento obtenido mediante la función `$()` puede hacer uso de las siguientes funciones:

`Element.visible()`: devuelve `true/false` si el elemento es visible/oculto (devuelve `true` para los campos tipo `hidden`)

`Element.show()` y `Element.hide()`: muestra y oculta el elemento indicado

`Element.toggle()`: si el elemento es visible, lo oculta. Si es elemento está oculto, lo muestra

`Element.scrollTo()`: baja o sube el *scroll* de la página hasta la posición del elemento indicado

`Element.getStyle()` y `Element.setStyle()`: obtiene/establece el valor del estilo CSS del elemento (el estilo completo, no la propiedad `className`)

`Element.classNames()`, `Element.hasClassName()`, `Element.addClassName()`, `Element.removeClassName()`: obtiene los `class` del elemento, devuelve `true/false` si incluye un determinado `class`, añade un `class` al elemento y elimina el `class` al elemento respectivamente

Todas las funciones anteriores se pueden invocar de dos formas diferentes:

```
// Las dos instrucciones son equivalentes
Element.toggle('principal');
$('principal').toggle()
```

10.1.4. Funciones para formularios

Prototype incluye muchas utilidades relacionadas con los formularios y sus elementos. A continuación se muestran las más útiles para los campos de un formulario:

`Field.clear()`: borra el valor de cada campo que se le pasa (admite uno o más parámetros)

`Field.present()`: devuelve `true` si los campos que se le indican han sido rellenados por parte del usuario, es decir, si contienen valores no vacíos (admite uno o más parámetros)

`Field.focus()`: establece el foco del formulario en el campo que se le indica

`Field.select()`: selecciona el valor del campo (solo para los campos en los que se pueda seleccionar su texto)

`Field.activate()`: combina en una única función los métodos `focus()` y `select()`

A las funciones anteriores se les debe pasar como parámetro una cadena de texto con el identificador del elemento o el propio elemento (obtenido por ejemplo con `$()`). Las funciones mostradas se pueden invocar de tres formas diferentes:

```
// Las 3 instrucciones son equivalentes
Form.Element.focus('id_elemento');
Field.focus('id_elemento')
$('id_elemento').focus()
```

Además de las funciones específicas para los campos de los formularios, Prototype también define utilidades para los propios formularios completos. Todas las funciones requieren un solo parámetro: el identificador o el objeto del formulario.

`Form.serialize()`: devuelve una cadena de texto de tipo *"query string"* con el valor de todos los campos del formulario (`"campo1=valor1&campo2=valor2&campo3=valor3"`)

`Form.findFirstElement()`: devuelve el primer campo activo del formulario

`Form.getElements()`: devuelve un array con todos los campos del formulario (incluyendo los elementos ocultos)

`Form.getInputs()`: devuelve un array con todos los elementos de tipo `<input>` del formulario. Admite otros dos parámetros para filtrar los resultados. El segundo parámetro indica el tipo de `<input>` que se quiere obtener y el tercer parámetro indica el nombre del elemento `<input>`.

`Form.disable()`: deshabilita todo el formulario deshabilitando todos sus campos

`Form.enable()`: habilita el formulario completo habilitando todos sus campos

`Form.focusFirstElement()`: pone el foco del formulario en el primer campo que sea visible y esté habilitado

`Form.reset()`: resetea el formulario completo, ya que es equivalente al método `reset()` de JavaScript

10.1.5. Funciones para arrays

Las utilidades añadidas a los arrays de JavaScript es otro de los puntos fuertes de Prototype:

`clear()`: vacía de contenidos el array y lo devuelve

`compact()`: devuelve el array sin elementos `null` o `undefined`

`first()`: devuelve el primer elemento del array

`flatten()`: convierte cualquier array que se le pase en un array unidimensional. Se realiza un proceso recursivo que va *"aplanando"* el array:

```
var array_original = ["1", "2", 3,
                      ["a", "b", "c",
                        ["A", "B", "C"]
                      ]
                     ];

var array_plano = array_original.flatten();
// array_plano = ["1", "2", 3, "a", "b", "c", "A", "B", "C"]
```

`indexOf(value)`: devuelve el valor de la posición del elemento en el array o `-1` si no lo encuentra

```
var array = ["1", "2", 3, ["a", "b", "c", ["A", "B", "C"] ] ];
array.indexOf(3);   // 2
array.indexOf("C"); // -1
```

`last()`: devuelve el último elemento del array

`reverse()`: devuelve el array original en sentido inverso:

```
var array = ["1", "2", 3, ["a", "b", "c", ["A", "B", "C"] ] ];
array.reverse();
// array = [["a", "b", "c", ["A", "B", "C"]], 3, "2", "1"]
```

`shift()`: devuelve el primer elemento del array y lo extrae del array (el array se modifica y su longitud disminuye en 1 elemento)

`without()`: devuelve el array del que se han eliminado todos los elementos que coinciden con los argumentos que se pasan a la función. Permite filtrar los contenidos de un array

```
var array = [12, 15, 16, 3, 40].without(16, 12)
// array = [15, 3, 40]
```

10.1.6. Funciones para objetos enumerables

Algunos tipos de objetos en JavaScript se comportan como *colecciones* de valores, también llamadas *"enumeraciones"* de valores. Prototype define varias utilidades para este tipo de objetos a través de `Enumerable`, que es uno de los pilares básicos del framework y una de las formas más sencillas de mejorar la productividad cuando se desarrollan aplicaciones JavaScript.

Algunos de los objetos obtenidos mediante las funciones de Prototype, ya incorporan todos los métodos de `Enumerable`. Sin embargo, si se quieren añadir estos métodos a un objeto propio, se pueden utilizar las utilidades de Prototype para crear objetos y extenderlos:

```
var miObjeto = Class.create();
Object.extend(miObjeto.prototype, Enumerable);
```

Gracias a `Enumerable`, se pueden recorrer los arrays de forma mucho más eficiente:

```
// Array original
var vocales = ["a", "e", "i", "o", "u"];

// Recorrer el array con JavaScript
for(var i=0; i<vocales.length; i++) {
  alert("Vocal " + vocales[i] + " está en la posición " + i);
}

// Recorrer el array con Prototype:
vocales.each(function(elemento, indice) {
  alert("Vocal " + elemento + " está en la posición " + indice);
});
```

El método `select()`, que es un alias del método `findAll()`, permite filtrar los contenidos de un array:

```
var numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
resultado = numeros.findAll(function(elemento) { return elemento > 5; });
// resultado = [6, 7, 8, 9, 10]
```

Otro método útil es `pluck()`, que permite obtener el valor de una misma propiedad para todos los elementos de la colección:

```
var numLetras = ['hola', 'mundo', 'que', 'bien', 'funciona',
'Prototype'].pluck('length');
// numLetras = [4, 5, 3, 4, 8, 9]
```

`Enumerable` incluye decenas de utilidades y métodos, algunos tan curiosos como `partition()` que permite dividir una colección en dos grupos: el de los elementos de tipo `true` y el de los elementos de tipo `false` (valores como `null`, `undefined`, etc.)

```
var valores = ['nombreElemento', 12, null, 2, true, , false].partition();
// valores = [['nombreElemento', 12, 2, true], [null, undefined, false]]
```

El método `partition()` permite asignar una función propia para decidir si un elemento se considera `true` o `false`. En el siguiente ejemplo, se divide un array con letras en dos grupos, el de las vocales y el de las consonantes:

```
var letras = $R('a', 'k').partition(function(n) {
  return ['a', 'e', 'i', 'o', 'u'].include(n);
})
// letras = [['a', 'e', 'i'], ['b', 'c', 'd', 'f', 'g', 'h', 'j', 'k']]
```

El método `invoke()` permite ejecutar una función para todos los elementos de la colección:

```
var palabras = ['hola', 'mundo', 'con', 'Prototype'].invoke('toUpperCase');
// palabras = ['HOLA', 'MUNDO', 'CON', 'PROTOTYPE']
```

La documentación de Enumerable (http://www.prototypejs.org/api/enumerable) incluye la definición y ejemplos de muchos otros métodos útiles como `inGroupsOf()` (agrupa elementos en subconjuntos del mismo tamaño), `sortBy()` (permite definir la ordenación de los elementos

mediante una función propia), `zip()` (asocia uno a uno los elementos de dos colecciones), `collect()` (permite transformar los elementos de la colección con una función propia), etc.

10.1.7. Otras funciones útiles

`Try.these()`: permite probar varias funciones de forma consecutiva hasta que una de ellas funcione. Es muy útil para las aplicaciones que deben funcionar correctamente en varios navegadores diferentes.

El propio código fuente de Prototype utiliza `Try.these()` para obtener el objeto encargado de realizar las peticiones AJAX:

```
var Ajax = {
  getTransport: function() {
    return Try.these(
      function() {return new XMLHttpRequest()},
      function() {return new ActiveXObject('Msxml2.XMLHTTP')},
      function() {return new ActiveXObject('Microsoft.XMLHTTP')}
    ) || false;
  },

  activeRequestCount: 0
}
```

`Class.create()`: permite crear clases de una forma elegante y sencilla:

```
MiClase = Class.create();
MiClase.prototype = {
  initialize: function(a, b) {
    this.a = a;
    this.b = b;
  }
}

var miClase = new MiClase("primer_valor", "segundo_valor");
```

`Object.extend()`: se emplea para añadir o sobrescribir las propiedades de un objeto en otro objeto. Se puede considerar como una forma primitiva y muy básica de herencia entre clases. En la llamada a la función, el primer objeto es el destino en el que se copian las propiedades del segundo objeto pasado como parámetro:

```
Object.extend(objetoDestino, objetoOrigen);
```

Esta función es muy útil para que las aplicaciones definan una serie de opciones por defecto y puedan tener en cuenta las opciones establecidas por cada usuario:

```
// El array "opciones" guarda las opciones por defecto de la aplicación
var opciones = {campo: "usuario", orden: "ASC"};

// El usuario establece sus propias opciones
var opciones_usuario = {orden: "DESC", tipoBusqueda: "libre"};

// Se mezclan los dos arrays de opciones, dando prioridad
// a las opciones establecidas por los usuarios
```

```
Object.extend(opciones, opciones_usuario);

// Ahora, opciones.orden = "DESC"
```

El código fuente de Prototype utiliza `Object.extend()` continuamente para añadir propiedades y métodos útiles a los objetos de JavaScript. El código que se muestra a continuación añade cinco métodos al objeto `Number` original de JavaScript:

```
Object.extend(Number.prototype, {
  toColorPart: function() {
    return this.toPaddedString(2, 16);
  },

  succ: function() {
    return this + 1;
  },

  times: function(iterator) {
    $R(0, this, true).each(iterator);
    return this;
  },

  toPaddedString: function(length, radix) {
    var string = this.toString(radix || 10);
    return '0'.times(length - string.length) + string;
  },

  toJSON: function() {
    return isFinite(this) ? this.toString() : 'null';
  }
});
```

10.1.8. Funciones para AJAX

Además de todas las funciones y utilidades para la programación *tradicional* de JavaScript, Prototype incluye numerosas funciones relacionadas con el desarrollo de aplicaciones AJAX. Los métodos que componen este módulo son `Ajax.Request()`, `Ajax.Updater()`, `Ajax.PeriodicalUpdater()` y `Ajax.Responders()`.

`Ajax.Request()` es el método principal de este módulo. Se utiliza para realizar peticiones AJAX y procesar sus resultados. Su sintaxis es:

```
new Ajax.Request(url, opciones);
```

El primer parámetro (`url`) es la URL que solicita la petición AJAX y el segundo parámetro (`opciones`) es opcional y se emplea para especificar valores diferentes a las opciones por defecto. Las opciones se indican en forma de array asociativo:

```
new Ajax.Request('/ruta/hasta/pagina.php', {
  method: 'post',
  asynchronous: true,
  postBody: 'parametro1=valor1&parametro2=valor2',
  onSuccess: procesaRespuesta,
```

```
  onFailure: muestraError
});
```

Como es habitual, para establecer la función que procesa la respuesta del servidor, se indica el nombre de la función sin paréntesis. Las funciones externas asignadas para procesar la respuesta, reciben como primer parámetro el objeto que representa la respuesta del servidor. Haciendo uso de este objeto, las funciones pueden acceder a todas las propiedades habituales:

```
function procesaRespuesta(respuesta) {
  alert(respuesta.responseText);
}
```

A continuación se incluye una tabla con todas las opciones que se pueden definir para el método `Ajax.Request()`:

Opción	**Descripción**
`method`	El método de la petición HTTP. Por defecto es `POST`
`parameters`	Lista de valores que se envían junto con la petición. Deben estar formateados como una *query string*: `parametro1=valor1&parametro2=valor2`
`encoding`	Indica la codificación de los datos enviados en la petición. Su valor por defecto es `UTF-8`
`asynchronous`	Controla el tipo de petición que se realiza. Por defecto es `true`, lo que indica que la petición realizada al servidor es asíncrona, el tipo de petición habitual en las aplicaciones AJAX
`postBody`	Contenido que se envía en el cuerpo de la petición de tipo `POST`
`contentType`	Indica el valor de la cabecera `Content-Type` utilizada para realizar la petición. Su valor por defecto es `application/x-www-form-urlencoded`
`requestHeaders`	Array con todas las cabeceras propias que se quieren enviar junto con la petición

`onComplete`
`onLoaded`
`on404`
`on500`
	Permiten asignar funciones para el manejo de las distintas fases de la petición. Se pueden indicar funciones para todos los códigos de estado válidos de HTTP

`onSuccess`	Permite indicar la función que se encarga de procesar las respuestas correctas de servidor
`onFailure`	Se emplea para indicar la función que se ejecuta cuando la respuesta ha sido incorrecta
`onException`	Permite indicar la función encargada de manejar las peticiones erróneas en las que la respuesta del servidor no es válida, los argumentos que se incluyen en la petición no son válidos, etc.

La función `Ajax.Updater()` es una *versión especial* de `Ajax.Request()` que se emplea para actualizar el contenido HTML de un elemento de la página con la respuesta del servidor.

```
<div id="info"></div>

new Ajax.Updater('info', '/ruta/hasta/pagina.php');
```

Si la respuesta del servidor es

```
<ul>
  <li>Lorem ipsum dolor sit amet</li>
  <li>Consectetuer adipiscing elit</li>
  <li>Curabitur risus magna, lobortis</li>
</ul>
```

Después de realizar la petición de tipo `Ajax.Updater()`, el contenido HTML de la respuesta del servidor se muestra dentro del `<div>`:

```
<div id="info">
  <ul>
    <li>Lorem ipsum dolor sit amet</li>
    <li>Consectetuer adipiscing elit</li>
    <li>Curabitur risus magna, lobortis</li>
  </ul>
</div>
```

La sintaxis de `Ajax.Updater()` se muestra a continuación:

```
new Ajax.Updater(elemento, url, opciones);
```

Además de todas las opciones de `Ajax.Request()`, la función `Ajax.Updater()` permite establecer las siguientes opciones:

Opción	Descripción
`insertion`	Indica cómo se inserta el contenido HTML en el elemento indicado. Puede ser `Insertion.Before`, `Insertion.Top`, `Insertion.Bottom` o `Insertion.After`
`evalScripts`	Si la respuesta del servidor incluye scripts en su contenido, esta opción permite indicar si se ejecutan o no. Su valor por defecto es `false`, por lo que no se ejecuta ningún script

La función `Ajax.PeriodicalUpdater()` es una versión especializada de `Ajax.Updater()`, que se emplea cuando se quiere ejecutar de forma repetitiva una llamada a `Ajax.Updater()`. Esta función puede ser útil para ofercer información *en tiempo real* como noticias:

```
<div id="titulares"></div>

new Ajax.PeriodicalUpdater('titulares', '/ruta/hasta/pagina.php', { frequency:30 });
```

El código anterior actualiza, cada 30 segundos, el contenido del `<div>` con la respuesta recibida desde el servidor.

Además de todas las opciones anteriores, `Ajax.PeriodicalUpdater()` dispone de las siguientes opciones propias:

Opción	Descripción
`frequency`	Número de segundos que se espera entre las peticiones. El valor por defecto es de 2 segundos
`decay`	Indica el factor que se aplica a la frecuencia de actualización cuando la última respuesta del servidor es igual que la anterior. Ejemplo: si la frecuencia es `10` segundos y el `decay` vale `3`, cuando una respuesta del servidor sea igual a la anterior, la siguiente petición se hará `3 * 10 = 30` segundos después de la última petición

Por último, `Ajax.Responders` permite asignar de forma global las funciones que se encargan de responder a los eventos AJAX. Una de las principales utilidades de `Ajax.Responders` es la de indicar al usuario en todo momento si se está realizando alguna petición AJAX.

Los dos métodos principales de `Ajax.Responders` son `register()` y `unregister()` a los que se pasa como argumento un objeto de tipo array asociativo que incluye las funciones que responden a cada evento:

```
Ajax.Responders.register({
  onCreate: function() {
    if($('info') && Ajax.activeRequestCount> 0) {
      $('info').innerHTML = Ajax.activeRequestCount + "peticiones pendientes";
    }
  },
  onComplete: function() {
    if($('info') && Ajax.activeRequestCount> 0) {
      $('info').innerHTML = Ajax.activeRequestCount + "peticiones pendientes";
    }
  }
});
```

10.1.9. Funciones para eventos

El módulo de eventos de Prototype es uno de los menos desarrollados, por lo que va a ser completamente rediseñado en las próximas versiones del framework. Aún así, Prototype ofrece una solución sencilla y compatible con todos los navegadores para manejar los eventos de la aplicación. `Event.observe()` registra los eventos, `Event` almacena el objeto con la información del evento producido y `Event.stopObserving()` permite eliminar los eventos registrados.

```
<div id="pinchable">Si se pulsa en este DIV, se muestra un mensaje</div>

// Registrar el evento
Event.observe('pinchable', 'click', procesaEvento, false);

// Eliminar el evento registrado
// Event.stopObserving('pinchable', 'click', procesaEvento, false);

function procesaEvento(e) {
  // Obtener el elemento que ha originado el evento (el DIV)
  var elemento = Event.element(e);

  // Determinar la posicion del puntero del ratón
  var coordenadas = [Event.pointerX(e), Event.pointerY(e)];

  // Mostrar mensaje con los datos obtenidos
  alert("Has pinchado el DIV '"+elemento.id+"' con el raton en la posicion
("+coordenadas[0]+","+coordenadas[1]+")");

  // Evitar la propagacion del evento
  Event.stop(e);
}
```

La sintaxis completa del método `Event.observe()` se muestra a continuación:

```
Event.observe(elemento, nombreEvento, funcionManejadora, [usarCapture]);
```

El primer argumento (`elemento`) indica el identificador del elemento o el propio elemento que puede originar el evento. El segundo argumento (`nombreEvento`) indica el nombre del evento que se quiere manejar, sin incluir el prefijo on (`load`, `click`, `mouseover`, etc.). El tercer argumento (`funcionManejadora`) es el nombre de la función que procesa el evento cuando se produce. El último parámetro (`usarCapture`) no se suele emplear, pero indica si se debe utilizar la fase de `capture` o la fase de `bubbling`.

