Unidad 5

JavaScript

Apuntes básicos de programación de JavaScript y el DOM de las páginas web

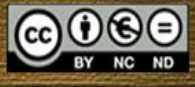
Apuntes realizados para la asignatura de FP Grado Superior:

Lenguajes de Marcas y Sistemas de Gestión de Información

del ciclo Administración de Sistemas Informáticos en Red

Autor: Jorge Sánchez Asenjo (www.jorgesanchez.net)

Versión del documento: 8.1, Año 2013



índice

(5) JavaScript

(5.1) introducción

(5.1.1) lenguajes script. JavaScript

código incrustado

El código **incrustado** o **embebido** (terrible nombre, por cierto) es un lenguaje pensado para incorporar sus instrucciones dentro de otro lenguaje.

Es una técnica casi tan vieja como la propia ciencia de la programación de aplicaciones. Así en el lenguaje **C**, por ejemplo, ha sido muy habitual añadir instrucciones de **ensamblador**; o en el lenguaje **Java** posibilitar incluir instrucciones **SQL** con las que comunicarse con la base de datos.

¿La razón? Que no hay lenguaje que sea el mejor en cada campo y, a veces, conviene utilizar varios lenguajes para abarcar todas las ventajas. Naturalmente esto implica que para traducir estas instrucciones junto con las del lenguaje anfitrión, el intérprete o el compilador entienda ambos lenguajes (o bien permita añadir extensiones que le posibiliten traducir el lenguaje embebido).

lenguajes script

Los lenguajes de script o **de guiones** aparecieron con los sistemas operativos. En especial con Unix. Se trata de archivos de texto que contienen instrucciones que el sistema es capaz de traducir.

Este archivo es **interpretado** (en lugar de **compilado**), es decir, se traduce línea a línea. De modo que si hay un error en la línea 8, se ejecutarán las primeras 7 sin problemas y en la octava se nos avisará del error (y si es grave se parará la ejecución del programa). Un compilador no ejecuta nada hasta que no se reparen todos los errores, lo cual asegura un código más limpio y eficiente, ya que el código máquina final se decide tras examinar todo el código y así se optimiza.

Lenguaje interpretado no es sinónimo de lenguaje de script, hay lenguajes interpretados como por ejemplo el **BASIC** clásico que no son considerados script. Para que un lenguaje sea de script tiene que hacer una labor sencilla; es decir, no es un lenguaje de propósito general (como C, Java, Python,…) sino que está pensado para una pequeña tarea.

Ejemplos de lenguajes script son:

- **Shell Script**. Lenguaje de tipo script que permite realizar programas para el sistema operativo. Es especialmente utilizado el Shell Script de Unix:
- **VBScript**. Lenguaje de macros de las aplicaciones de Microsoft. Permite aumentar las posibilidades de trabajo con los documentos de **Microsoft Office**.
- **PostScript**. Lenguaje para programar las impresoras.
- **ActionScript**. El lenguaje interno de las aplicaciones Flash.

lenguajes de script para HTML. aparición de JavaScript

Las limitaciones de HTML han provocado que la idea de incrustar un lenguaje de script dentro del código HTML, fructificara desde el principio.

Inicialmente la forma de conseguir dar más potencia a las páginas web (que entonces sólo eran textos y como mucho imágenes estáticas) fue añadir tecnologías de servidor como **CGI** (interfaz de programación que permitía comunicar las páginas web con programas que se ejecutaban en el lado del servidor) o bien incrustar en el código HTML código del exitoso lenguaje Java (que se creó en 1995) en forma de lo que se conoce como **applet** (pequeña aplicación Java cuyo tamaño y posición dentro de la página se establecía mediante la etiqueta del mismo nombre **<applet>).**

No obstante ambas técnicas eran excesivamente complejas, salvo para los programadores; muchos diseñadores de páginas web no se sienten cómodos con lenguajes completos como **Java**, **Perl** o **Python** (los tres muy relacionados con la creación de aplicaciones web), pero sí deseaban añadir potencia similar a la que estos lenguajes aportan.

De ahí que se creara un lenguaje mucho más sencillo que Java (y, de hecho, muy distinto), ese fue el **LiveScript** ideado por la empresa **Netscape** (concretamente por **Brendan Eich)**. Como Netscape busco la asociación con **Sun** (empresa creadora de java) para proyectar su lenguaje de script, el nombre final fue JavaScript (de hecho resultó ser un éxito comercial la elección de ese nombre).