El objeto `Event` incluye la información disponible sobre el evento producido. A continuación se muestra una tabla con sus métodos y propiedades principales:

Método/Propiedad	**Descripción**
`element()`	Devuelve el elemento que ha originado el evento (un `div`, un botón, etc.)
`isLeftClick()`	Indica si se ha pulsado el botón izquierdo del ratón
`pointerX()` `pointerY()`	Posición `x` e `y` del puntero del ratón
`stop()`	Detiene la propagación del evento
`observers()`	Devuelve un array con todos los eventos registrados en la página

Además, `Event` define una serie de constantes para referirse a las teclas más habituales que se manejan en las aplicaciones (tabulador, `ENTER`, flechas de dirección, etc.) Las constantes definidas son `KEY_BACKSPACE`, `KEY_TAB`, `KEY_RETURN`, `KEY_ESC`, `KEY_LEFT`, `KEY_UP`, `KEY_RIGHT`, `KEY_DOWN`, `KEY_DELETE`.

Prototype también incluye otros métodos útiles para la gestión de eventos con formularios:

```
Form.Observer(formulario, frecuencia, funcionManejadora);
```

`Form.Observer()` permite *monitorizar* el formulario indicado cada cierto tiempo (el tiempo se indica en segundos mediante el parámetro `frecuencia`). Si se produce cualquier cambio en el formulario, se ejecuta la función cuyo nombre se indica en el parámetro `funcionManejadora`. `Form.Observer()` se emplea para los formularios que contienen elementos sin eventos registrados que procesen sus cambios.

Otra función similar es `Form.EventObserver()` cuya definición formal es:

```
Form.EventObserver(formulario, funcionManejadora);
```

La principal diferencia de `Form.EventObserver()` respecto a `Form.Observer()` es que, en este caso, se utilizan los eventos registrados por los elementos del formulario para detectar si se ha producido algún cambio en el formulario.

10.1.10. Métodos para funciones

Cuando se pasan referencias a funciones en el código de JavaScript, es posible que se pierda el contexto original de la función. El contexto de las funciones es fundamental para el correcto funcionamiento de la palabra reservada `this`.

Prototype incluye la posibilidad de asegurar que una función se va a ejecutar en un determinado contexto. Para ello, extiende la clase `Function()` para añadir el método `bind()`. Considerando el siguiente ejemplo:

```
nombre = 'Estoy fuera';
var objeto = {
  nombre: 'Estoy dentro',
  funcion: function() {
    alert(this.nombre);
  }
};

function ejecutaFuncion(f) {
  f();
}

var funcion2 = objeto.funcion.bind(objeto);

ejecutaFuncion(objeto.funcion);
ejecutaFuncion(funcion2);
```

El código anterior define en primer lugar la variable global `nombre` y le asigna el valor `Estoy fuera`. A continuación, se define un objeto con un atributo llamado también `nombre` y con un método sencillo que muestra el valor del atributo utilizando la palabra reservada `this`.

Si se ejecuta la función del objeto a través de una referencia suya (mediante la función `ejecutaFuncion()`), la palabra reservada `this` se resuelve en el objeto `window` y por tanto el mensaje que se muestra es `Estoy fuera`. Sin embargo, si se utiliza el método `bind(objeto)` sobre la función, siempre se ejecuta considerando su contexto igual al objeto que se pasa como parámetro al método `bind()`.

Prototype incluye además el método `bindAsEventListener()` que es equivalente a `bind()` pero que se puede emplear para evitar algunos de los problemas comunes de los eventos que se producen en algunos navegadores como Internet Explorer.

10.1.11. Rehaciendo ejemplos con Prototype

Las aplicaciones realizadas con el framework Prototype suelen ser muy concisas en comparación con las aplicaciones JavaScript tradicionales, pero siguen manteniendo una gran facilidad para leer su código y entenderlo.

Por ejemplo, el ejercicio que mostraba y ocultaba diferentes secciones de contenidos se realizó de la siguiente manera:

```
function muestraOculta() {
  // Obtener el ID del elemento
  var id = this.id;
  id = id.split('_');
  id = id[1];

  var elemento = document.getElementById('contenidos_'+id);
  var enlace = document.getElementById('enlace_'+id);
```

```
  if(elemento.style.display == "" || elemento.style.display == "block") {
    elemento.style.display = "none";
    enlace.innerHTML = 'Mostrar contenidos';
  }
  else {
    elemento.style.display = "block";
    enlace.innerHTML = 'Ocultar contenidos';
  }
}

window.onload = function() {
  document.getElementById('enlace_1').onclick = muestraOculta;
  document.getElementById('enlace_2').onclick = muestraOculta;
  document.getElementById('enlace_3').onclick = muestraOculta;
}
```

Con Prototype, su código se puede reducir a las siguientes instrucciones:

```
function muestraOculta() {
  var id = (this.id).split('_')[1];

  $('contenidos_'+id).toggle();
  $('enlace_'+id).innerHTML = (!$('contenidos_'+id).visible()) ? 'Ocultar contenidos' :
'Mostrar contenidos';
}

window.onload = function() {
  $R(1, 3).each(function(n) {
    Event.observe('enlace_'+n, 'click', muestraOculta);
  });
}
```

Los métodos `$R()`, `toggle()` y `visible()` permiten simplificar el código original a una mínima parte de su longitud, pero conservando el mismo funcionamiento, además de ser un código sencillo de entender.

Otro de los ejercicios anteriores realizaba peticiones AJAX al servidor para comprobar si un determinado nombre de usuario estaba libre. El código original de JavaScript era:

```
var READY_STATE_COMPLETE = 4;
var peticion_http = null;

function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function comprobar() {
  var login = document.getElementById("login").value;
  peticion_http = inicializa_xhr();
```

```
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/compruebaDisponibilidad.php", true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send("login="+login+"&nocache="+Math.random());
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      var login = document.getElementById("login").value;
      if(peticion_http.responseText == "si") {
        document.getElementById("disponibilidad").innerHTML = "El nombre elegido
["+login+"] está disponible";
      }
      else {
        document.getElementById("disponibilidad").innerHTML = "NO está disponible el
nombre elegido ["+login+"]";
      }
    }
  }
}

window.onload = function() {
  document.getElementById("comprobar").onclick = comprobar;
}
```

Con Prototype se puede conseguir el mismo comportamiento con tres veces menos de líneas de código:

```
function comprobar() {
  var login = $F('login');
  var url = 'http://localhost/compruebaDisponibilidad.php?nocache=' + Math.random();
  var peticion = new Ajax.Request(url, {
    method: 'post',
    postBody: 'login='+login,
    onSuccess: function(respuesta) {
      $('disponibilidad').innerHTML = (respuesta.responseText == 'si') ?
        'El nombre elegido ['+login+'] está disponible' : 'NO está disponible el nombre
elegido ['+login+']';
    },
    onFailure: function() { alert('Se ha producido un error'); }
  });
}

window.onload = function() {
  Event.observe('comprobar', 'click', comprobar);
}
```

10.2. La librería scriptaculous

Script.aculo.us (http://script.aculo.us/) es una de las muchas librerías que han surgido para facilitar el desarrollo de aplicaciones JavaScript y que están basadas en Prototype. El autor original de la librería es **Thomas Fuchs**, aunque actualmente recibe contribuciones de numerosos programadores, ya que la librería se distribuye de forma completamente gratuita y dispone de una buena documentación: http://wiki.script.aculo.us/scriptaculous/

La librería está dividida en varios módulos:

- Efectos: permite añadir de forma muy sencilla efectos especiales a cualquier elemento de la página. La librería incluye una serie de efectos básicos y otros efectos complejos construidos con la combinación de esos efectos básicos. Entre los efectos prediseñados se encuentran el parpadeo, movimiento rápido, aparecer/desaparecer, aumentar/disminuir de tamaño, desplegarse, etc.
- Controles: define varios *controles* que se pueden añadir directamente a cualquier aplicación web. Los tres controles que forman este módulo son: "arrastrar y soltar", que permite definir los elementos que se pueden arrastrar y las zonas en las que se pueden soltar elementos; "autocompletar", que permite definir un cuadro de texto en el que los valores que se escriben se autocompletan con ayuda del servidor; editor de contenidos, que permite modificar los contenidos de cualquier página web añadiendo un sencillo editor AJAX en cada elemento.
- Utilidades: la utilidad principal que incluye se llama `builder`, que se utiliza para crear fácilmente nodos y fragmentos complejos de DOM.

La documentación de script.aculo.us es muy completa e incluye muchos ejemplos, por lo que a continuación sólo se muestra el uso de uno de sus componentes más populares. En uno de los ejercicios anteriores, se realizaba un ejemplo de autocompletar el texto introducido por el usuario. El código completo del ejercicio ocupa más de 140 líneas.

El siguiente código hace uso de script.aculo.us para conseguir el mismo resultado con un 90% menos de líneas de código:

```
window.onload = function() {
  // Crear elemento de tipo <div> para mostrar las sugerencias del servidor
  var elDiv = Builder.node('div', {id:'sugerencias'});
  document.body.appendChild(elDiv);

  new Ajax.Autocompleter('municipio', 'sugerencias',
    'http://localhost/autocompletaMunicipios.php?modo=ul',
    {paramName: 'municipio'}
  );
}
```

La sintaxis del control `Ajax.Autocompleter()` es la siguiente:

```
new Ajax.Autocompleter(idCuadroTexto, idDivResultados, url, opciones);
```

El primer parámetro (`idCuadroTexto`) es el identificador del cuadro de texto en el que el usuario escribe las letras que se van a autocompletar. El segundo parámetro (`idDivResultados`) indica el

identificador del elemento `<div>` en el que se va a mostrar la respuesta del servidor. En el ejemplo anterior, este `<div>` se crea dinámicamente cuando se carga la página. El tercer parámetro (`url`) indica la URL del script de servidor que recibe las letras escritas por el usuario y devuelve la lista de sugerencias que se muestran. El último parámetro (`opciones`) permite modificar algunas de las opciones por defecto de este control.

A continuación se muestran las opciones más importantes disponibles para el control de autocompletar:

Opción	Descripción
`paramName`	El nombre del parámetro que se envía al servidor con el texto escrito por el usuario. Por defecto es igual que el atributo `name` del cuadro de texto utilizado para autocompletar
`tokens`	Permite autocompletar más de un valor en un mismo cuadro de texto. Más adelante se explica con un ejemplo.
`minChars`	Número mínimo de caracteres que el usuario debe escribir antes de que se realice la petición al servidor. Por defecto es igual a 1 carácter.
`indicator`	Elemento que se muestra mientras se realiza la petición al servidor y que se vuelve a ocultar al recibir la respuesta del servidor. Normalmente es una imagen animada que se utiliza para indicar al usuario que en ese momento se está realizando una petición al servidor
`updateElement`	Función que se ejecuta después de que el usuario seleccione un elemento de la lista de sugerencias. Por defecto el comportamiento consiste en seleccionar el elemento, mostrarlo en el cuadro de texto y ocultar la lista de sugerencias. Si se indica una función propia, no se ejecuta este comportamiento por defecto.
`afterUpdateElement`	Similar a la opción `updateElement`. En este caso, la función indicada se ejecuta después de la función por defecto y no en sustitución de esa función por defecto.

La opción `tokens` permite indicar los caracteres que separan los diferentes elementos de un cuadro de texto. En el siguiente ejemplo:

```
new Ajax.Autocompleter('municipio', 'sugerencias',
  'http://localhost/autocompletaMunicipios.php?modo=ul',
  { paramName: 'municipio', tokens: ',' }
);
```

La opción `tokens` indica que el carácter `","` separa los diferentes elementos dentro de un mismo cuadro de texto. De esta forma, si después de autocompletar una palabra se escribe un carácter `","` el script autocompletará la siguiente palabra.

10.3. La librería jQuery

jQuery (http://jquery.com/) es la librería JavaScript que ha irrumpido con más fuerza como alternativa a Prototype. Su autor original es **John Resig**, aunque como sucede con todas las librerías exitosas, actualmente recibe contribuciones de decenas de programadores. jQuery también ha sido programada de forma muy eficiente y su versión comprimida apenas ocupa 20 KB.

jQuery comparte con Prototype muchas ideas e incluso dispone de funciones con el mismo nombre. Sin embargo, su diseño interno tiene algunas diferencias drásticas respecto a Prototype, sobre todo el *"encadenamiento"* de llamadas a métodos.

La documentación de jQuery es muy completa en inglés e incluye muchos ejemplos. Además, también existen algunos recursos útiles en español para aprender su funcionamiento básico: http://docs.jquery.com/

10.3.1. Funciones y métodos básicos

La función básica de jQuery y una de las más útiles tiene el mismo nombre que en Prototype, ya que se trata de la *"función dolar"*: `$()`. A diferencia de la función de Prototype, la de jQuery es mucho más que un simple atajo mejorado de la función `document.getElementById()`.

La cadena de texto que se pasa como parámetro puede hacer uso de Xpath o de CSS para seleccionar los elementos. Además, separando expresiones con un carácter `","` se puede seleccionar un número ilimitado de elementos.

```
// Selecciona todos los enlaces de la página
$('a')

// Selecciona el elemento cuyo id sea "primero"
$('#primero')

// Selecciona todos los h1 con class "titular"
$('h1.titular')

// Selecciona todo lo anterior
$('a, #primero, h1.titular')
```

Las posibilidades de la función `$()` van mucho más allá de estos ejemplos sencillos, ya que soporta casi todos los selectores definidos por CSS 3 (algo que dispondrán los navegadores dentro de varios años) y también permite utilizar XPath:

```
// Selecciona todos los párrafos de la página que tengan al menos un enlace
$('p[a]')

// Selecciona todos los radiobutton de los formularios de la página
$('input:radio')

// Selecciona todos los enlaces que contengan la palabra "Imprimir"
 $('a:contains("Imprimir")');

// Selecciona los div que no están ocultos
$('div:visible')

// Selecciona todos los elementos pares de una lista
$("ul#menuPrincipal li:even")

// Selecciona todos los elementos impares de una lista
$("ul#menuPrincipal li:odd")
```

```
// Selecciona los 5 primeros párrafos de la página
$("p:lt(5)")
```

Como se puede comprobar, las posibilidades de la función `$()` son prácticamente ilimitadas, por lo que la documentación de jQuery sobre los selectores (http://docs.jquery.com/Selectors) disponibles es la mejor forma de descubrir todas sus posibilidades.

10.3.2. Funciones para eventos

Una de las utilidades más interesantes de jQuery está relacionada con el evento `onload` de la página. Las aplicaciones web más complejas suelen utilizar un código similar al siguiente para iniciar la aplicación:

```
window.onload = function() {
  ...
};
```

Hasta que no se carga la página, el navegador no construye el árbol DOM, lo que significa que no se pueden utilizar funciones que seleccionen elementos de la página, ni se pueden añadir o eliminar elementos. El problema de `window.onload` es que el navegador espera a que la página se cargue completamente, incluyendo todas las imágenes y archivos externos que se hayan enlazado.

jQuery propone el siguiente código para ejecutar las instrucciones una vez que se ha cargado la página:

```
$(document).ready(function() {
  ...
});
```

La gran ventaja del método propuesto por jQuery es que la aplicación no espera a que se carguen todos los elementos de la página, sino que sólo espera a que se haya descargado el contenido HTML de la página, con lo que el árbol DOM ya está disponible para ser manipulado. De esta forma, las aplicaciones JavaScript desarrolladas con jQuery pueden iniciarse más rápidamente que las aplicaciones JavaScript tradicionales.

En realidad, `ready()` no es más que una de las muchas funciones que componen el módulo de los eventos. Todos los eventos comunes de JavaScript (`click`, `mousemove`, `keypress`, etc.) disponen de una función con el mismo nombre que el evento. Si se utiliza la función sin argumentos, se ejecuta el evento:

```
// Ejecuta el evento 'onclick' en todos los párrafos de la página
$('p').click();

// Ejecuta el evento 'mouseover' sobre un 'div' con id 'menu'
$('div#menu').mouseover();
```

No obstante, el uso más habitual de las funciones de cada evento es el de establecer la función manejadora que se va a ejecutar cuando se produzca el evento:

```
// Establece la función manejadora del evento 'onclick'
// a todos los párrafos de la página
$('p').click(function() {
```

```
  alert($(this).text());
});

// Establece la función manejadora del evento 'onblur'
// a los elementos de un formulario
$('#elFormulario :input').blur(function() {
  valida($(this));
});
```

Entre las utilidades definidas por jQuery para los eventos se encuentra la función `toggle()`, que permite ejecutar dos funciones de forma alterna cada vez que se pincha sobre un elemento:

```
$("p").toggle(function(){
  alert("Me acabas de activar");
},function(){
  alert("Me acabas de desactivar");
});
```

En el ejemplo anterior, la primera vez que se pincha sobre el elemento (y todas las veces impares), se ejecuta la primera función y la segunda vez que se pincha el elemento (y todas las veces pares) se ejecuta la segunda función.

10.3.3. Funciones para efectos visuales

Las aplicaciones web más avanzadas incluyen efectos visuales complejos para construir interacciones similares a las de las aplicaciones de escritorio. jQuery incluye en la propia librería varios de los efectos más comunes:

```
// Oculta todos los enlaces de la página
$('a').hide();

// Muestra todos los 'div' que estaban ocultos
$('div:hidden').show();

// Muestra los 'div' que estaba ocultos y oculta
// los 'div' que eran visibles
$('div').toggle();
```

Todas las funciones relacionadas con los efectos visuales permiten indicar dos parámetros opcionales: el primero es la duración del efecto y el segundo parámetro es la función que se ejecuta al finalizar el efecto visual.

Otros efectos visuales incluidos son los relacionados con el fundido o *"fading"* (`fadeIn()` muestra los elementos con un fundido suave, `fadeOut()` oculta los elementos con un fundido suave y `fadeTo()` establece la opacidad del elemento en el nivel indicado) y el despliegue de elementos (`slideDown()` hace aparecer un elemento desplegándolo en sentido descendente, `slideUp()` hace desaparecer un elemento desplegándolo en sentido ascendente, `slideToggle()` hace desaparecer el elemento si era visible y lo hace aparecer si no era visible).

10.3.4. Funciones para AJAX

Como sucede con Prototype, las funciones y utilidades relacionadas con AJAX son parte fundamental de jQuery. El método principal para realizar peticiones AJAX es `$.ajax()` (importante no olvidar el punto entre `$` y `ajax`). A partir de esta función básica, se han definido otras funciones relacionadas, de más alto nivel y especializadas en tareas concretas: `$.get()`, `$.post()`, `$.load()`, etc.

La sintaxis de `$.ajax()` es muy sencilla:

```
$.ajax(opciones);
```

Al contrario de lo que sucede con Prototype, la URL que se solicita también se incluye dentro del array asociativo de opciones. A continuación se muestra el mismo ejemplo básico que se utilizó en Prototype realizado con `$.ajax()`:

```
$.ajax({
  url: '/ruta/hasta/pagina.php',
  type: 'POST',
  async: true,
  data: 'parametro1=valor1&parametro2=valor2',
  success: procesaRespuesta,
  error: muestraError
});
```

La siguiente tabla muestra todas las opciones que se pueden definir para el método `$.ajax()`:

Opción	Descripción
`async`	Indica si la petición es asíncrona. Su valor por defecto es `true`, el habitual para las peticiones AJAX
`beforeSend`	Permite indicar una función que modifique el objeto `XMLHttpRequest` antes de realizar la petición. El propio objeto `XMLHttpRequest` se pasa como único argumento de la función
`complete`	Permite establecer la función que se ejecuta cuando una petición se ha completado (y después de ejecutar, si se han establecido, las funciones de `success` o `error`). La función recibe el objeto `XMLHttpRequest` como primer parámetro y el resultado de la petición como segundo argumento
`contentType`	Indica el valor de la cabecera `Content-Type` utilizada para realizar la petición. Su valor por defecto es `application/x-www-form-urlencoded`
`data`	Información que se incluye en la petición. Se utiliza para enviar parámetros al servidor. Si es una cadena de texto, se envía tal cual, por lo que su formato debería ser `parametro1=valor1&parametro2=valor2`. También se puede indicar un array asociativo de pares clave/valor que se convierten automáticamente en una cadena tipo *query string*
`dataType`	El tipo de dato que se espera como respuesta. Si no se indica ningún valor, jQuery lo deduce a partir de las cabeceras de la respuesta. Los posibles valores son: `xml` (se devuelve un documento XML correspondiente al valor `responseXML`), `html` (devuelve directamente la respuesta del servidor mediante el valor `responseText`), `script` (se evalúa la respuesta como si fuera JavaScript y se devuelve el resultado) y `json` (se evalúa la respuesta como si fuera JSON y se devuelve el objeto JavaScript generado)

`error`	Indica la función que se ejecuta cuando se produce un error durante la petición. Esta función recibe el objeto `XMLHttpRequest` como primer parámetro, una cadena de texto indicando el error como segundo parámetro y un objeto con la excepción producida como tercer parámetro
`ifModified`	Permite considerar como correcta la petición solamente si la respuesta recibida es diferente de la anterior respuesta. Por defecto su valor es `false`
`processData`	Indica si se transforman los datos de la opción `data` para convertirlos en una cadena de texto. Si se indica un valor de `false`, no se realiza esta transformación automática
`success`	Permite establecer la función que se ejecuta cuando una petición se ha completado de forma correcta. La función recibe como primer parámetro los datos recibidos del servidor, previamente formateados según se especifique en la opción `dataType`
`timeout`	Indica el tiempo máximo, en milisegundos, que la petición espera la respuesta del servidor antes de anular la petición
`type`	El tipo de petición que se realiza. Su valor por defecto es `GET`, aunque también se puede utilizar el método `POST`
`url`	La URL del servidor a la que se realiza la petición

Además de la función `$.ajax()` genérica, existen varias funciones relacionadas que son versiones simplificadas y especializadas de esa función. Así, las funciones `$.get()` y `$.post()` se utilizan para realizar de forma sencilla peticiones `GET` y `POST`:

```
// Petición GET simple
$.get('/ruta/hasta/pagina.php');

// Petición GET con envío de parámetros y función que
// procesa la respuesta
$.get('/ruta/hasta/pagina.php',
  { articulo: '34' },
  function(datos) {
    alert('Respuesta = '+datos);
  });
```

Las peticiones `POST` se realizan exactamente de la misma forma, por lo que sólo hay que cambiar `$.get()` por `$.post()`. La sintaxis de estas funciones son:

```
$.get(url, datos, funcionManejadora);
```

El primer parámerto (`url`) es el único obligatorio e indica la URL solicitada por la petición. Los otros dos parámetros son opcionales, siendo el segundo (`datos`) los parámetros que se envían junto con la petición y el tercero (`funcionManejadora`) el nombre o el código JavaScript de la función que se encarga de procesar la respuesta del servidor.

La función `$.get()` dispone a su vez de una versión especializada denominada `$.getIfModified()`, que también obtiene una respuesta del servidor mediante una petición `GET`, pero la respuesta sólo está disponible si es diferente de la última respuesta recibida.

jQuery también dispone de la función `$.load()`, que es idéntica a la función `Ajax.Updater()` de Prototype. La función `$.load()` inserta el contenido de la respuesta del servidor en el elemento

de la página que se indica. La forma de indicar ese elemento es lo que diferencia a jQuery de Prototype:

```
<div id="info"></div>

// Con Prototype
new Ajax.Updater('info', '/ruta/hasta/pagina.php');

// Con jQuery
$('#info').load('/ruta/hasta/pagina.php');
```

Al igual que sucedía con la función `$.get()`, la función `$.load()` también dispone de una versión específica denominada `$.loadIfModified()` que carga la respuesta del servidor en el elemento sólo si esa respuesta es diferente a la última recibida.

Por último, jQuery también dispone de las funciones `$.getJSON()` y `$.getScript()` que cargan y evalúan/ejecutan respectivamente una respuesta de tipo JSON y una respuesta con código JavaScript.

10.3.5. Funciones para CSS

jQuery dispone de varias funciones para la manipulación de las propiedades CSS de los elementos. Todas las funciones se emplean junto con una selección de elementos realizada con la función `$()`.

Si la función obtiene el valor de las propiedades CSS, sólo se obtiene el valor de la propiedad CSS del primer elemento de la selección realizada. Sin embargo, si la función establece el valor de las propiedades CSS, se establecen para todos los elementos seleccionados.

```
// Obtiene el valor de una propiedad CSS
// En este caso, solo para el primer 'div' de la página
$('div').css('background');

// Establece el valor de una propiedad CSS
// En este caso, para todos los 'div' de la página
$('div').css('color', '#000000');

// Establece varias propiedades CSS
// En este caso, para todos los 'div' de la página
$('div').css({ padding: '3px', color: '#CC0000' });
```

Además de las funciones anteriores, CSS dispone de funciones específicas para obtener/establecer la altura y anchura de los elementos de la página:

```
// Obtiene la altura en píxel del primer 'div' de la página
$('div').height();

// Establece la altura en píxel de todos los 'div' de la página
$('div').height('150px');

// Obtiene la anchura en píxel del primer 'div' de la página
$('div').width();
```

```
// Establece la anchura en píxel de todos los 'div' de la página
$('div').width('300px');
```

10.3.6. Funciones para nodos DOM

La función `$()` permite seleccionar elementos (nodos DOM) de la página de forma muy sencilla. jQuery permite, además, seleccionar nodos relacionados con la selección realizada. Para seleccionar nodos relacionados, se utilizan funciones de filtrado y funciones de búsqueda.

Los filtros son funciones que modifican una selección realizada con la función `$()` y permiten limitar el número de nodos devueltos.

La función `contains()` limita los elementos seleccionados a aquellos que contengan en su interior el texto indicado como parámetro:

```
// Sólo obtiene los párrafos que contengan la palabra 'importante'
$('p').contains('importante');
```

La función `not()` elimina de la selección de elementos aquellos que cumplan con el selector indicado:

```
// Selecciona todos los enlaces de la página, salvo el que
// tiene una 'class' igual a 'especial'
$('a').not('.especial');
// La siguiente instrucción es equivalente a la anterior
$('a').not($('.especial'));
```

La función `filter()` es la inversa de `not()`, ya que elimina de la selección de elementos aquellos que no cumplan con la expresión indicada. Además de una expresión, también se puede indicar una función para filtrar la selección:

```
// Selecciona todas las listas de elementos de la página y quedate
// sólo con las que tengan una 'class' igual a 'menu'
$('ul').filter('.menu');
```

Una función especial relacionada con los filtros y buscadores es `end()`, que permite volver a la selección original de elementos después de realizar un filtrado de elementos. La documentación de jQuery incluye el siguiente ejemplo:

```
$('a')
  .filter('.pinchame')
  .click(function(){
    alert('Estás abandonando este sitio web');
  })
  .end()
  .filter('ocultame')
    .click(function(){
      $(this).hide();
      return false;
    })
  .end();
```

El código anterior obtiene todos los enlaces de la página `$('a')` y aplica diferentes funciones manejadoras del evento `click` en función del tipo de enlace. Aunque se podrían incluir dos

instrucciones diferentes para realizar cada filtrado, la función `end()` permite encadenar varias selecciones.

El primer filtrado (`$('a').filter('.pinchame')`) selecciona todos los elementos de la página cuyo atributo `class` sea igual a `pinchame`. Después, se asigna la función manejadora para el evento de pinchar con el ratón mediante la función `click()`.

A continuación, el código anterior realiza otro filtrado a partir de la selección original de enlaces. Para volver a la selección original, se utiliza la función `end()` antes de realizar un nuevo filtrado. De esta forma, la instrucción `.end().filter('ocultame')` es equivalente a realizar el filtrado directamente sobre la selección original `$('a').filter('.ocultame')`.

El segundo grupo de funciones para la manipulación de nodos DOM está formado por los buscadores, funciones que buscan/seleccionan nodos relacionados con la selección realizada. De esta forma, jQuery define la función `children()` para obtener todos los *nodos hijo* o descendientes del nodo actual, `parent()` para obtener el *nodo padre* o nodo ascendente del nodo actual (`parents()` obtiene todos los ascendentes del nodo hasta la raíz del árbol) y `siblings()` que obtiene todos los *nodos hermano* del nodo actual, es decir, todos los nodos que tienen el mismo *nodo padre* que el nodo actual.

La navegación entre *nodos hermano* se puede realizar con las funciones `next()` y `pev()` que avanzan o retroceden a través de la lista de *nodos hermano* del nodo actual.

Por último, jQuery también dispone de funciones para manipular fácilmente el contenido de los nodos DOM (http://docs.jquery.com/Manipulation) . Las funciones `append()` y `prepend()` añaden el contenido indicado como parámetro al principio o al final respectivamente del contenido original del nodo.

Las funciones `after()` y `before()` añaden el contenido indicado como parámetro antes de cada uno de los elementos seleccionados. La función `wrap()` permite *"envolver"* un elemento con el contenido indicado (se añade parte del contenido por delante y el resto por detrás).

La función `empty()` vacía de contenido a un elemento, `remove()` elimina los elementos seleccionados del árbol DOM y `clone()` copia de forma exacta los nodos seleccionados.

10.3.7. Otras funciones útiles

jQuery detecta automáticamente el tipo de navegador en el que se está ejecutando y permite acceder a esta información a través del objeto `$.browser`:

```
$.browser.msie;    // 'true' para navegadores de la familia Internet Explorer
$.browser.mozilla; // 'true' para navegadores de la familia Firefox
$.browser.opera;   // 'true' para navegadores de la familia Opera
$.browser.safari;  // 'true' para navegadores de la familia Safari
```

Recorrer arrays y objetos también es muy sencillo con jQuery, gracias a la función `$.each()`. El primer parámetro de la función es el objeto que se quiere recorrer y el segundo parámetro es el código de la función que lo recorre (a su vez, a esta función se le pasa como primer parámetro el índice del elemento y como segundo parámetro el valor del elemento):

```
// Recorrer arrays
var vocales = ['a', 'e', 'i', 'o', 'u'];

$.each( vocales, function(i, n){
  alert('Vocal número ' + i + " = " + n);
});

// Recorrer objetos
var producto = { id: '12DW2', precio: 12.34, cantidad: 5 };

$.each( producto, function(i, n){
  alert(i + ' : ' + n);
});
```

10.3.8. Rehaciendo ejemplos con jQuery

Como sucedía con Prototype, cuando se rehace una aplicación JavaScript con jQuery, el resultado es un código muy conciso pero que mantiene su facilidad de lectura y comprensión.

Por ejemplo, el ejercicio que mostraba y ocultaba diferentes secciones de contenidos se realizó con JavaScript de la siguiente manera:

```
function muestraOculta() {
  // Obtener el ID del elemento
  var id = this.id;
  id = id.split('_');
  id = id[1];

  var elemento = document.getElementById('contenidos_'+id);
  var enlace = document.getElementById('enlace_'+id);

  if(elemento.style.display == "" || elemento.style.display == "block") {
    elemento.style.display = "none";
    enlace.innerHTML = 'Mostrar contenidos';
  }
  else {
    elemento.style.display = "block";
    enlace.innerHTML = 'Ocultar contenidos';
  }
}

window.onload = function() {
  document.getElementById('enlace_1').onclick = muestraOculta;
  document.getElementById('enlace_2').onclick = muestraOculta;
  document.getElementById('enlace_3').onclick = muestraOculta;
}
```

Con Prototype, su código se redujo a las siguientes instrucciones:

```
function muestraOculta() {
  var id = (this.id).split('_')[1];

  $('contenidos_'+id).toggle();
  $('enlace_'+id).innerHTML = (!$('contenidos_'+id).visible()) ? 'Ocultar contenidos' :
'Mostrar contenidos';
```

```
}

window.onload = function() {
  $R(1, 3).each(function(n) {
    Event.observe('enlace_'+n, 'click', muestraOculta);
  });
}
```

Con jQuery, el mismo código se puede escribir de la siguiente forma:

```
$(document).ready(function(){
  $.each([1, 2, 3], function(i, n){
    $('#enlace_'+n).toggle(
      function() { $('#contenidos_'+n).toggle(); $(this).html('Mostrar contenidos'); },
      function() { $('#contenidos_'+n).toggle(); $(this).html('Ocultar contenidos'); }
    );
  })
});
```

El código anterior utiliza la función `toggle()` como evento que permite alternar la ejecución de dos funciones y como función que oculta un elemento visible y muestra un elemento oculto.

Otro de los ejercicios anteriores realizaba peticiones AJAX al servidor para comprobar si un determinado nombre de usuario estaba libre. El código original de JavaScript era:

```
var READY_STATE_COMPLETE=4;
var peticion_http = null;

function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function comprobar() {
  var login = document.getElementById("login").value;
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/compruebaDisponibilidad.php", true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send("login="+login+"&nocache="+Math.random());
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      var login = document.getElementById("login").value;
      if(peticion_http.responseText == "si") {
        document.getElementById("disponibilidad").innerHTML = "El nombre elegido
["+login+"] está disponible";
```

```
      }
      else {
        document.getElementById("disponibilidad").innerHTML = "NO está disponible el
nombre elegido ["+login+"]";
      }
    }
  }
}

window.onload = function() {
  document.getElementById("comprobar").onclick = comprobar;
}
```

Con Prototype se puede conseguir el mismo comportamiento con tres veces menos de líneas de código:

```
function comprobar() {
  var login = $F('login');
  var url = 'http://localhost/compruebaDisponibilidad.php?nocache=' + Math.random();
  var peticion = new Ajax.Request(url, {
    method:'post',
    postBody:'login='+login,
    onSuccess: function(respuesta) {
      $('disponibilidad').innerHTML = (respuesta.responseText == 'si') ?
        'El nombre elegido ['+login+'] está disponible' : 'NO está disponible el nombre
elegido ['+login+']';
    },
    onFailure: function() { alert('Se ha producido un error'); }
  });
}

window.onload = function() {
  Event.observe('comprobar', 'click', comprobar);
}
```

jQuery también permite simplificar notablemente el código de la aplicación original:

```
function comprobar() {
  var login = $('#login').value;
  var peticion = $.ajax({
    url: 'http://localhost/compruebaDisponibilidad.php?nocache=' + Math.random(),
    method: 'POST',
    data: { login: login },
    success: function(respuesta) {
      $('#disponibilidad').html((respuesta.responseText == 'si') ?
        'El nombre elegido ['+login+'] está disponible' :
        'NO está disponible el nombre elegido ['+login+']');
    },
    error: function() { alert('Se ha producido un error'); }
  });
}

$(document).ready(function(){
  $('#comprobar').click(comprobar);
});
```

10.4. Otros frameworks importantes

El *boom* de las aplicaciones web con interfaces dinámicas complejas y que incluyen múltiples interacciones AJAX ha provocado la irrupción de un gran número de frameworks especializados para el desarrollo de aplicaciones con JavaScript. Además de Prototype y jQuery, existen otros frameworks destacables:

- Dojo (http://dojotoolkit.org/) es mucho más que un framework, ya que sus propios creadores lo denominan *"el conjunto de herramientas ("toolkit") de JavaScript que permite desarrollar aplicaciones web profesionales de forma sencilla y más rápida"*. Además, Dojo dispone de una licencia de tipo software libre.
- Mootools (http://mootools.net/) es un framework que destaca por su reducido tamaño y por lo modular de su desarrollo. De hecho, al descargar el framework, se pueden elegir los componentes que se van a utilizar, para descargar una versión comprimida que sólo contenga los componentes escogidos. De esta forma, se puede reducir al mínimo el tamaño de los archivos descargados por los usuarios.
- Ext JS (http://extjs.com/) es otro de los frameworks más populares de JavaScript. Aunque comenzó siendo un añadido de la librería YUI de Yahoo (http://developer.yahoo.com/yui/) , pronto adquirió entidad propia. Además de las utilidades comunes, incluye una serie de componentes listos para usar y tiene una licencia de tipo software libre y otra licencia de tipo comercial si se desea obtener soporte técnico.

Capítulo 11. Otras utilidades

11.1. Detener las peticiones HTTP erróneas

La creación de aplicaciones AJAX implica la aparición de nuevos tipos de errores y excepciones. Probablemente, el problema más importante sea el de realizar una petición al servidor y que este no responda en un periodo de tiempo razonable.

Aunque las peticiones se realizan de forma asíncrona y el usuario puede continuar utilizando la aplicación mientras se realiza la petición al servidor en un segundo plano, normalmente es necesario disponer de una respuesta rápida del servidor.