Fue **Netscape Navigator 2.0** el primer navegador capaz de interpretar instrucciones JavaScript dentro del código HTML; y eso ocurrió a principios de 1996. En agosto de 1996, Microsoft en su navegador **Internet Explorer 3.0** incorporó la posibilidad también de interpretar código de un lenguaje llamado **JScript** (por los problemas de patentes) que era otra versión de JavaScript.

Con esta situación apareció el primer problema: la incompatibilidad del lenguaje entre los navegadores. En 1997 la **ECMA** (***European Computer Manufacturers Association***) decidió estandarizar el lenguaje, pero no lo llamó JavaScript, sino ECMAScript. Se sigue considerando el único estándar de JavaScript.

Además el W3C (***World Wide Web Consortium***), organismo encargado de estandarizar HTML decidió proponer un estándar del modelo de objetos del documento (**DOM**) que permite a los lenguajes script (especialmente pensando en JavaScript) acceder a los diferentes elementos de una página web para modificar sus propiedades.

scripts del cliente vs script del servidor

A lo largo de estos años han aparecido tanto lenguajes de script en el lado del cliente como lenguajes de script en el lado del servidor.

La diferencia es que en el caso de los lenguajes de script en el lado del cliente es el navegador el que tiene que interpretar el código y eso implica que los usuarios deben contar con navegadores que reconozcan JavaScript; además, como no hay un estándar real, deberemos tener un navegador capaz de reconocer ese JavaScript en concreto.

los lenguajes de script de servidor se crean como los del cliente; es decir, dentro del código HTML se incorporan instrucciones de ese otro lenguaje. La diferencia es que cuando el cliente pide una página web, será el servidor de aplicaciones web el encargado de traducir las instrucciones del lenguaje de script; de modo, que al navegador del cliente le llega una página web normal.

Los lenguajes de script de servidor más populares son:

- **PHP**

- ASP
- JSP
- ColdFussion
- Ruby para *Ruby on Rails*

desventajas de JavaScript sobre los lenguajes de script del servidor

- Los lenguajes de script en el lado del cliente, dependen de que el navegador del cliente sea compatible con el lenguaje utilizado. En el caso de los lenguajes en el lado del servidor, el creador de la página, sabe de antemano qué tecnología posee el servidor y adaptarse a la misma (en el caso de los clientes no se puede adaptar a todos porque existen miles de clientes distintos).
- La potencia de un servidor es mayor que la de los ordenadores de los usuarios, por lo que las tecnologías del lado del servidor pueden utilizar lenguajes más potentes.
- El código queda oculto al cliente ya que lo que le llega es la traducción de esas instrucciones (normalmente a HTML). Eso permite proteger la forma de acceder a la base de datos por ejemplo (ocultando nombres de usuario y contraseñas).

ventajas de JavaScript

- La interactividad de una página es más potente cuando el que interpreta las órdenes es el navegador. De otro modo hay que esperar la respuesta a una traducción que ocurre en un servidor, cuyo resultado es HTML, es decir poco interactivo.
- JavaScript es un lenguaje sencillo que no tiene posibilidad de utilizarse para crear código dañino.

En realidad en la actualidad lo lógico es usar ambas tecnologías para producir lo mejor de ambos mundos.

(5.1.2) el triunfo de JavaScript

Hace unos años parecía que JavaScript estaba en decadencia debido al auge de las páginas que incorporaban Flash (tecnología del lado del cliente) y/o lenguajes de servidor (especialmente PHP) para crear aplicaciones ricas (las llamadas **RIA**, *Rich Internet Applications*).

Sin embargo Flash impone al usuario un **plugin** en sus navegadores para reproducir contenidos Flash y, además, es una tecnología muy distinta a los lenguajes de la web. Sumado al hecho de que los nuevos dispositivos de la empresa **Apple** (como el **iPhone** y el **iPad**) no reproducen Flash en sus navegadores, ha propiciado el alzamiento, parece que definitivo, de JavaScript.

La dificultad de JavaScript es que es un lenguaje de programación; fácil, pero un lenguaje de programación. Esto significa que los creadores de sitios web que no sean programadores van a tener dificultades para trabajar en JavaScript.

Además sigue pendiente el tema de la compatibilidad que hace que un creador de páginas web tenga que escribir diferentes versiones de su código para los distintos navegadores. A este respecto, librerías como **jQuery** junto con toda una ordenada de nuevas librerías estimuladas por el éxito de jQuery; han facilitado enormemente el trabajo, ya que permiten escribir código que funcionará perfectamente en cualquier navegador.