La función `setTimeout()` se puede emplear para establecer una cuenta atrás al iniciar una nueva petición. Si el servidor responde antes de que expire la cuenta atrás, se elimina esa cuenta atrás y se continúa con la ejecución normal de la aplicación. Si el servidor no responde y la cuenta atrás finaliza, se ejecuta una función encargada de detener la petición, reintentarla, mostrar un mensaje al usuario, etc.

```
// Variable global que almacena el identificador de la cuenta atrás
var cuentaAtras = null;
var tiempoMaximo = 5000; // 5000 = 5 segundos

function cargaContenido(url, metodo, funcion) {
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    // Establecer la cuenta atrás al realizar la petición HTTP
    cuentraAtras = setTimeout(expirada, tiempoMaximo);
    peticion_http.onreadystatechange = funcion;
    peticion_http.open(metodo, url, true);
    peticion_http.send(null);
  }
}

function muestraMensaje() {
  ...
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      // Si se ha recibido la respuesta del servidor, eliminar la cuenta atrás
      clearTimeout(cuentaAtras);
      ...
    }
  }
}

function expirada() {
  // La cuentra atrás se ha cumplido, detener la petición HTTP pendiente
  peticion_http.abort();
  alert("Se ha producido un error en la comunicación con el servidor. Inténtalo un poco
más adelante.");
}
```

Además de la falta de respuesta del servidor, las aplicaciones AJAX deben estar preparadas para otros tipos de respuestas que pueden generar los servidores. El tipo de respuesta se comprueba mediante el valor del atributo `status` del objeto XMLHTTPRequest.

A continuación se muestran las tablas de los códigos de estado más comunes que pueden devolver los servidores:

Códigos de información

status	statusText	Explicación
100	Continue	Una parte de la petición (normalmente la primera) se ha recibido sin problemas y se puede enviar el resto de la petición
101	Switching protocols	El servidor va a cambiar el protocolo con el que se envía la información de la respuesta. En la cabecera `Upgrade` indica el nuevo protocolo

Códigos de petición y respuesta correctas

status	statusText	Explicación
200	OK	La petición se ha recibido correctamente y se está enviando la respuesta. Este código es con mucha diferencia el que mas devuelven los servidores
201	Created	Se ha creado un nuevo recurso (por ejemplo una página web o un archivo) como parte de la respuesta
202	Accepted	La petición se ha recibido correctamente y se va a responder, pero no de forma inmediata
203	Non-Authoritative Information	La respuesta que se envía la ha generado un servidor externo. A efectos prácticos, es muy parecido al código `200`
204	No Content	La petición se ha recibido de forma correcta pero no es necesaria una respuesta
205	Reset Content	El servidor solicita al navegador que inicialice el documento desde el que se realizó la petición, como por ejemplo un formulario
206	Partial Content	La respuesta contiene sólo la parte concreta del documento que se ha solicitado en la petición

Códigos de redirección

status	statusText	Explicación
300	Multiple Choices	El contenido original ha cambiado de sitio y se devuelve una lista con varias direcciones alternativas en las que se puede encontrar el contenido
301	Moved Permanently	El contenido original ha cambiado de sitio y el servidor devuelve la nueva URL del contenido. La próxima vez que solicite el contenido, el navegador utiliza la nueva URL
302	Found	El contenido original ha cambiado de sitio de forma temporal. El servidor devuelve la nueva URL, pero el navegador debe seguir utilizando la URL original en las próximas peticiones
303	See Other	El contenido solicitado se puede obtener en la URL alternativa devuelta por el servidor. Este código no implica que el contenido original ha cambiado de sitio

304	Not Modified	Normalmente, el navegador guarda en su caché los contenidos accedidos frecuentemente. Cuando el navegador solicita esos contenidos, incluye la condición de que no hayan cambiado desde la última vez que los recibió. Si el contenido no ha cambiado, el servidor devuelve este código para indicar que la respuesta sería la misma que la última vez
305	Use Proxy	El recurso solicitado sólo se puede obtener a través de un proxy, cuyos datos se incluyen en la respuesta
307	Temporary Redirect	Se trata de un código muy similar al 302, ya que indica que el recurso solicitado se encuentra de forma temporal en otra URL

Códigos de error del navegador

status	statusText	Explicación
400	Bad Request	El servidor no entiende la petición porque no ha sido creada de forma correcta
401	Unauthorized	El recurso solicitado requiere autorización previa
402	Payment Required	Código reservado para su uso futuro
403	Forbidden	No se puede acceder al recurso solicitado por falta de permisos o porque el usuario y contraseña indicados no son correctos
404	Not Found	El recurso solicitado no se encuentra en la URL indicada. Se trata de uno de los códigos más utilizados y responsable de los típicos errores de *Página no encontrada*
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta
409	Conflict	El navegador no puede procesar la petición, ya que implica realizar una operación no permitida (como por ejemplo crear, modificar o borrar un archivo)
410	Gone	Similar al código 404. Indica que el recurso solicitado ha cambiado para siempre su localización, pero no se proporciona su nueva URL
411	Length Required	El servidor no procesa la petición porque no se ha indicado de forma explícita el tamaño del contenido de la petición
412	Precondition Failed	No se cumple una de las condiciones bajo las que se realizó la petición
413	Request Entity Too Large	La petición incluye más datos de los que el servidor es capaz de procesar. Normalmente este error se produce cuando se adjunta en la petición un archivo con un tamaño demasiado grande
414	Request-URI Too Long	La URL de la petición es demasiado grande, como cuando se incluyen más de 512 bytes en una petición realizada con el método GET

415	Unsupported Media Type	Al menos una parte de la petición incluye un formato que el servidor no es capaz procesar
416	Requested Range Not Suitable	El trozo de documento solicitado no está disponible, como por ejemplo cuando se solicitan bytes que están por encima del tamaño total del contenido
417	Expectation Failed	El servidor no puede procesar la petición porque al menos uno de los valores incluidos en la cabecera `Expect` no se pueden cumplir

Códigos de error del servidor

status	statusText	Explicación
500	Internal Server Error	Se ha producido algún error en el servidor que impide procesar la petición
501	Not Implemented	Procesar la respuesta requiere ciertas características no soportadas por el servidor
502	Bad Gateway	El servidor está actuando de proxy entre el navegador y un servidor externo del que ha obtenido una respuesta no válida
503	Service Unavailable	El servidor está sobrecargado de peticiones y no puede procesar la petición realizada
504	Gateway Timeout	El servidor está actuando de proxy entre el navegador y un servidor externo que ha tardado demasiado tiempo en responder
505	HTTP Version Not Supported	El servidor no es capaz de procesar la versión HTTP utilizada en la petición. La respuesta indica las versiones de HTTP que soporta el servidor

11.2. Mejorar el rendimiento de las aplicaciones complejas

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.

11.3. Ofuscar el código JavaScript

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 comportamiento 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);m++;}}
```

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/) .

11.4. Evitar el problema de los dominios diferentes

Como ya se ha explicado y se ha podido comprobar en algunos de los ejercicios, los navegadores imponen restricciones muy severas a las conexiones de red que se pueden realizar mediante AJAX. Esta característica se conoce como *"el problema de los dominios diferentes"* (en inglés, *"cross-domain problem"*).

El código JavaScript alojado en un servidor, no puede realizar conexiones con otros dominios externos. También existen problemas entre subdominios de un mismo sitio web, que se pueden evitar con el uso de la propiedad `document.domain`.

Afortunadamente, existen opciones para poder realizar conexiones con cualquier dominio externo al servidor que aloja el código JavaScript. Todas las soluciones que son viables técnicamente y que funcionan de la misma forma en cualquier navegador hacen uso de recursos en el servidor que aloja el código JavaScript original.

La solución más sencilla es la de habilitar el módulo `mod_rewrite` en los servidores web de tipo Apache. Con este módulo activado, Apache se puede convertir en un proxy transparente que realice las peticiones externas en nombre del script y le devuelva los resultados.

En el siguiente ejemplo, el navegador descarga el script desde el `servidor1`. Por este motivo, el código del script puede solicitar recursos del `servidor1`, pero no puede establecer conexiones con el `servidor2`:

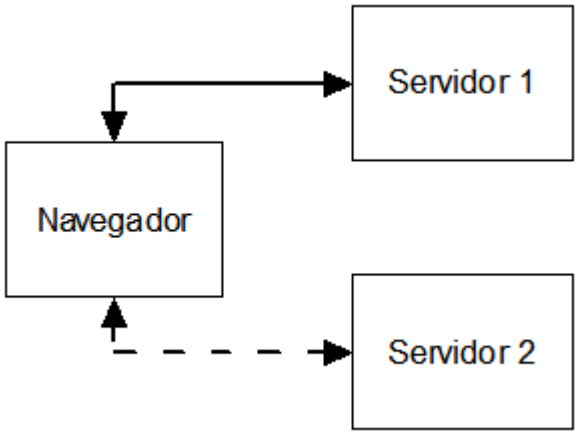

Figura 11.1. El script descargado desde el servidor1 no puede establecer conexiones de red con el servidor2

La solución más sencilla para resolver este problema consiste en configurar el servidor web del `servidor1`. Si se utiliza el servidor web Apache, para configurar el proxy transparente, se habilita el módulo `mod_rewrite` y se añaden las siguientes directivas a la configuración de Apache:

```
RewriteEngine on
RewriteRule ^/ruta/al/recurso$ http://www.servidor2.com/ruta/al/recurso [P]
```

Ahora, el código de la aplicación puede acceder a cualquier recurso del `servidor2` ya que:

- El script realiza peticiones a: http://www.servidor1.com/ruta/al/recurso
- En el `servidor1` se transforma automáticamente a: http://www.servidor2.com/ruta/al/recurso
- El `servidor1` obtiene la respuesta del `servidor2` y la envía de nuevo al script

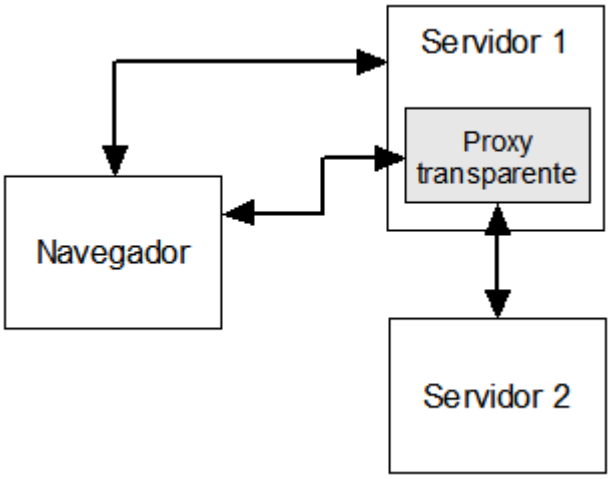

Figura 11.2. Utilizando un proxy transparente, los scripts pueden acceder a los recursos que se encuentren en cualquier servidor

Además de utilizar el servidor web como proxy transparente, también es posible diseñar un proxy a medida mediante software. Yahoo por ejemplo ofrece una extensa documentación para los desarrolladores de aplicaciones web. Entre esta documentación, se encuentra un artículo sobre el uso de proxys para evitar el problema de las peticiones externas de AJAX: http://developer.yahoo.com/javascript/howto-proxy.html

Además, Yahoo ofrece un proxy de ejemplo realizado con PHP y que puede ser utilizado para conectar aplicaciones JavaScript con sus servicios web.

Capítulo 12. Recursos útiles

Documentación y referencias:

- Ajax: A New Approach to Web Applications (http://www.adaptivepath.com/publications/essays/archives/000385.php) : el artículo en el que se acuñó por primera vez el término AJAX y en el que se explica su funcionamiento básico.
- Documentación de referencia sobre el objeto `XMLHttpRequest`: W3C (borrador) (http://www.w3.org/TR/XMLHttpRequest/) , Microsoft (Internet Explorer) (http://msdn2.microsoft.com/en-us/library/ms535874.aspx) , Mozilla (Firefox) (http://developer.mozilla.org/en/docs/XMLHttpRequest) , Apple (Safari) (http://developer.apple.com/internet/webcontent/xmlhttpreq.html) .

Noticias y actualidad:

- Ajaxian (http://www.ajaxian.com) : el blog más popular dedicado al mundo AJAX.

Librerías y frameworks:

- Prototype (http://www.prototypejs.org/) : el primer framework de JavaScript/AJAX de uso masivo.
- script.aculo.us (http://script.aculo.us/) : la librería basada en Prototype más utilizada.
- jQuery (http://jquery.com/) : la librería de JavaScript que ha surgido como alternativa a Prototype.

Otras utilidades:

- Ajaxload (http://www.ajaxload.info/) : utilidad para construir los pequeños iconos animados que las aplicaciones AJAX suelen utilizar para indicar al usuario que se está realizando una petición.
- MiniAjax (http://miniajax.com/) : decenas de ejemplos reales de aplicaciones AJAX listas para descargar (galerías de imágenes, reflejos para imágenes, formularios, tablas reordenables, etc.)

Capítulo 13. Bibliografía

Bibliografía sobre JavaScript:

- *Professional JavaScript for Web Developers*, Nicholas C. Zakas (ISBN: 978-0-7645-7908-0). A pesar de ser un libro *antiguo*, sigue siendo una buena referencia de aprendizaje de JavaScript para los programadores de aplicaciones web. Ver más información sobre el libro (http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764579088.html) .
- *JavaScript: The Definitive Guide (Fifth Edition)*, David Flanagan (ISBN-13: 9780596101992). Referencia completa de JavaScript en más de 1.000 páginas que explican con detalle cualquier aspecto de este lenguaje de programación. Ver más información sobre el libro (http://www.oreilly.com/catalog/jscript5/index.html)

Bibliografía sobre AJAX:

- *Pragmatic Ajax: A Web 2.0 Primer*, Justin Gehtland, Ben Galbraith, Dion Almaer (ISBN—13: 978-0976694083). Se trata de un libro muy práctico que incluye muchos ejemplos reales fáciles de entender a pesar de su complejidad. Ver más información sobre el libro (http://www.pragmaticprogrammer.com/title/ajax/) .
- *Ajax in Action*, Dave Crane, Eric Pascarello, Darren James (ISBN-13: 978-1932394610). Aunque no es tan práctico como el anterior, el código de los ejemplos incluidos está muy bien programado, lo que ayuda a crear aplicaciones muy profesionales. Ver más información sobre el libro (http://www.manning.com/crane/) .
- *Ajax Hacks*, Bruce Perry (ISBN-13: 978-0596101695). Colección de trucos y pequeñas utilidades listas para copiar+pegar en las aplicaciones reales. Ver más información sobre el libro (http://www.oreilly.com/catalog/ajaxhks/) .

Capítulo 14. Ejercicios resueltos

14.1. Ejercicio 1

```
<!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>Ejercicio 1 - Objetos</title>
<script type="text/javascript">
// Estructura básica del objeto Factura
var factura = {
  empresa: {
    nombre:    "Nombre de la empresa",
    direccion: "Dirección de la empresa",
    telefono:  "900900900",
    nif:       ""
  },
  cliente: {
    nombre:    "Nombre del cliente",
    direccion: "Dirección del cliente",
    telefono:  "600600600",
    nif:       "XXXXXXXXX"
  },
  elementos: [
    { descripcion: "Producto 1", cantidad: 0, precio: 0 },
    { descripcion: "Producto 2", cantidad: 0, precio: 0 },
    { descripcion: "Producto 3", cantidad: 0, precio: 0 }
  ],
  informacion: {
    baseImponible:  0,
    iva:            16,
    total:          0,
    formaPago:      "contado"
  }
};

// Métodos de cálculo del total y de visualización del total
factura.calculaTotal = function() {
  for(var i=0; i<this.elementos.length; i++) {
    this.informacion.baseImponible += this.elementos[i].cantidad *
this.elementos[i].precio;
  }
  this.informacion.total = this.informacion.baseImponible * this.informacion.iva;
}

factura.muestraTotal = function() {
  this.calculaTotal();
  alert("TOTAL = " + this.informacion.total + " euros");
}
```

```
factura.muestraTotal();
</script>
</head>

<body>
</body>
</html>
```

14.2. Ejercicio 2

```
<!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>Ejercicio 2 - Clases</title>
<script type="text/javascript">
// Definición de la clase Cliente
function Cliente(nombre, direccion, telefono, nif) {
  this.nombre = nombre;
  this.direccion = direccion;
  this.telefono = telefono;
  this.nif = nif;
}

// Definición de la clase Elemento
function Elemento(descripcion, cantidad, precio) {
  this.descripcion = descripcion;
  this.cantidad = cantidad;
  this.precio = precio;
}

// Definición de la clase Factura
function Factura(cliente, elementos) {
  this.cliente = cliente;
  this.elementos = elementos;
  this.informacion = {
    baseImponible:  0,
    iva:            16,
    total:          0,
    formaPago:      "contado"
  };
};

// La información de la empresa que emite la factura se
// añade al prototype porque se supone que no cambia
Factura.prototype.empresa = {
  nombre:    "Nombre de la empresa",
  direccion: "Direccion de la empresa",
  telefono:  "900900900",
  nif:       "XXXXXXXX"
};

// Métodos añadidos al prototype de la Factura
Factura.prototype.calculaTotal = function() {
```

```
  for(var i=0; i<this.elementos.length; i++) {
    this.informacion.baseImponible += this.elementos[i].cantidad *
this.elementos[i].precio;
  }
  this.informacion.total = this.informacion.baseImponible * this.informacion.iva;
}

Factura.prototype.muestraTotal = function() {
  this.calculaTotal();
  alert("TOTAL = " + this.informacion.total + " euros");
}


// Creación de una factura
var elCliente = new Cliente("Cliente 1", "", "", "");
var losElementos = [new Elemento("elemento1", "1", "5"),
                    new Elemento("elemento2", "2", "12"),
                    new Elemento("elemento3", "3", "42")
                   ];
var laFactura = new Factura(elCliente, losElementos);
laFactura.muestraTotal();
</script>
</head>

<body>
</body>
</html>
```

14.3. Ejercicio 3

```
<!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>Ejercicio 3 - Prototype</title>
<script type="text/javascript">
// Funcion que añade elementos al final del array
Array.prototype.anadir = function(elemento) {
  this[this.length] = elemento;
}

var array1 = [0, 1, 2];
array1.anadir(2);
alert(array1);

// Funcion que añade elementos al final del array y
//  permite evitar añadir elementos duplicados
Array.prototype.contiene = function(elemento) {
  for(var i=0; i<this.length; i++) {
          // Es importante utilizar el operador === para comprobar
    // que los elementos sean exactamente iguales
    if(this[i] === elemento) {
      return true;
    }
```

```
  }
  return false;
}

Array.prototype.anadir = function(elemento, permitirDuplicados) {
  var permitir = (permitirDuplicados == null) ? true : permitirDuplicados;

  if (!permitir) {
    if(!(this.contiene(elemento))) {
      this[this.length] = elemento;
    }
  }
  else {
    this[this.length] = elemento;
  }
}

var array2 = [0, 1, 2];
array2.anadir(2);
alert(array2);

var array3 = [0, 1, 2];
array3.anadir(2, false);
alert(array3);
</script>
</head>

<body>
</body>
</html>
```

14.4. Ejercicio 4

```
<!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>Ejercicio 4 - Prototype</title>
<script type="text/javascript">
// Funcion que trunca la longitud de una cadena
String.prototype.truncar = function(longitud) {
  longitud = longitud || 10;
  if(this.length > longitud) {
    return this.substring(0, longitud);
  }
  else {
    return this;
  }
}

var cadena = "hola mundo";
alert(cadena.truncar(6));

// Funcion que trunca la longitud de una cadena y añade
```

```
//  un indicador de cadena truncada
String.prototype.truncar = function(longitud, indicador) {
  longitud = longitud || 10;
  indicador = indicador || '...';

  if(this.length > longitud) {
    return this.substring(0, longitud-indicador.length) + indicador;
  }
  else {
    return this;
  }
}

var cadena = "En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho
tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y
galgo corredor.";
alert(cadena.truncar(50, '... (sigue)'));
alert(cadena.truncar(50));
</script>
</head>

<body>
</body>
</html>
```

14.5. Ejercicio 5

```
<!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>Ejercicio 5 - Prototype</title>
<script type="text/javascript">
// Devuelve un array sin los elementos que coinciden con
//  el elemento que se pasa como parámetro
Array.prototype.sin = function(elemento) {
  var filtrado = [];
  for(var i=0; i<this.length; i++) {
    // Es importante utilizar el operador !== para comprobar
    // que los elementos no sean exactamente iguales
    if(this[i] !== elemento) {
      filtrado.push(this[i]);
    }
  }
  return filtrado;
}

var array1 = [1, 2, 3, 4, 5];
var filtrado = array1.sin(1);
alert(filtrado);
</script>
</head>

<body>
```

```
</body>
</html>
```

14.6. Ejercicio 6

```
<!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>Ejercicio 6 - Reflexión y Prototype</title>
<script type="text/javascript">
// Determina si un objeto implementa un método cuyo nombre
//  se pasa como parámetro
Object.prototype.implementa = function (nombreMetodo){
  return this[nombreMetodo] && this[nombreMetodo] instanceof Function;
}

var objeto1 = {
  muestraMensaje: function(){
    alert("hola mundo");
  }
};

var objeto2 = {
  a: 0,
  b: 0,
  suma: function() {
    return this.a + this.b;
  }
};

alert(objeto1.implementa("muestraMensaje"));
alert(objeto1.implementa("suma"));
alert(objeto2.implementa("muestraMensaje"));
alert(objeto2.implementa("suma"));
alert(objeto1.implementa("implementa"));
// Un objeto vacío también debería incluir el método "implementa"
alert({}.implementa('implementa'));
</script>
</head>

<body>
</body>
</html>
```

14.7. Ejercicio 7

```
<!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>Ejercicio 7 - DOM</title>
<script type="text/javascript">
```

```
window.onload = function() {
  // Numero de enlaces de la pagina
  var enlaces = document.getElementsByTagName("a");
  alert("Numero de enlaces = "+enlaces.length);

  // Direccion del penultimo enlace
  var penultimo = enlaces[enlaces.length-2];
  alert("El penultimo enlace apunta a: "+penultimo.getAttribute('href'));

  // Numero de enlaces que apuntan a http://prueba
  var contador = 0;
  for(var i=0; i<enlaces.length; i++) {
    if(enlaces[i].getAttribute('href') == "http://prueba") {
      contador++;
    }
  }
  alert(contador + " enlaces apuntan a http://prueba");

  // Numero de enlaces del tercer párrafo
  var parrafos = document.getElementsByTagName("p");
  enlaces = parrafos[2].getElementsByTagName("a");
  alert("Numero de enlaces del tercer párrafo = "+enlaces.length);
}
</script>
</head>

<body>
<p>Lorem ipsum dolor sit amet, <a href="http://prueba">consectetuer adipiscing
elit</a>. Sed mattis enim vitae orci. Phasellus libero. Maecenas nisl arcu, consequat
congue, commodo nec, commodo ultricies, turpis. Quisque sapien nunc, posuere vitae,
rutrum et, luctus at, pede. Pellentesque massa ante, ornare id, aliquam vitae, ultrices
porttitor, pede. Nullam sit amet nisl elementum elit convallis malesuada. Phasellus
magna sem, semper quis, faucibus ut, rhoncus non, mi. <a href="http://prueba2">Fusce
porta</a>. Duis pellentesque, felis eu adipiscing ullamcorper, odio urna consequat
arcu, at posuere ante quam non dolor. Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Duis scelerisque. Donec lacus neque, vehicula in, eleifend vitae,
venenatis ac, felis. Donec arcu. Nam sed tortor nec ipsum aliquam ullamcorper. Duis
accumsan metus eu urna. Aenean vitae enim. Integer lacus. Vestibulum venenatis erat eu
odio. Praesent id metus.</p>

<p>Aenean at nisl. Maecenas egestas dapibus odio. Vestibulum ante ipsum primis in
faucibus orci luctus et ultrices posuere cubilia Curae; Proin consequat auctor diam. <a
href="http://prueba">Ut bibendum blandit est</a>. Curabitur vestibulum. Nunc malesuada
porttitor sapien. Aenean a lacus et metus venenatis porta. Suspendisse cursus, sem non
dapibus tincidunt, lorem magna porttitor felis, id sodales dolor dolor sed urna. Sed
rutrum nulla vitae tellus. Sed quis eros nec lectus tempor lacinia. Aliquam nec lectus
nec neque aliquet dictum. Etiam <a href="http://prueba3">consequat sem quis massa</a>.
Donec aliquam euismod diam. In magna massa, mattis id, pellentesque sit amet, porta sit
amet, lectus. Curabitur posuere. Aliquam in elit. Fusce condimentum, arcu in
scelerisque lobortis, ante arcu scelerisque mi, at cursus mi risus sed tellus.</p>

<p>Donec sagittis, nibh nec ullamcorper tristique, pede velit feugiat massa, at
sollicitudin justo tellus vitae justo. Vestibulum aliquet, nulla sit amet imperdiet
suscipit, nunc erat laoreet est, a <a href="http://prueba">aliquam leo odio sed
sem</a>. Quisque eget eros vehicula diam euismod tristique. Ut dui. Donec in metus sed
```

```
risus laoreet sollicitudin. Proin et nisi non arcu sodales hendrerit. In sem. Cras id
augue eu lorem dictum interdum. Donec pretium. Proin <a
href="http://prueba4">egestas</a> adipiscing ligula. Duis iaculis laoreet turpis.
Mauris mollis est sit amet diam. Curabitur hendrerit, eros quis malesuada tristique,
ipsum odio euismod tortor, a vestibulum nisl mi at odio. <a href="http://prueba5">Sed
non lectus non est pellentesque</a> auctor.</p>
</body>
</html>
```

14.8. Ejercicio 8

```
<!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>Ejercicio 8 - DOM</title>

<script type="text/javascript">
window.onload = function() {
  // Todos los enlaces deben cambiarse por un protocolo más seguro: "https://"
  var enlaces = document.getElementsByTagName("a");
  for(var i=0; i<enlaces.length; i++) {
    var href_anterior = enlaces[i].getAttribute('href');
    var nuevo_href = href_anterior.replace('http://', "https://");
    // También se puede utilizar la función split()
    // var trozos = href_anterior.split('://');
    // var nuevo_href = 'https://' + trozos[1];
    enlaces[i].setAttribute('href', nuevo_href);
  }

  // Los parrafos cuyo class="importante" se modifica por class="resaltado"
  // y el resto de parrafos se les añade class="normal"
  // Solución al problema de setAttribute() en Internet Explorer extraída de
  // http://www.digitalmediaminute.com/article/1394/
the-browser-dom-and-the-class-attribute
  var nombreAtributoClass = document.all ? 'className' : 'class';

  var parrafos = document.getElementsByTagName("p");
  for(var i=0; i<parrafos.length; i++) {
    if(parrafos[i].getAttribute(nombreAtributoClass) == 'importante') {
      parrafos[i].setAttribute(nombreAtributoClass, "resaltado");
    }
    else {
      parrafos[i].setAttribute(nombreAtributoClass, "normal");
    }
  }

  // Enlaces cuyo class="importante", se les añade el atributo "name" con un valor
  // autogenerado que sea igual a importante+i, donde i empieza en 0
  var enlaces = document.getElementsByTagName("a");
  var j=0;
  for(var i=0; i<enlaces.length; i++) {
    if(enlaces[i].getAttribute(nombreAtributoClass) == 'importante') {
      enlaces[i].setAttribute('name', 'importante'+j);
```

```
      j++;
    }
  }
}
</script>
</head>

<body>
<p>Lorem ipsum dolor sit amet, <a class="importante"
href="http://www.prueba.com">consectetuer adipiscing elit</a>. Sed mattis enim vitae
orci. Phasellus libero. Maecenas nisl arcu, consequat congue, commodo nec, commodo
ultricies, turpis. Quisque sapien nunc, posuere vitae, rutrum et, luctus at, pede.
Pellentesque massa ante, ornare id, aliquam vitae, ultrices porttitor, pede. Nullam sit
amet nisl elementum elit convallis malesuada. Phasellus magna sem, semper quis,
faucibus ut, rhoncus non, mi. <a class="importante" href="http://prueba2">Fusce
porta</a>. Duis pellentesque, felis eu adipiscing ullamcorper, odio urna consequat
arcu, at posuere ante quam non dolor. Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Duis scelerisque. Donec lacus neque, vehicula in, eleifend vitae,
venenatis ac, felis. Donec arcu. Nam sed tortor nec ipsum aliquam ullamcorper. Duis
accumsan metus eu urna. Aenean vitae enim. Integer lacus. Vestibulum venenatis erat eu
odio. Praesent id metus.</p>

<p class="importante">Aenean at nisl. Maecenas egestas dapibus odio. Vestibulum ante
ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin consequat
auctor diam. <a href="http://prueba">Ut bibendum blandit est</a>. Curabitur vestibulum.
Nunc malesuada porttitor sapien. Aenean a lacus et metus venenatis porta. Suspendisse
cursus, sem non dapibus tincidunt, lorem magna porttitor felis, id sodales dolor dolor
sed urna. Sed rutrum nulla vitae tellus. Sed quis eros nec lectus tempor lacinia.
Aliquam nec lectus nec neque aliquet dictum. Etiam <a href="http://prueba3">consequat
sem quis massa</a>. Donec aliquam euismod diam. In magna massa, mattis id, pellentesque
sit amet, porta sit amet, lectus. Curabitur posuere. Aliquam in elit. Fusce
condimentum, arcu in scelerisque lobortis, ante arcu scelerisque mi, at cursus mi risus
sed tellus.</p>

<p>Donec sagittis, nibh nec ullamcorper tristique, <span class="importante">pede velit
feugiat massa</span>, at sollicitudin justo tellus vitae justo. Vestibulum aliquet,
nulla sit amet imperdiet suscipit, nunc erat laoreet est, a <a
href="http://prueba">aliquam leo odio sed sem</a>. Quisque eget eros vehicula diam
euismod tristique. Ut dui. Donec in metus sed risus laoreet sollicitudin. Proin et nisi
non arcu sodales hendrerit. In sem. Cras id augue eu lorem dictum interdum. Donec
pretium. Proin <a href="http://prueba4">egestas</a> adipiscing ligula. Duis iaculis
laoreet turpis. Mauris mollis est sit amet diam. Curabitur hendrerit, eros quis
malesuada tristique, ipsum odio euismod tortor, a vestibulum nisl mi at odio. <a
class="importante" href="http://prueba5">Sed non lectus non est pellentesque</a>
auctor.</p>
</body>
</html>
```

14.9. Ejercicio 9

```
<!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>Ejercicio 9 - Crear, eliminar y modidicar nodos DOM</title>
<script type="text/javascript">
function genera() {
  // Generar números aleatorios entre 0 y 10
  var numero1 = (Math.random()*10).toFixed();
  var numero2 = (Math.random()*10).toFixed();

  // Obtener los nodos padre de los párrafos
  var primero = document.getElementById("primero");
  var segundo = document.getElementById("segundo");

  // Obtener el cada uno de los párrafos
  var parrafoAnterior1 = primero.firstChild;
  var parrafoAnterior2 = segundo.firstChild;

  // Crear el nuevo párrafo
  var parrafo1 = document.createElement("p");
  var texto1 = document.createTextNode(numero1);
  parrafo1.appendChild(texto1);
  // Si ya existía un párrafo, sustituirlo. Si no, añadirlo
  if(parrafoAnterior1 != null) {
    primero.replaceChild(parrafo1, parrafoAnterior1);
  }
  else {
    primero.appendChild(parrafo1);
  }

  // Crear el otro nuevo párrafo
  var parrafo2 = document.createElement("p");
  var texto2 = document.createTextNode(numero2);
  parrafo2.appendChild(texto2);
  // Si ya existía un párrafo, sustituirlo. Si no, añadirlo
  if(parrafoAnterior2 != null) {
    segundo.replaceChild(parrafo2, parrafoAnterior2);
  }
  else {
    segundo.appendChild(parrafo2);
  }
}

function compara() {
  // Obtener los nodos padre de los párrafos
  var primero = document.getElementById("primero");
  var segundo = document.getElementById("segundo");

  // Obtener los párrafos (existen varios métodos...)
  var parrafo1 = primero.getElementsByTagName("p")[0];
  var parrafo2 = segundo.firstChild;

  // Obtener los números a través del nodo Text de cada
  //  nodo de tipo Element de los parrafos
  var numero1 = parseInt(parrafo1.firstChild.nodeValue);
  var numero2 = parseInt(parrafo2.firstChild.nodeValue);

  // Determinar el nodo del párrafo cuyo nodo es mayor
```

```
  var parrafoMayor = (numero1 > numero2)? parrafo1 : parrafo2;

  // Obtener el nodo padre del párrafo resultado
  var resultado = document.getElementById("resultado");

  var parrafoAnterior = resultado.firstChild;
  // Si ya existía un párrafo de resultado anterior, sustituirlo. Si no, añadirlo
  if(parrafoAnterior != null) {
    resultado.replaceChild(parrafoMayor, parrafoAnterior);
  }
  else {
    resultado.appendChild(parrafoMayor);
  }
}
</script>

<style type="text/css">
#primero, #segundo, #resultado {width: 150px; height: 150px; border: thin solid silver;
background: #F5F5F5; float: left; margin:20px; font-size: 6em; color: #333; text-align:
center; padding: 5px; font-family:Arial, Helvetica, sans-serif;}
#primero p, #segundo p, #resultado p {margin:.2em 0;}
#resultado {margin-left:1.3em; border-color: black;}
.clear {clear:both;}
#compara {margin-left:11em;}
#genera {font-size:1.2em; margin-left:8em;}
</style>
</head>

<body>

<input id="genera" type="button" value="  Genera !!" onclick="genera()" />

<div id="primero"></div>

<div id="segundo"></div>

<div class="clear"></div>

<input id="compara" type="button" value="<< Comparar >>" onclick="compara()" />

<div id="resultado"></div>

</body>
</html>
```

14.10. Ejercicio 10

```
<!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>Ejercicio 10 - DOM básico y atributos XHTML</title>

<script type="text/javascript">
```

```
function muestraOculta() {
  // Obtener el ID del elemento
  var id = this.id;
  id = id.split('_');
  id = id[1];

  var elemento = document.getElementById('contenidos_'+id);
  var enlace = document.getElementById('enlace_'+id);

  if(elemento.style.display == "" || elemento.style.display == "block") {
    elemento.style.display = "none";
    enlace.innerHTML = 'Mostrar contenidos';
  }
  else {
    elemento.style.display = "block";
    enlace.innerHTML = 'Ocultar contenidos';
  }
}

window.onload = function() {
  document.getElementById('enlace_1').onclick = muestraOculta;
  document.getElementById('enlace_2').onclick = muestraOculta;
  document.getElementById('enlace_3').onclick = muestraOculta;
}
</script>
</head>

<body>

<p id="contenidos_1">[1] Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Sed
mattis enim vitae orci. Phasellus libero. Maecenas nisl arcu, consequat congue, commodo
nec, commodo ultricies, turpis. Quisque sapien nunc, posuere vitae, rutrum et, luctus
at, pede. Pellentesque massa ante, ornare id, aliquam vitae, ultrices porttitor, pede.
Nullam sit amet nisl elementum elit convallis malesuada. Phasellus magna sem, semper
quis, faucibus ut, rhoncus non, mi. Duis pellentesque, felis eu adipiscing ullamcorper,
odio urna consequat arcu, at posuere ante quam non dolor. Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Duis scelerisque.</p>
<a id="enlace_1" href="#">Ocultar contenidos</a>

<br/>

<p id="contenidos_2">[2] Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Sed
mattis enim vitae orci. Phasellus libero. Maecenas nisl arcu, consequat congue, commodo
nec, commodo ultricies, turpis. Quisque sapien nunc, posuere vitae, rutrum et, luctus
at, pede. Pellentesque massa ante, ornare id, aliquam vitae, ultrices porttitor, pede.
Nullam sit amet nisl elementum elit convallis malesuada. Phasellus magna sem, semper
quis, faucibus ut, rhoncus non, mi. Duis pellentesque, felis eu adipiscing ullamcorper,
odio urna consequat arcu, at posuere ante quam non dolor. Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Duis scelerisque.</p>
<a id="enlace_2" href="#">Ocultar contenidos</a>

<br/>
<p id="contenidos_3">[3] Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Sed
mattis enim vitae orci. Phasellus libero. Maecenas nisl arcu, consequat congue, commodo
nec, commodo ultricies, turpis. Quisque sapien nunc, posuere vitae, rutrum et, luctus
```

```
at, pede. Pellentesque massa ante, ornare id, aliquam vitae, ultrices porttitor, pede.
Nullam sit amet nisl elementum elit convallis malesuada. Phasellus magna sem, semper
quis, faucibus ut, rhoncus non, mi. Duis pellentesque, felis eu adipiscing ullamcorper,
odio urna consequat arcu, at posuere ante quam non dolor. Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Duis scelerisque.</p>
<a id="enlace_3" href="#">Ocultar contenidos</a>