El éxito de JavaScript es tan patente que se han creado motores (como **Node.js**) que interpretan código JavaScript para crear aplicaciones completas; es decir permiten utilizar JavaScript como lenguaje independiente, no incrustado dentro de HTML; en definitiva, permiten programar aplicaciones (especialmente servicios de red) completas.

Eso significa que JavaScript está más vivo que nunca y que está pasando a ser uno de los lenguajes imprescindibles para los profesionales informáticos.

(5.1.3) herramientas para escribir JavaScript

Puesto que el código JavaScript se escribe dentro del código HTML, lo lógico es utilizar nuestro editor habitual de código HTML. Pero sería interesante que dicho editor reconozca la sintaxis de JavaScript para que nos ayude a escribir el código de forma cómoda.

Algunas herramientas que facilitan la escritura de JavaScript son:

- **Aptana**. Entorno gratuito basado en **Eclipse** muy utilizado para escribir aplicaciones PHP y Ruby (aunque reconoce muchos más lenguajes). Más información en http://www.aptana.com/
- **NetBeans**. Inicialmente pensado para desarrollar en lenguaje Java, se ha convertido en una opción también más que válida para programar páginas con JavaScript. Más información: https://netbeans.org/index_es.html
- **WebStorm**. Basado en otro entorno Java, **IntellyJ**, es una versión orientada a escribir páginas web. Muy buena opción para desarrollar en JavaScript. Vale unos 50$, aunque hay opciones más baratas (incluso gratis) para estudiantes y desarrolladores de software libre.

  Disponible en http://www.jetbrains.com/webstorm/
- **Sublime Text**. Se trata de un editor de texto que permite escribir código de casi cualquier lenguaje de programación. Posee numerosas extensiones y facilidades para programar JavaScript cómodamente. Vale 70$, pero es posible utilizarlo libremente (eso sí se nos recordará continuamente que el programa es de pago en realidad). Véase http://www.sublimetext.com/
- **Coda**. El editor de código más popular para el entorno Mac. Vale 100$. Disponible en http://panic.com/coda/
- **Komodo Edit**. Versión gratuita de **Komodo IDE**, entorno completo (350$) para programar aplicaciones en diversos lenguajes. La versión **Edit** (que es gratuita) tiene recortadas algunas prestaciones, pero es un muy buen editor de HTML, CSS y JavaScript (además de otros muchos lenguajes). Más información: http://www.activestate.com/komodo-edit
- **Dreamweaver**. Es un entorno de diseño web más pensado para diseñar de forma visual. Pero tiene un buen editor de código que también reconoce JavaScript. Su precio, unos 300 €.

  Más información: http://www.adobe.com/es/products/dreamweaver.html
- **Cloud9 IDE**. Entorno de desarrollo en la nube. Nuestro trabajo se aloja directamente en Internet y el entorno de trabajo se utiliza a través de un navegador. Es una forma de trabajar novedosa, pero que tiene la ventaja de que no necesitamos instalar nada en nuestra máquina, ya que utilizamos una máquina virtual en la nube que podremos configurar a voluntad con lo necesario para programar cómodamente en JavaScript.

Es gratuita para un espacio de trabajo. Si necesitamos más podremos contratarles por un precio (no muy alto) mensual. Más información: https://c9.io/

(5.2) bases de JavaScript

(5.2.1) añadir código JavaScript en las páginas HTML

etiqueta script

La etiqueta script es la encargada de añadir JavaScript a una página HTML. Dentro de esa etiqueta el texto se interpreta como código.

script tiene dos atributos relacionados con JavaScript:

- type. Al que se le indica el valor text/javascript.
- language. Con valor JavaScript

Ejemplo (código JavaScript que muestra el mensaje hola):

```
<script language="JavaScript" type="text/javascript">
    alert("Hola");
</script>
```

añadir JavaScript de un archivo externo

Podemos añadir código JavaScript procedente de un archivo de texto (a los archivos con código JavaScript se les pone la extensión .js) Para ello simplemente se añade un atributo a la etiqueta script con la ruta al archivo JavaScript. Dicho atributo es src. Ejemplo:

```
<script language="JavaScript" type="text/javascript" src="scripts/cabecera.js>
</script>
```

(5.2.2) reglas del lenguaje JavaScript

A la hora de escribir código en JavaScript hay que tener en cuenta estos detalles:

- JavaScript distingue entre mayúsculas y minúsculas. En general, en JavaScript todo se escribe en minúsculas. En cualquier caso, debemos respetar escrupulosamente las mayúsculas y las minúsculas; para JavaScript no es lo mismo la palabra while que la palabra While.
- comentarios. Se pueden escribir comentarios dentro del código a fin de documentar el mismo. Ese código es ignorado por los intérpretes de JavaScript (pero es muy importante poner comentarios para que se entiende mejor el código). Hay dos tipos de comentarios:
  - Comentarios que empiezan por /* y terminan con */. Pueden abarcar varias líneas de código. Ejemplo:

```
/* este es un
comentario
de varias líneas */
```

  - Comentarios que comienzan con //. Se usa para comentarios de una sola línea. Ejemplo:

```
var x=18 //x vale 18
```

- **Punto y coma**. En JavaScript (como en C y como en Java) cada línea de código termina por punto y coma. No obstante sólo es obligatorio indicar este símbolo si escribimos dos instrucciones en la misma línea. Es decir en este código:

```
var x=18; var texto="Hola";
```

Al menos hay que escribir el primer punto y coma. Pero si lo hubiéramos escrito así:

```
var x=18
var y=19
```

También sería correcto, **pero no es recomendable**, ya que genera malos hábitos y es más fácil cometer errores.

(5.2.3) escritura de texto en la página

alert

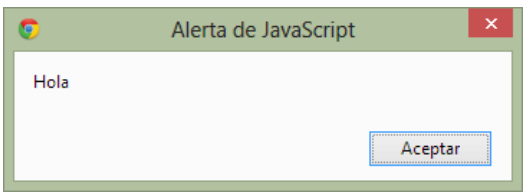
Permite mostrar un mensaje al usuario. Sintaxis:

```
alert(texto_del_mensaje);
```

Ejemplo:

```
<!doctype html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
      <title></title>
  </head>
  <body>
    <script type="text/javascript">
          alert("Hola")
    </script>
  </body>
</html>
```

Resultado al ver la página:



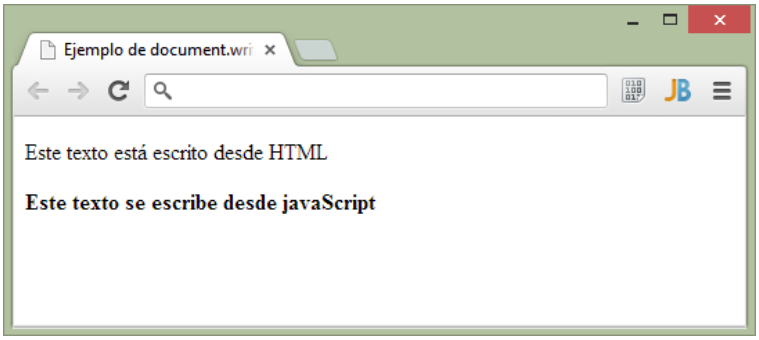
document.write

Es la forma habitual en JavaScript de conseguir escribir dentro de la página. Ejemplo:

```
<!doctype html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
      <title>Ejemplo de document.write</title>
  </head>
  <body>
    <p>Este texto está escrito desde HTML</p>

    <script type="text/javascript">
          document.write(
            "<p><strong>Este texto se escribe desde JavaScript</strong></p>");
    </script>
  </body>
</html>
```

El resultado sería:



Hay que observar que **document.write** puede escribir texto HTML, es decir puede contener etiquetas. Lo cierto es que escribir desde **document.write** es como escribir directamente desde la página.

console.log

Esta función es la única que no está pensada para escribir dentro de una página web, sino que escribe en la consola del sistema.

Esta consola sólo está disponible si escribimos código JavaScript para motores de ejecución JavaScript como **node.js** o si utilizamos la consola en herramientas de los navegadores como la extensión **FireBug** de **Mozilla Firefox** o el **Google Chrome Developer Tools,** disponible en los navegadores Google Chrome pulsando las teclas **Ctrl+Mayús+J**

Ejemplo (mostrando la consola desde Google Chrome, una vez lanzada la página pulsaríamos **Ctrl+Mayús+J**):