</body>
</html>
```

14.11. Ejercicio 11

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Ejercicio 11 - Estados de la petición AJAX</title>
<style type="text/css">
  body { font: 13px Arial, Helvetica, sans-serif; }
  h2 { margin-bottom: 0; font-size: 1.2em; }
  #recurso, #enviar { padding: .3em; font-size: 1.2em; }
  #principal { float: left; width: 70%; }
  #secundario { float: right; width: 25%; }
  #contenidos, #estados, #cabeceras, #codigo {
    border: 2px solid #CCC;
    background: #FAFAFA;
    padding: 1em;
    white-space: pre;
  }
  #contenidos {
    min-height: 400px;
    max-height: 600px;
    overflow: scroll;
  }
  #estados { min-height: 200px; }
  #cabeceras { min-height: 200px; }
  #codigo { min-height: 100px; font-size: 1.5em; }
</style>
<script type="text/javascript">
  String.prototype.transformaCaracteresEspeciales = function() {
    return unescape(escape(this).
                    replace(/%0A/g, '<br/>').
                    replace(/%3C/g, '&lt;').
                    replace(/%3E/g, '&gt;'));
  }

  var estadosPosibles = ['No inicializado', 'Cargando', 'Cargado', 'Interactivo',
'Completado'];
  var tiempoInicial = 0;

  window.onload = function() {
    // Cargar en el input text la URL de la página
    var recurso = document.getElementById('recurso');
    recurso.value = location.href;
```

```
    // Cargar el recurso solicitado cuando se pulse el botón MOSTRAR CONTENIDOS
    document.getElementById('enviar').onclick = cargaContenido;
  }

  function cargaContenido() {
    // Borrar datos anteriores
    document.getElementById('contenidos').innerHTML = "";
    document.getElementById('estados').innerHTML = "";

    // Instanciar objeto XMLHttpRequest
    if(window.XMLHttpRequest) {
      peticion = new XMLHttpRequest();
    }
    else {
      peticion = new ActiveXObject("Microsoft.XMLHTTP");
    }

    // Preparar función de respuesta
    peticion.onreadystatechange = muestraContenido;

    // Realizar petición
    tiempoInicial = new Date();
    var recurso = document.getElementById('recurso').value;
    peticion.open('GET', recurso+'?nocache='+Math.random(), true);
    peticion.send(null);
  }

  // Función de respuesta
  function muestraContenido() {
    var tiempoFinal = new Date();
    var milisegundos = tiempoFinal - tiempoInicial;

    var estados = document.getElementById('estados');
    estados.innerHTML += "[" + milisegundos + " mseg.] " +
estadosPosibles[peticion.readyState] + "<br/>";

    if(peticion.readyState == 4) {
      if(peticion.status == 200) {
        var contenidos = document.getElementById('contenidos');
        contenidos.innerHTML = peticion.responseText.transformaCaracteresEspeciales();
      }
      muestraCabeceras();
      muestraCodigoEstado();
    }
  }

  function muestraCabeceras() {
    var cabeceras = document.getElementById('cabeceras');
    cabeceras.innerHTML =
peticion.getAllResponseHeaders().transformaCaracteresEspeciales();
  }

  function muestraCodigoEstado() {
    var codigo = document.getElementById('codigo');
```

```
    codigo.innerHTML = peticion.status + "<br/>" + peticion.statusText;
  }
</script>
</head>
<body>
<form action="#">
  URL: <input type="text" id="recurso" size="70" />
  <input type="button" id="enviar" value="Mostrar contenidos" />
</form>

<div id="principal">
  <h2>Contenidos del archivo:</h2>
  <div id="contenidos"></div>

  <h2>Cabeceras HTTP de la respuesta del servidor:</h2>
  <div id="cabeceras"></div>
</div>

<div id="secundario">
  <h2>Estados de la petición:</h2>
  <div id="estados"></div>

  <h2>Código de estado:</h2>
  <div id="codigo"></div>
</div>
</body>
</html>
```

14.12. Ejercicio 12

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Ejercicio 12 - Actualización periódica de contenidos</title>
<style type="text/css">
body { margin: 0; }
#contenidos { padding: 1em; }
#ticker {
  height: 20px;
  padding: .3em;
  border-bottom: 1px solid #CCC;
  background: #FAFAFA;
  font-family: Arial, Helvetica, sans-serif;
}
#ticker strong { margin-right: 1em; }
#acciones {
  position: absolute;
  top: 3px;
  right: 3px;
}
</style>
<script type="text/javascript">
Number.prototype.toString = function(){
```

```
  if (this < 10) {
    return '0' + this;
  }
  else {
    return this;
  }
}

var peticion = null;
var intervalo = null;
var noticias = [];
var numeroElemento = null;

window.onload = function(){
  intervalo = setInterval(descargaNoticia, 1000);
  document.getElementById('detener').onclick = detener;
  document.getElementById('anterior').onclick = anterior;
  document.getElementById('siguiente').onclick = siguiente;
}

function descargaNoticia(){
  if (peticion == null) {
    if (window.XMLHttpRequest) {
      peticion = new XMLHttpRequest();
    }
    else {
      peticion = new ActiveXObject("Microsoft.XMLHTTP");
    }
  }
  else {
    peticion.abort();
  }

  peticion.onreadystatechange = procesaNoticia;

  peticion.open('GET', 'http://localhost/RUTA_HASTA_ARCHIVO/
generaContenidos.php'+'?nocache='+Math.random(), true);
  peticion.send(null);
}

function procesaNoticia(){
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
      var fechaHora = new Date();
      var hora = fechaHora.getHours().toString() + ":" +
fechaHora.getMinutes().toString() + ":" + fechaHora.getSeconds().toString();
      noticias.push({
        hora: hora,
        titular: peticion.responseText
      });
      muestraNoticia(noticias[noticias.length - 1]);
    }
  }
}
```

```
function detener(){
  clearInterval(intervalo);
  this.value = 'Iniciar';
  this.onclick = iniciar;
}

function iniciar(){
  intervalo = setInterval(descargaNoticia, 1000);
  this.value = 'Detener';
  this.onclick = detener;

  numeroElemento = null;
}

function anterior(){
  var detener = document.getElementById('detener');
  clearInterval(intervalo);
  detener.value = 'Iniciar';
  detener.onclick = iniciar;

  if (numeroElemento == null) {
    numeroElemento = noticias.length - 1;
  }

  if (numeroElemento > 0) {
    numeroElemento--;
  }

  var noticia = noticias[numeroElemento];
  muestraNoticia(noticia);
}

function siguiente(){
  var detener = document.getElementById('detener');
  clearInterval(intervalo);
  detener.value = 'Iniciar';
  detener.onclick = iniciar;

  if (numeroElemento == null) {
    numeroElemento = noticias.length - 1;
  }

  if (numeroElemento < noticias.length - 1) {
    numeroElemento++;
  }

  var noticia = noticias[numeroElemento];
  muestraNoticia(noticia);
}

function muestraNoticia(noticia){
  var ticker = document.getElementById('ticker');
  ticker.innerHTML = "<strong>" + noticia.hora + "</strong> " + noticia.titular;
  ticker.style.backgroundColor = '#FFFF99';
  setTimeout(limpiaTicker, 300);
```

```
}

function limpiaTicker(){
  var ticker = document.getElementById('ticker');
  ticker.style.backgroundColor = '#FAFAFA';
}
</script>
</head>
<body>
<div id="ticker"></div>

<div id="acciones">
  <input type="button" id="detener" value="Detener"/>
  <input type="button" id="anterior" value="&laquo; Anterior" />
  <input type="button" id="siguiente" value="Siguiente &raquo;" />
</div>

<div id="contenidos">
<h1>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</h1>
<p>Proin tristique condimentum sem. Fusce lorem sem, laoreet nec, laoreet et, venenatis
nec, ligula.
Nunc dictum sodales lorem. Fusce turpis. Nullam semper, ipsum ut ultrices mattis, nulla
magna luctus
purus, sit amet vehicula magna magna vel velit.</p>

<h2>Morbi a lacus. Proin pharetra nisi id est.</h2>

<p>Maecenas mollis suscipit sapien. Pellentesque blandit dui eu mauris. Etiam elit
urna, iaculis non,
dignissim in, fermentum nec, ipsum. Nulla commodo aliquam lectus. Sed vulputate diam ac
sapien accumsan
consequat. Aliquam id urna sed dolor tincidunt tempor.</p>

<h2>Quisque consequat. Nullam vel justo.</h2>

<p>Cras sit amet elit a mauris ultricies viverra. Phasellus placerat quam et magna.
Nunc sed tellus.
Pellentesque hendrerit pellentesque nunc. Aenean turpis. Sed justo tellus, mollis at,
euismod at,
pellentesque eu, tellus. Nam vulputate. Nunc porttitor sapien tristique velit.
Vestibulum tempus,
quam non dapibus pellentesque, sem nulla sagittis ligula, et volutpat turpis felis
vitae nunc.</p>

<p>Ut eros magna, congue in, sodales ac, facilisis ac, dolor. Aenean faucibus
pellentesque est. Proin
cursus. Vivamus mollis enim in magna. Donec urna risus, convallis eget, aliquet non,
auctor sit amet, leo.
Duis tellus purus, pharetra in, cursus sed, posuere semper, lorem. Fusce eget velit nec
felis tempus
gravida. Donec et augue vitae nulla posuere hendrerit. Nulla vehicula scelerisque
massa. Phasellus eget
lorem id quam molestie ultrices. Integer ac ligula sit amet lectus condimentum euismod.
Sed malesuada
orci eu neque.</p>
```

```
</div>
</body>
</html>
```

14.13. Ejercicio 13

```
<!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>Ejercicio 13 - Comprobar disponibilidad del login</title>

<script type="text/javascript">
var READY_STATE_COMPLETE=4;
var peticion_http = null;

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function comprobar() {
  var login = document.getElementById("login").value;
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/ajax/compruebaDisponibilidad.php",
true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send("login="+login+"&nocache="+Math.random());
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if (peticion_http.status == 200) {
      var login = document.getElementById("login").value;
      if(peticion_http.responseText == "si") {
        document.getElementById("disponibilidad").innerHTML = "El nombre elegido
["+login+"] está disponible";
      }
      else {
        document.getElementById("disponibilidad").innerHTML = "NO está disponible el
nombre elegido ["+login+"]";
      }
    }
  }
}

window.onload = function() {
```

```
  document.getElementById("comprobar").onclick = comprobar;
}

</script>
</head>

<body>
<h1>Comprobar disponibilidad del login</h1>
<form>
  <label for="login">Nombre de usuario:</label>
  <input type="text" name="login" id="login" />
  <a id="comprobar" href="#">Comprobar disponibilidad...</a>
</form>

<div id="disponibilidad"></div>

</body>
</html>
```

14.14. Ejercicio 14

```
<!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>Ejercicio 14 - Comprobar disponibilidad del login y mostrar alternativas</title>
<script type="text/javascript">
var READY_STATE_COMPLETE=4;
var peticion_http = null;

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function comprobar() {
  var login = document.getElementById("login").value;
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/ajax/compruebaDisponibilidadXML.php",
true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send("login="+login+"&nocache="+Math.random());
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if (peticion_http.status == 200) {
```

```
      var login = document.getElementById("login").value;
      var documento_xml = peticion_http.responseXML;
      var raiz = documento_xml.getElementsByTagName("respuesta")[0];

      var disponible = raiz.getElementsByTagName("disponible")[0].firstChild.nodeValue;

      if(disponible == "si") {
        document.getElementById("disponibilidad").innerHTML = "El nombre elegido
["+login+"] está disponible";
      }
      else {
        var mensaje = "NO está disponible el nombre elegido ["+login+"]. Puedes probar
con las siguientes alternativas.";
        var alternativas = raiz.getElementsByTagName("alternativas")[0];
        var logins = alternativas.getElementsByTagName("login");
        mensaje += "<ul>";
        for(var i=0; i<logins.length; i++) {
          mensaje += "<li><a href=\"#\"
onclick=\"selecciona('"+logins[i].firstChild.nodeValue+"'); return
false\">"+logins[i].firstChild.nodeValue+"<\/a><\/li>";
        }
        mensaje += "<\/ul>";
        document.getElementById("disponibilidad").innerHTML = mensaje;
      }
    }
  }
}

function selecciona(login) {
  var cuadroLogin = document.getElementById("login");
  cuadroLogin.value = login;
}

window.onload = function() {
  document.getElementById("comprobar").onclick = comprobar;
}

</script>
</head>

<body>
<h1>Comprobar disponibilidad del login y mostrar alternativas</h1>
<form>
  <label for="login">Nombre de usuario:</label>
  <input type="text" name="login" id="login" />
  <a id="comprobar" href="#">Comprobar disponibilidad...</a>
</form>
<div id="disponibilidad"></div>
</body>
</html>
```

14.15. Ejercicio 15

```
<!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>Ejercicio 15 - Comprobar disponibilidad del login y mostrar alternativas</title>
<script type="text/javascript">
var READY_STATE_COMPLETE=4;
var peticion_http = null;

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function comprobar() {
  var login = document.getElementById("login").value;
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/ajax/compruebaDisponibilidadJSON.php",
true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send("login="+login+"&nocache="+Math.random());
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if (peticion_http.status == 200) {
      var login = document.getElementById("login").value;
      var respuesta_json = peticion_http.responseText;
      var respuesta = eval("("+respuesta_json+")");

      if(respuesta.disponible == "si") {
        document.getElementById("disponibilidad").innerHTML = "El nombre elegido
["+login+"] está disponible";
      }
      else {
        var mensaje = "NO está disponible el nombre elegido ["+login+"]. Puedes probar
con las siguientes alternativas.";
        mensaje += "<ul>";
        for(var i in respuesta.alternativas) {
          mensaje += "<li><a href=\"#\"
onclick=\"selecciona('"+respuesta.alternativas[i]+"'); return
false\">"+respuesta.alternativas[i]+"<\/a><\/li>";
        }
        mensaje += "<\/ul>";
        document.getElementById("disponibilidad").innerHTML = mensaje;
      }
    }
  }
}
```

```
function selecciona(login) {
  var cuadroLogin = document.getElementById("login");
  cuadroLogin.value = login;
}

window.onload = function() {
  document.getElementById("comprobar").onclick = comprobar;
}

</script>
</head>

<body>
<h1>Comprobar disponibilidad del login y mostrar alternativas</h1>
<form>
  <label for="login">Nombre de usuario:</label>
  <input type="text" name="login" id="login" />
  <a id="comprobar" href="#">Comprobar disponibilidad...</a>
</form>
<div id="disponibilidad"></div>
</body>
</html>
```

14.16. Ejercicio 16

```
<!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>Ejercicio 16 - Listas desplegables encadenadas</title>

<script type="text/javascript">
var peticion = null;

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function muestraProvincias() {
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
      var lista = document.getElementById("provincia");
      var documento_xml = peticion.responseXML;

      var provincias = documento_xml.getElementsByTagName("provincias")[0];
      var lasProvincias = provincias.getElementsByTagName("provincia");
      lista.options[0] = new Option("- selecciona -");

      // Método 1: Crear elementos Option() y añadirlos a la lista
```

```
      for(i=0; i<lasProvincias.length; i++) {
        var codigo =
lasProvincias[i].getElementsByTagName("codigo")[0].firstChild.nodeValue;
        var nombre =
lasProvincias[i].getElementsByTagName("nombre")[0].firstChild.nodeValue;
        lista.options[i+1] = new Option(nombre, codigo);
      }

      // Método 2: crear el código HTML de <option value="">...</option> y utilizar el
innerHTML de la lista

      /*
      var codigo_html = "";
      codigo_html += "<option>- selecciona -<\/option>";
      for(var i=0; i<lasProvincias.length; i++) {
        var codigo =
lasProvincias[i].getElementsByTagName("codigo")[0].firstChild.nodeValue;
        var nombre =
lasProvincias[i].getElementsByTagName("nombre")[0].firstChild.nodeValue;
        codigo_html += "<option value=\""+codigo+"\">"+nombre+"<\/option>";
      }

      // La separacion siguiente se tiene que hacer por este bug de microsoft:
      // http://support.microsoft.com/default.aspx?scid=kb;en-us;276228
      var esIE = navigator.userAgent.toLowerCase().indexOf('msie')!=-1;
      if(esIE) {
        document.getElementById("provincia").outerHTML = "<select
id=\"provincia\">"+codigo_html+"</select>";
      }
      else {
        document.getElementById("provincia").innerHTML = codigo_html;
      }
      */
    }
  }
}

function cargaMunicipios() {
  var lista = document.getElementById("provincia");
  var provincia = lista.options[lista.selectedIndex].value;
  if(!isNaN(provincia)) {
    peticion = inicializa_xhr();
    if (peticion) {
      peticion.onreadystatechange = muestraMunicipios;
      peticion.open("POST", "http://localhost/RUTA_HASTA_ARCHIVO/
cargaMunicipiosXML.php?nocache=" + Math.random(), true);
      peticion.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
      peticion.send("provincia=" + provincia);
    }
  }
}

function muestraMunicipios() {
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
```

```
      var lista = document.getElementById("municipio");
      var documento_xml = peticion.responseXML;

      var municipios = documento_xml.getElementsByTagName("municipios")[0];
      var losMunicipios = municipios.getElementsByTagName("municipio");

      // Borrar elementos anteriores
      lista.options.length = 0;

      // Se utiliza el método de crear elementos Option() y añadirlos a la lista
      for(i=0; i<losMunicipios.length; i++) {
        var codigo =
losMunicipios[i].getElementsByTagName("codigo")[0].firstChild.nodeValue;
        var nombre =
losMunicipios[i].getElementsByTagName("nombre")[0].firstChild.nodeValue;
        lista.options[i] = new Option(nombre, codigo);
      }
    }
  }
}

window.onload = function() {
  peticion = inicializa_xhr();
  if(peticion) {
    peticion.onreadystatechange = muestraProvincias;
    peticion.open("GET", "http://localhost/RUTA_HASTA_ARCHIVO/
cargaProvinciasXML.php?nocache="+Math.random(), true);
    peticion.send(null);
  }

  document.getElementById("provincia").onchange = cargaMunicipios;
}

</script>
</head>

<body>
<h1>Listas desplegables encadenadas</h1>

<form>
  <label for="provincia">Provincia</label>
  <select id="provincia">
    <option>Cargando...</option>
  </select>
  <br/><br/>
  <label for="municipio">Municipio</label>
  <select id="municipio">
    <option>- selecciona una provincia -</option>
  </select>
</form>

</body>
</html>
```

14.17. Ejercicio 17

```
<!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>Ejercicio 17 - Listas desplegables encadenadas</title>

<script type="text/javascript">
var peticion = null;

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function muestraProvincias() {
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
      var lista = document.getElementById("provincia");
      var provincias = eval('(' + peticion.responseText + ')');

      lista.options[0] = new Option("- selecciona -");
      var i=1;
      for(var codigo in provincias) {
        lista.options[i] = new Option(provincias[codigo], codigo);
        i++;
      }
    }
  }
}

function cargaMunicipios() {
  var lista = document.getElementById("provincia");
  var provincia = lista.options[lista.selectedIndex].value;
  if(!isNaN(provincia)) {
    peticion = inicializa_xhr();
    if (peticion) {
      peticion.onreadystatechange = muestraMunicipios;
      peticion.open("POST", "http://localhost/RUTA_HASTA_ARCHIVO/
cargaMunicipiosJSON.php?nocache=" + Math.random(), true);
      peticion.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
      peticion.send("provincia=" + provincia);
    }
  }
}

function muestraMunicipios() {
  if (peticion.readyState == 4) {
    if (peticion.status == 200) {
      var lista = document.getElementById("municipio");
```

```
      var municipios = eval('(' + peticion.responseText + ')');

      lista.options.length = 0;
      var i=0;
      for(var codigo in municipios) {
        lista.options[i] = new Option(municipios[codigo], codigo);
        i++;
      }
    }
  }
}

window.onload = function() {
  peticion = inicializa_xhr();
  if(peticion) {
    peticion.onreadystatechange = muestraProvincias;
    peticion.open("GET", "http://localhost/RUTA_HASTA_ARCHIVO/
cargaProvinciasJSON.php?nocache="+Math.random(), true);
    peticion.send(null);
  }

  document.getElementById("provincia").onchange = cargaMunicipios;
}

</script>
</head>

<body>
<h1>Listas desplegables encadenadas</h1>

<form>
  <label for="provincia">Provincia</label>
  <select id="provincia">
    <option>Cargando...</option>
  </select>
  <br/><br/>
  <label for="municipio">Municipio</label>
  <select id="municipio">
    <option>- selecciona una provincia -</option>
  </select>
</form>

</body>
</html>
```

14.18. Ejercicio 18

El archivo `util.js` que utiliza el ejercicio se incluye en el archivo ZIP de la solución completa.

```
<!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" xml:lang="es" lang="es">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

```
<title>Ejercicio 18 - Teclado virtual</title>

<style type="text/css" media="screen">
body, html { padding:0; margin:0; }
p { font-family:verdana,tahoma,arial,helvetica,sans-serif; font-size:13px;
margin-left:10px; }
#keyboard {
  padding:30px 8px 8px 8px;
  margin:5px 0 0 0;
  height:300px;
  width:1000px;
  background-color:#eee;
  border-top:3px solid #ccc;
  border-right:3px solid #aaa;
  border-bottom:5px solid #888;
  border-left:3px solid #bbb;
}
kbd, .tecla {
  font-family:verdana,tahoma,arial,helvetica,sans-serif;
  font-size:13px;
  border-top:2px solid #ccc;
  border-right:5px solid #aaa;
  border-bottom:9px solid #888;
  border-left:5px solid #bbb;
  padding:3px 6px 3px 6px;
  white-space:nowrap;
  color:#000;
  background:#eee;
  width:22px;
  height:30px;
  float:left;
  overflow:hidden;
  cursor: pointer; cursor: hand;
}
.delete { padding:3px 8px 3px 4px; }
img { margin-top:2px; }
.narrow, .lock { font-family:arial,helvetica,sans-serif; font-size:9px; }
#keyboard div, .clear { clear:both; margin:0; }
.fontgap { margin-left:10px; }
.rightgap { margin-right:12px; }
#esc, #f4, #f8 { margin-right:30px; }
#row1 { margin-bottom:12px; width:990px; height:50px; }
.tab { width:48px; }
.capslock { width:62px; }
.shift-right { width:98px; margin-right:56px; }
.backspace { width:68px; }
.spacebar { width:234px; }
.numpad-0 { width:66px; }
.shift-left, #ctrl-left, #ctrl-right, #windows-left, #windows-right, #alt-left,
.alt-right, #application { width:36px; }
.enter-top { border-bottom:0; height:39px; width:42px; }
#enter-bottom { border-top:0; height:32px; width:28px; margin-right:156px; }
.numpad-top { border-bottom:0; height:10px; padding-top:32px;}
.numpad-bottom { border-top:0; height:32px; }
#arrow-up {margin-right:56px;}
```

```
#sysreq, #break { font-family:'arial narrow'; font-size:10px; position:absolute;
top:72px; left:696px; z-index:2; }
#break {left:785px;}
.lock { margin-left:30px; margin-top:25px; float:left;}
#enter-edge { position:absolute; top:183px; left:609px; z-index:2; }
#led-active, #led2, #led3 {
  position:absolute;
  top:40px;
  left:850px;
  width:6px;
  height:6px;
  margin-left:5px;
  border-top:2px solid #888;
  border-right:2px solid #bbb;
  border-bottom:2px solid #ccc;
  border-left:2px solid #aaa;
  background-color:#6f0;
}
#led2 {left:900px; background-color:#060;}
#led3 {left:950px; background-color:#060;}
.pulsada { background: #FFFF99; border-top-width:5px; border-bottom-width: 6px;}
.enter-top.pulsada {border-bottom: none; border-top-width:2px;}
#contenido, #contenidoGuardado { margin: 10px; background-color: #ededed; border: 2px
solid #a0a0a0; padding: 1em;}
</style>
<script src="util.js" type="text/javascript"></script>
<script type="text/javascript" language="javascript">
var teclados = {};
var tecladoActivo = null;
var teclasPulsadas = [];
var teclasEspeciales = ["borrado", "tabulador", "enter", "espaciadora", "mayusculas",
"shift_izquierdo", "shift_derecho", "altgr"];
var tecladoIdioma = null;
var tecladoVariante = null;
var estado = {mayusculas: false, shift: false, altgr: false};

function descargaTeclado(idioma) {
  var cargador = new net.CargadorContenidosCompleto("http://localhost/
RUTA_HASTA_EL_ARCHIVO/tecladoVirtual.php?nocache="+Math.random(),
                              function() { teclados[idioma] =
eval('('+this.req.responseText+')'); },
                              null,
                              "POST",
                              "accion=cargaTeclado&idioma="+idioma,
                              "application/x-www-form-urlencoded",
                              false);
}

function cargaTeclado() {
  // Obtener o descargar el teclado requerido
  if(teclados[tecladoIdioma] == null) {
    descargaTeclado(tecladoIdioma);
  }
  var teclado = teclados[tecladoIdioma][tecladoVariante];
```

```
  // Cargar teclas normales y añadir el evento a cada una
  for(var i=0; i<teclado.length; i++) {
    if(teclado[i] != undefined) {
      document.getElementById('tecla_'+i).innerHTML = teclado[i];
      document.getElementById('tecla_'+i).onclick = pulsaTecla;
    }
    else {
      document.getElementById('tecla_'+i).innerHTML = '';
    }
  }

  // Añadir eventos a las teclas especiales
  for(var i=0; i<teclasEspeciales.length; i++) {
    document.getElementById('tecla_especial_'+teclasEspeciales[i]).onclick =
pulsaTeclaEspecial;
  }

  tecladoActivo = teclado;
}

function pulsaTecla() {
  var teclaPulsada = this.id.replace(/tecla_/gi, "");
  var caracter = tecladoActivo[teclaPulsada];

  teclasPulsadas.push(caracter);

  // Iluminar la tecla pulsada
  this.className+=" pulsada";
  setTimeout(apagaTecla, 100);

  mostrarContenidos();
}

function apagaTecla() {
  for(var i in tecladoActivo) {
    if(tecladoActivo[i]) {
      document.getElementById('tecla_'+i).className =
document.getElementById('tecla_'+i).className.replace(/pulsada/ig, "");
    }
  }
  for(var i in teclasEspeciales) {
    if(teclasEspeciales[i]) {
      document.getElementById('tecla_especial_'+teclasEspeciales[i]).className =
document.getElementById('tecla_especial_'+teclasEspeciales[i]).className.replace(/pulsada/
ig, "");
    }
  }
}

function pulsaTeclaEspecial() {
  var teclaPulsada = this.id.replace(/tecla_especial_/gi, "");

  // Iluminar la tecla pulsada
  this.className+=" pulsada";
  setTimeout(apagaTecla, 100);
```

```
  switch(teclaPulsada) {
    case 'borrado':
      teclasPulsadas.pop();
      break;
    case 'tabulador':
      teclasPulsadas.push('\t');
      break;
    case 'enter':
      teclasPulsadas.push('\n');
      break;
    case 'espaciadora':
      teclasPulsadas.push(' ');
      break;
    case 'mayusculas':
      cargaVarianteTeclado('mayusculas');
      break;
    case 'shift_izquierdo':
      cargaVarianteTeclado('shift_izquierdo');
      break;
    case 'shift_derecho':
      cargaVarianteTeclado('shift_derecho');
      break;
    case 'altgr':
      cargaVarianteTeclado('altgr');
      break;
  }

  mostrarContenidos();
}

function cargaVarianteTeclado(variante) {
  var nombreVariante = {mayusculas: 'caps', shift_izquierdo: 'shift', shift_derecho:
'shift', altgr: 'altgr'};

  if(estado[variante] == true) {
    estado[variante] = false;
    tecladoVariante = 'normal';
  }
  else {
    estado[variante] = true;
    tecladoVariante = nombreVariante[variante];
  }
  cargaTeclado();
}

function mostrarContenidos(texto, zona) {
  var elemento = zona || "contenido";
  var contenido = texto || teclasPulsadas.join("");

  contenido = contenido.replace(/\n/gi, '<br/>');
  contenido = contenido.replace(/\t/gi, '    ');
  contenido = contenido.replace(/ /gi, ' ');

  document.getElementById(elemento).innerHTML = contenido;
```

```
}

function muestraIdiomas() {
  var respuesta = eval('('+this.req.responseText+')');
  var lista = document.getElementById("idiomas");

  var i=0;
  for(var codigo in respuesta.idiomas) {
    // Añadir idiomas a la lista desplegable
    lista.options[i] = new Option(respuesta.idiomas[codigo], codigo);
    i++;
    // Crear los objetos que almacenan los teclados de cada idioma
    teclados[codigo] = null;
  }

  // Cargar el teclado del idioma por defecto (la variante normal)
  tecladoIdioma = respuesta.defecto;
  tecladoVariante = 'normal';
  cargaTeclado();
}

function cambiaIdioma() {
  var lista = document.getElementById("idiomas");
  tecladoIdioma = lista.options[lista.selectedIndex].value;
  cargaTeclado();
}

function guarda() {
  var cargador = new net.CargadorContenidosCompleto("http://localhost/
RUTA_HASTA_EL_ARCHIVO/tecladoVirtual.php?nocache="+Math.random(),
                            function() {
mostrarContenidos(unescape(this.req.responseText), "contenidoGuardado"); },
                            null,
                            "POST",

"accion=guardar&contenido="+escape(teclasPulsadas.join("")),
                            "application/x-www-form-urlencoded");
}

window.onload = function() {
  var cargador = new net.CargadorContenidosCompleto("http://localhost/
RUTA_HASTA_EL_ARCHIVO/tecladoVirtual.php?nocache="+Math.random(),
                              muestraIdiomas,
                              null,
                              "POST",
                              "accion=listaIdiomas",
                              "application/x-www-form-urlencoded");

  // Evento necesario para cambiar el idioma del teclado
  document.getElementById("idiomas").onchange = cambiaIdioma;

  setInterval(guarda, 30 * 1000);
}

/*
```

```
Los teclados de cada idioma y algunas ideas de implementación son originales del
JavaScript Virtual Keyboard, cuya nota de copyright
se incluye a continuación:
*/
 /*
 * JavaScript Virtual Keyboard, version 2.2
 *
 * Copyright (C) 2006-2007 Dmitriy Khudorozhkov
 *
 * This software is provided "as-is", without any express or implied warranty.
 * In no event will the author be held liable for any damages arising from the
 * use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 *
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 *
 * 3. This notice may not be removed or altered from any source distribution.
 *
 *  - Dmitriy Khudorozhkov, kh_dmitry2001@mail.ru
 */

/*
El diseño del teclado es obra de Chris Hester y se puede descargar desde
http://www.designdetector.com/archives/05/03/KeyboardDemo.php
*/

</script>

</head>

<body>

<div id="led-active"> </div>
<div id="led2"> </div>
<div id="led3"> </div>


<div id="keyboard">

<div id="row1">
  <kbd id="esc">Esc</kbd><kbd>F1</kbd><kbd>F2</kbd><kbd>F3</kbd><kbd id="f4">F4</kbd>
  <kbd>F5</kbd><kbd>F6</kbd><kbd>F7</kbd><kbd
id="f8">F8</kbd><kbd>F9</kbd><kbd>F10</kbd>
  <kbd>F11</kbd><kbd class="rightgap">F12</kbd><kbd class="narrow">Impr<br>Pant</kbd>
  <kbd class="narrow">Bloq<br>Des</kbd><kbd class="narrow">Pausa</kbd>
  <span class="lock">Bloq<br>Num</span><span class="lock">Bloq<br>Mayus</span>
```

```
  <span class="lock">Bloq<br>Des</span>
</div>

<div id="row2">
  <kbd id="tecla_0"></kbd><kbd id="tecla_1"></kbd><kbd id="tecla_2"></kbd>
  <kbd id="tecla_3"></kbd><kbd id="tecla_4"></kbd><kbd id="tecla_5"></kbd>
  <kbd id="tecla_6"></kbd><kbd id="tecla_7"></kbd><kbd id="tecla_8"></kbd>
  <kbd id="tecla_9"></kbd><kbd id="tecla_10"></kbd><kbd id="tecla_11"></kbd>
  <kbd id="tecla_12"></kbd>
  <kbd id="tecla_especial_borrado" class="rightgap backspace"><img src="imagenes/
backspace.gif" alt="" height="19" width="52"></kbd>
  <kbd class="narrow">Insert</kbd><kbd class="narrow">Inicio</kbd><kbd class="narrow
rightgap">Re<br>Pag</kbd>
  <kbd class="narrow">Bloq<br>Num</kbd><kbd>/</kbd><kbd>*</kbd><kbd>-</kbd>
</div>

<div>
  <kbd id="tecla_especial_tabulador" class="tab"><img src="imagenes/tab.gif" alt=""
height="27" width="23"></kbd>
  <kbd id="tecla_13"></kbd><kbd id="tecla_14"></kbd><kbd id="tecla_15"></kbd><kbd
id="tecla_16"></kbd>
  <kbd id="tecla_17"></kbd><kbd id="tecla_18"></kbd><kbd id="tecla_19"></kbd><kbd
id="tecla_20"></kbd>
  <kbd id="tecla_21"></kbd><kbd id="tecla_22"></kbd><kbd id="tecla_23"></kbd><kbd
id="tecla_24"></kbd>
  <kbd id="tecla_especial_enter" class="rightgap enter-top"><img src="imagenes/
enter.gif" alt="" height="24" width="24"></kbd>
  <kbd class="narrow delete">Supr</kbd><kbd class="narrow">Fin</kbd><kbd class="narrow
rightgap">Av<br>Pag</kbd>
  <kbd>7<br><span class="narrow">Inicio</span></kbd><kbd>8<br><img src="imagenes/
arrow-up-small.gif" alt="" height="13" width="21"></kbd>
  <kbd>9<br><span class="narrow">RePag</span></kbd><kbd class="numpad-top">+</kbd>
</div>

<div>
  <kbd id="tecla_especial_mayusculas" class="narrow capslock">Bloq Mayus</kbd>
  <kbd id="tecla_25"></kbd><kbd id="tecla_26"></kbd><kbd id="tecla_27"></kbd><kbd
id="tecla_28"></kbd>
  <kbd id="tecla_29"></kbd><kbd id="tecla_30"></kbd><kbd id="tecla_31"></kbd><kbd
id="tecla_32"></kbd>
  <kbd id="tecla_33"></kbd><kbd id="tecla_34"></kbd><kbd id="tecla_35"></kbd><kbd
id="tecla_36"></kbd>
  <kbd id="enter-bottom"> </kbd><kbd>4<br><img src="imagenes/arrow-left-small.gif"
alt="" height="7" width="13"></kbd>
  <kbd>5</kbd><kbd>6<br><img src="imagenes/arrow-right-small.gif" alt="" height="7"
width="13"></kbd>
  <kbd class="numpad-bottom"> </kbd>
</div>

<div>
  <kbd id="tecla_especial_shift_izquierdo" class="narrow shift-left"><img src="imagenes/
shift.gif" alt="" height="20" width="17"></kbd>
  <kbd>&gt;<br>&lt;</kbd><kbd id="tecla_37"></kbd><kbd id="tecla_38"></kbd><kbd
id="tecla_39"></kbd>
  <kbd id="tecla_40"></kbd><kbd id="tecla_41"></kbd><kbd id="tecla_42"></kbd><kbd
```

```
id="tecla_43"></kbd>
  <kbd id="tecla_44"></kbd><kbd id="tecla_45"></kbd><kbd id="tecla_46"></kbd>
  <kbd id="tecla_especial_shift_derecho" class="shift-right"><img src="imagenes/
shift.gif" alt="" height="20" width="17"></kbd>
  <kbd id="arrow-up"><img src="imagenes/arrow-up.gif" alt="" height="22"
width="21"></kbd>
  <kbd>1<br><span class="narrow">Fin</span></kbd>
  <kbd>2<br><img src="imagenes/arrow-down-small.gif" alt="" height="13"
width="21"></kbd>
  <kbd>3<br><span class="narrow">AvPag</span></kbd>
  <kbd class="numpad-top narrow">Enter</kbd>
</div>

<div>
  <kbd id="ctrl-left" class="narrow">Ctrl</kbd>
  <kbd id="windows-left"><img src="imagenes/windows-key.gif" alt="Windows Key"
height="13" width="16"></kbd>
  <kbd id="alt-left" class="narrow">Alt</kbd>
  <kbd id="tecla_especial_espaciadora" class="spacebar"> </kbd>
  <kbd  id="tecla_especial_altgr" class="narrow alt-right">Alt Gr</kbd>
  <kbd id="windows-right"><img src="imagenes/windows-key.gif" alt="" height="13"
width="16"></kbd>
  <kbd id="application"><img src="imagenes/windows-application-key.gif" alt=""
height="13" width="12"></kbd>
  <kbd id="ctrl-right" class="narrow rightgap">Ctrl</kbd>
  <kbd><img src="imagenes/arrow-left.gif" alt="" height="19" width="21"></kbd>
  <kbd><img src="imagenes/arrow-down.gif" alt="" height="22" width="21"></kbd>
  <kbd class="rightgap"><img src="imagenes/arrow-right.gif" alt="" height="19"
width="21"></kbd>
  <kbd class="numpad-0" id="_0">0<br><span class="narrow">Ins</span></kbd>
  <kbd>.<br><span class="narrow">Supr</span></kbd>
  <kbd class="numpad-bottom"> </kbd>
</div>

</div>

<select id="idiomas"><option>Cargando...</option></select>

<div id="contenido"></div>

<p>Contenido en el servidor [se actualiza automáticamente]</p>
<div id="contenidoGuardado"></div>

<p style="font-size:.9em;color:#888;margin-top:2em;">&copy; El diseño del teclado es
obra de Chris Hester y se puede descargar desde http://www.designdetector.com/archives/
05/03/KeyboardDemo.php</p>
</body>

</html>
```

14.19. Ejercicio 19

```
<!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>Ejercicio 19 - Autocompletar</title>
<script type="text/javascript">
var peticion = null;
var elementoSeleccionado = -1;
var sugerencias = null;
var cacheSugerencias = {};

function inicializa_xhr() {
  if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

Array.prototype.formateaLista = function() {
  var codigoHtml = "";

  codigoHtml = "<ul>";
  for(var i=0; i<this.length; i++) {
    if(i == elementoSeleccionado) {
      codigoHtml += "<li class=\"seleccionado\">"+this[i]+"</li>";
    }
    else {
      codigoHtml += "<li>"+this[i]+"</li>";
    }
  }
  codigoHtml += "</ul>";

  return codigoHtml;
};

function autocompleta() {
  var elEvento = arguments[0] || window.event;
  var tecla = elEvento.keyCode;

  if(tecla == 40) { // Flecha Abajo
    if(elementoSeleccionado+1 < sugerencias.length) {
      elementoSeleccionado++;
    }
    muestraSugerencias();
  }
  else if(tecla == 38) { // Flecha Arriba
    if(elementoSeleccionado > 0) {
      elementoSeleccionado--;
    }
    muestraSugerencias();
  }
  else if(tecla == 13) { // ENTER o Intro
    seleccionaElemento();
  }
  else {
    var texto = document.getElementById("municipio").value;
```

```
    // Si es la tecla de borrado y el texto es vacío, ocultar la lista
    if(tecla == 8 && texto == "") {
      borraLista();
      return;
    }

    if(cacheSugerencias[texto] == null) {
      peticion = inicializa_xhr();

      peticion.onreadystatechange = function() {
        if(peticion.readyState == 4) {
          if(peticion.status == 200) {
            sugerencias = eval('('+peticion.responseText+')');
            if(sugerencias.length == 0) {
              sinResultados();
            }
            else {
              cacheSugerencias[texto] = sugerencias;
              actualizaSugerencias();
            }
          }
        }
      };

      peticion.open('POST', 'http://localhost/RUTA_HASTA_ARCHIVO/
autocompletaMunicipios.php?nocache='+Math.random(), true);
      peticion.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
      peticion.send('municipio='+encodeURIComponent(texto));
    }
    else {
      sugerencias = cacheSugerencias[texto];
      actualizaSugerencias();
    }
  }
}

function sinResultados() {
  document.getElementById("sugerencias").innerHTML = "No existen municipios que
empiecen con ese texto";
  document.getElementById("sugerencias").style.display = "block";
}

function actualizaSugerencias() {
  elementoSeleccionado = -1;
  muestraSugerencias();
}

function seleccionaElemento() {
  if(sugerencias[elementoSeleccionado]) {
    document.getElementById("municipio").value = sugerencias[elementoSeleccionado];
    borraLista();
  }
}
```

```
function muestraSugerencias() {
  var zonaSugerencias = document.getElementById("sugerencias");

  zonaSugerencias.innerHTML = sugerencias.formateaLista();
  zonaSugerencias.style.display = 'block';
}

function borraLista() {
  document.getElementById("sugerencias").innerHTML = "";
  document.getElementById("sugerencias").style.display = "none";
}

window.onload = function() {
  // Crear elemento de tipo <div> para mostrar las sugerencias del servidor
  var elDiv = document.createElement("div");
  elDiv.id = "sugerencias";
  document.body.appendChild(elDiv);

  document.getElementById("municipio").onkeyup = autocompleta;
  document.getElementById("municipio").focus();
}
</script>

<style type="text/css">
body {font-family: Arial, Helvetica, sans-serif;}
#sugerencias {width:200px; border:1px solid black; display:none; margin-left: 83px;}
#sugerencias ul {list-style: none; margin: 0; padding: 0; font-size:.85em;}
#sugerencias ul li {padding: .2em; border-bottom: 1px solid silver;}
.seleccionado {font-weight:bold; background-color: #FFFF00;}
</style>

</head>

<body>
<h1>Autocompletar texto</h1>

<form>
  <label for="municipio">Municipio</label>   
  <input type="text" id="municipio" name="municipio" size="30" />
  <input type="text" id="oculto" name="oculto" style="display:none;" />
</form>

</body>
</html>
```

14.20. Ejercicio 20

El archivo `util.js` que utiliza el ejercicio se incluye en el archivo ZIP de la solución completa.

```
<!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>Ejercicio 20 - Monitorizar</title>
```

```
<script type="text/javascript" src="util.js"></script>
<script type="text/javascript">
var nodos = [{"nombre": "Google", "url": "http://www.google.com"}, {"nombre": "Yahoo",
"url": "http://www.yahoo.com"},
            {"nombre": "MSN", "url": "http://www.msn.com"}, {"nombre": "eBay", "url":
"http://www.ebay.com"},
            {"nombre": "YouTube", "url": "http://www.youtube.com"}, {"nombre":
"Flickr", "url": "http://www.flickr.com"}];
var intervalo;

function monitoriza() {
  intervalo = setInterval(monitorizaNodos, 1000);
}

function detiene() {
  clearInterval(intervalo);
}

function monitorizaNodos() {
  for(var i=0; i<nodos.length; i++) {
    document.getElementById("nodo"+i).style.border = "3px solid #000000";
    var ping = new net.CargadorContenidosCompleto(nodos[i].url, procesaPing,
noDisponible, "HEAD");
  }
}

function procesaPing() {
  if(new Date(this.req.getResponseHeader("Date"))) {
    var numeroNodo = calculaNumeroNodo(this.url);
    document.getElementById("nodo"+numeroNodo).style.border = "3px solid #00FF00";
    document.getElementById("nodo"+numeroNodo).className = "on";
    document.getElementById("datos"+numeroNodo).innerHTML =
this.req.getResponseHeader("Server");
  }
}

function noDisponible() {
  var numeroNodo = calculaNumeroNodo(this.url);
  document.getElementById("nodo"+numeroNodo).style.border = "3px solid #FF0000";
  document.getElementById("nodo"+numeroNodo).className = "off";
}

function calculaNumeroNodo(url) {
  for(var i=0; i<nodos.length; i++) {
    if(nodos[i].url == url) {
      return i;
    }
  }
}

window.onload = function() {
  // Crear elemento de tipo <div> para mostrar cada uno de los nodos
  for(i=0; i<nodos.length; i++) {
    var nodo = document.createElement("div");
```

```
    nodo.id = "nodo"+i;
    nodo.innerHTML = "<strong>" + nodos[i].nombre + "</strong><br>" + nodos[i].url +
"<span id=\"datos"+i+"\"></span>";
    document.getElementById("mapa_red").appendChild(nodo);
    document.getElementById("nodo"+i).className = "normal";
  }

  // Establecer los eventos en los botones
  document.getElementById("monitoriza").onclick = monitoriza;
  document.getElementById("detiene").onclick = detiene;
}

</script>

<style type="text/css">
body {font-size:14px; font-family:Arial, Helvetica, sans-serif;}
.normal, .consulta, .on, .off {width: 140px; text-align: center; margin: .5em; padding:
.5em; }
form {display: inline; }
.normal {background-color: #FFFFFF; border: 3px solid #C0C0C0;}
.consulta {border:3px solid #000000;}
.on {background-color: #00CC00; border: 3px solid #00FF00;}
.off {background-color: #CC0000; border: 3px solid #FF0000;}
#mapa_red {border:5px solid #D0D0D0; float: left; padding: 1em 0; margin: 1em 0; width:
50%;}
#mapa_red div { float: left; margin: 1em; height: 5em; width: 35%;}
div span {display:block; padding:.3em;}
</style>

</head>

<body>
<h1>Consola de monitorización</h1>

<form>
  <input type="button" id="monitoriza" value="Monitorizar"></input>
  <input type="button" id="detiene" value="Detener"></input>
</form>

<div id="mapa_red"></div>

</body>
</html>
```

14.21. Ejercicio 21

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejercicio 21 - RSS</title>
<script type="text/javascript">
Object.prototype.get = function(etiqueta) {
  return this.getElementsByTagName(etiqueta)[0].textContent;
```

```
}

var rss = {
  canal: {},
  items: []
};

window.onload = function(){
  document.getElementById('mostrar').onclick = cargaRss;
}

function cargaRss(){
  // Obtener URL de RSS
  borrarLog();
  log('Averiguar la URL del canal RSS');

  var url_original = document.getElementById('url').value;
  log('URL original es ' + url_original);

  var url_rss = descubreRss(url_original);
  log('La URL descubierta es ' + url_rss);

  // Descargar canal RSS
  log('Descargando canal RSS');
  descargaRss(url_rss);
}

function descubreRss(url){
  var peticion = new XMLHttpRequest();
  peticion.onreadystatechange = function(){};
  peticion.open('GET', 'http://localhost/RUTA_HASTA_ARCHIVO/descubreRss.php?url=' +
encodeURIComponent(url), false);
  peticion.send(null);

  return peticion.responseText;
}

function descargaRss(url){
  var peticion = new XMLHttpRequest();
  peticion.onreadystatechange = procesaRss;
  peticion.open('GET', 'http://localhost/RUTA_HASTA_ARCHIVO/proxy.php?url=' +
encodeURIComponent(url) + '&ct=text/xml', true);
  peticion.send(null);

  function procesaRss(){
    if (peticion.readyState == 4) {
      if (peticion.status == 200) {
        var xml = peticion.responseXML;

        var canal = xml.getElementsByTagName('channel')[0];
        var titulo = canal.getElementsByTagName('title')[0].textContent;
        rss.canal.titulo = titulo;

        if(canal.getElementsByTagName('image').length > 0) {
          var url_imagen =
```

```
canal.getElementsByTagName('image')[0].getElementsByTagName('url')[0].textContent;
          rss.canal.titulo = '<img src="'+url_imagen+'" />'+rss.canal.titulo;
        }

        var enlace = canal.getElementsByTagName('link')[0].textContent;
        rss.canal.enlace = enlace;

        var items = xml.getElementsByTagName('item');
        for (var i = 0; i < items.length; i++) {
          var item = items[i];
          var titulo = item.get('title');
          var enlace = item.getElementsByTagName('link')[0].textContent;
          var descripcion = item.getElementsByTagName('description')[0].textContent;
          var fecha = item.getElementsByTagName('pubDate')[0].textContent;

          rss.items[i] = {
            titulo: titulo,
            enlace: enlace,
            descripcion: descripcion,
            fecha: fecha
          };
        }

        muestraRss();
      }
    }
  }
}

function muestraRss(){
  document.getElementById('noticias').style.display = 'block';
  document.getElementById('titulares').innerHTML = '';
  document.getElementById('contenidos').innerHTML = '';

  document.getElementById('titulo').innerHTML = '<a href="' + rss.canal.enlace + '">' +
rss.canal.titulo + '</a>';

  var titulares = document.getElementById('titulares');

  for (var i = 0; i < rss.items.length; i++) {
    titulares.innerHTML += '<a href="#" onclick="muestraElemento(' + i + ')">' +
rss.items[i].titulo + '</a> <br/>';
  }
}

function muestraElemento(indice){
  var item = rss.items[indice];
  var html = "";
  html += "<h1><a href=\"" + item.enlace + "\">" + item.titulo + "</a></h1>";
  if (item.fecha != undefined) {
    html += "<h2>" + item.fecha + "</h2>";
  }
  html += "<p>" + item.descripcion + "</p>";

  document.getElementById("contenidos").innerHTML = html;
```

```
}

function log(mensaje){
  document.getElementById('info').innerHTML += mensaje + "<br/>";
}

function borrarLog(){
  document.getElementById('info').innerHTML = "";
}
</script>
<style type="text/css">
body { font-family: Arial, Helvetica, sans-serif; }
form { margin: 0; }
#info { margin: 0; font-size: .7em; color: #777; }
#noticias {
  position: absolute;
  width: 80%;
  margin-top: 1em;
  border: 2px solid #369;
  padding: 0;
  display: none;
}
#titulo { background-color: #DDE8F3; padding: .3em; border-bottom: 1px solid #369; }
#titulares { width: 20%; float: left; border: none; border-right: 1px solid #D9E5F2; }
#contenidos { margin-left: 22%; padding: 0px 20px; vertical-align: top; }
#titulo h2 { font-weight: bold; color: #00368F; font-size: 1.4em; margin: 0; }
#titulares ul { list-style: none; margin: 0; padding: 0; }
#titulares ul li { border-bottom: 1px solid #EDEDED; padding: 6px; line-height: 1.4em; }
#titulares a { display: block; font-size: 12px; color: #369; }
#titulares a:hover { text-decoration: none; color: #C00; }
#contenidos h1 { font-weight: bold; color: #00368F; font-size: 1.4em; padding: .2em;
margin: .3em 0 0 0; }
#contenidos h2 { font-weight: bold; color: #888; font-size: .9em; padding: .2em;
margin: .3em 0 0 0; }
#contenidos p { color: #222; font-size: 1.1em; padding: 4px; line-height: 1.5em; }
</style>
</head>
<body>
<form action="#">
  <input type="text" size="40" id="url" value="http://www.microsiervos.com" />
  <input type="button" value="Mostrar RSS" id="mostrar" />
  <div id="info"></div>
</form>

<div id="noticias">
  <div id="titulo"></div>
  <div id="titulares"></div>
  <div id="contenidos"></div>
</div>
</body>
</html>
```

14.22. Ejercicio 22

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd">
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejercicio 22 - Google Maps</title>
<script src="http://maps.google.com/
maps?file=api&amp;amp;v=2&amp;amp;hl=es&amp;amp;key=ABQIAAAA30JtKUU8se-7KKPRGSfCMBT2yXp_ZAY8_ufC3CFXhHIE1NvwkxRZ
type="text/javascript"></script>
<script type="text/javascript">
function load() {
  if (GBrowserIsCompatible()) {
    // Variables para el mapa
    var lat = 42.845007;
    var lon = -2.673;
    var zoom = 5;

    // Crear un nuevo mapa
    var map = new GMap2(document.getElementById("map"));

    // Centrarlo en un punto geográfico y con un nivel de zoom
    map.setCenter(new GLatLng(lat, lon), zoom);

    // Añadir los controles de zoom y de tipo de mapa
    map.addControl(new GSmallMapControl());
    map.addControl(new GMapTypeControl());

    // Crear el segundo mapa
    var map2 = new GMap2(document.getElementById("map2"));
    // Calcular las antipodas del punto geográfico
    var antipodas = calculaAntipodas(lat, lon);
    // Centrarlo en el punto geográfico de las antípodas y con el mismo nivel de zoom
    map2.setCenter(new GLatLng(antipodas.lat, antipodas.lon), zoom);

    // Cuando se mueve el primer mapa, se sincroniza la posición y aspecto del segundo
    GEvent.addListener(map, "move", function() {
      var centro = map.getCenter();
      var antipodas = calculaAntipodas(centro.lat(), centro.lng());
      map2.setCenter(new GLatLng(antipodas.lat, antipodas.lon), map.getZoom());
      map2.setMapType(map.getCurrentMapType());
    });

    GEvent.addListener(map, "click", function(marcador, punto) {
      // Crear el nuevo marcador y mostrar sus coordenadas
      var nuevoMarcador = new GMarker(punto);
      GEvent.addListener(nuevoMarcador, "click", function() {
        this.openInfoWindowHtml("Lat: " + this.getPoint().lat() + "<br/>Lon: " +
this.getPoint().lng());
      });
      map.addOverlay(nuevoMarcador);

      // Crear el marcador correspondiente a las antípodas
```

```
      var antipodas = calculaAntipodas(punto.lat(), punto.lng());
      var marcador2 = new GMarker(new GPoint(antipodas.lon, antipodas.lat));
      GEvent.addListener(marcador2, "click", function() {
        this.openInfoWindowHtml("Lat: " + this.getPoint().lat() + "<br/>Lon: " +
this.getPoint().lng());
      });
      map2.addOverlay(marcador2);
    });
  }

  function calculaAntipodas(lat, lon) {
    var antiLat = -lat;
    var antiLon = 180 - Math.abs(lon);
    return {lat: antiLat, lon: antiLon};
  }
}

window.onload = load;
window.onunload = GUnload;

</script>
<style type="text/css">
  #map, #map2 {border:1px solid black; float: left;}
</style>
</head>

<body>
  <div id="map" style="width: 500px; height: 300px"></div>
  <div id="map2" style="width: 500px; height: 300px"></div>
 </body>
</html>
```

14.23. Ejercicio 23

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Ejercicio 23 - Google Maps</title>
<script src="http://maps.google.com/
maps?file=api&amp;v=2&amp;hl=es&amp;key=ABQIAAAA30JtKUU8se-7KKPRGSfCMBT2yXp_ZAY8_ufC3CFXhHIE1NvwkxRZ
type="text/javascript"></script>
<script type="text/javascript">
var map = null;
var mgr = null;
var lat = 40.41558722527384;
var lon = -3.6968994140625;
var zoom = 6;
var puntos = {};
var peticion = null;

function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
```

```
  } else if (window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function load() {
  if(GBrowserIsCompatible()) {
    map = new GMap2(document.getElementById("map"));
    map.setCenter(new GLatLng(lat, lon), zoom);
    map.setMapType(G_SATELLITE_MAP);
    setInterval(cargaPrediccion, 3000);
  }
}

function cargaPrediccion() {
  peticion = inicializa_xhr();
  peticion.onreadystatechange = muestraPrediccion;
  peticion.open('GET', 'http://localhost/RUTA_HASTA_ARCHIVO/
previsionMeteorologica.php?nocache='+Math.random(), true);
  peticion.send(null);
}

function muestraPrediccion() {
  if(peticion.readyState == 4) {
    if(peticion.status == 200) {
      puntos = eval("("+peticion.responseText+")");
      map.clearOverlays();
      mgr = new GMarkerManager(map);
      mgr.addMarkers(getMarcadores(), 3);
      mgr.refresh();
    }
  }
}

function getMarcadores() {
  var marcadores = [];
  for (var i=0; i<puntos.length; ++i) {
    var marcador = new GMarker(getPunto(i), { icon: getIcono(i) });
    marcadores.push(marcador);
  }
  return marcadores;
}

function getPunto(i) {
  var punto = puntos[i];
  var lat = punto.latlon[0];
  var lon = punto.latlon[1];
  return new GLatLng(lat, lon);
}

function getIcono(i) {
  var punto = puntos[i];
  var icono = new GIcon();
  icono.image = "imagenes/" + punto.prediccion + ".png";
  icono.iconAnchor = new GPoint(16, 16);
```

```
  icono.iconSize = new GSize(32, 32);

  return icono;
}
</script>
</head>

<body onload="load()" onunload="GUnload()">
  <div id="map" style="width: 600px; height: 600px"></div>
</body>
</html>
```