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8">
```

```
    <title>Ejemplo de funciones de escritura</title>
</head>
<body>
<script type="text/javascript">
    document.write("Este texto está escrito con document.write, ");
    document.write("sale en la página web");
    console.log("Este texto se ha escrito con console.log, sale por la consola");
</script>
</body>
</html>
```

La página se vería así en el navegador **Chrome** de **Google** (tras pulsar **Ctrl+Mayús+J** en Google Chrome para acudir al depurador):

Cuando se programa para entornos de JavaScript fuera de una página web (como **Node.js**), **console.log** es la forma de escribir información desde el programa JavaScript.

(5.3) variables

(5.3.1) crear variables

identificadores

Las variables son un elemento de cualquier lenguaje de programación que permite asignar un nombre (que en realidad se conoce como **identificador** de la variable) a un determinado valor.

El identificador (el nombre) que se le pone a la variable tiene que cumplir unas normas estrictas. En el caso de JavaScript son las siguientes:

- Sólo admite letras minúsculas, mayúsculas, números o el símbolo _ (guion bajo)
- Debe empezar con letras o el guion bajo
- Como máximo se pueden utilizar 31 caracteres
- Se pueden utilizar letras Unicode (es válido el identificador **año**). Aunque por la problemática de la codificación de los documentos de texto conviene tener

especial cautela con esta posibilidad (deberemos asegurar que guardamos siempre en formato Unicode nuestro código y que los servidores donde alojemos nuestras páginas también serán capaces de hacerlo).

Ejemplos de nombres:

```
variable5 →Correcto
variable_5 →Correcto
variable 5 →Incorrecto
5variable →Incorrecto
saldoMaximoAbsoluto →Correcto
dinero$ →Incorrecto
```

Por otro lado hay que recordar el hecho de que JavaScript diferencia entre mayúsculas y minúsculas.

declarar variables

La declaración de variables en JavaScript se realiza con la palabra clave **var**. De otro modo, intentar utilizar una variable provocaría un error, ejemplo (ejecutando JavaScript para la consola):

```
console.log(x);
```

Si x no se ha declarado ocurriría este error en la consola:

```
ReferenceError: x is not defined
```

Si hubiéramos declarado:

```
var x;
console.log(x);
```

Ahora no ocurre un error, aunque por pantalla aparece como contenido:

```
undefined
```

Indicando que aún no se ha definido qué tipo de valores tendrá ***x*** (ya que aún no se le ha asignado ninguno). **undefined** es un valor especial de JavaScript mediante el cual se nos indica si una variable ha sido definida; es muy útil a la hora de programar.

Podemos declarar más de una variable a la vez:

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

asignar valores

El operador de asignación es el símbolo **=**. Dicho símbolo nos permite dar un valor a una variable. Una variable (de ahí su nombre) puede cambiar de valor todas las veces que quiera. Ejemplo:

```
var x;
x=9; //x vale 9
x="Hola"; //x ahora vale "Hola"
```

En el ejemplo anterior se observa que además de cambiar de valor, incluso los valores que toma una variable son de distinto tipo (a diferencia de lo que ocurre con lenguajes más estructurados como Java). Se dice que JavaScript no es un **lenguaje fuertemente**

tipado; es decir, que el tipo de datos de una variable puede cambiar durante la ejecución del programa.

Por otro lado podemos asignar valores a la vez que declaramos la variable:

```
var x=27;
```

De hecho, es lo habitual ya que de otro modo la variable queda en estado de indefinida (**undefined**). Ejemplo:

```
var x;
console.log(x); //La consola muestra undefined
```

Es una muy buena práctica de programación declarar las variables al principio del código y asignarlas siempre un valor inicial para evitar la posibilidad de tener variables *undefined*.

(5.3.2) tipos de datos

números

En JavaScript (a diferencia de lenguajes como C y Java) sólo existe un tipo para los números. Es decir, todos los números (decimales o no) son de tipo **number**:

```
var x=9,y=8.5,z=4.321;
console.log(typeof(x));
console.log(typeof(y));
console.log(typeof(z));
```

En este código JavaScript, se hace uso de la función **typeof**, que sirve para saber el tipo de datos de un número. En el caso anterior, siempre se mostraría por pantalla el texto *number*.

Puesto que para JavaScript todos los números son iguales este código:

```
var x=17/2;
```

Hace que *x* valga *8,5*.

En JavaScript el decimal se indica con un punto, por ejemplo:

```
x=2.3412;
```

Es posible utilizar también notación científica:

```
var x=1.23E+6;
console.log(x); //En la consola sale 1230000
```

Se pueden indicar números octales colocando un cero antes del número. Ejemplo:

```
var x=0276;
console.log(x)
//En la consola sale 190, es decir el equivalente decimal del número octal 170
```

También se pueden utilizar números en hexadecimal adelante **0x** (*cero equis*) al número. Ejemplo:

```
var x=0xF9;
console.log(x)
```

```
//En la consola sale 249, es decir el equivalente decimal del hexadecimal 0xF9
```

operadores numéricos

Con los números podemos utilizar los siguientes operadores:

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo, resto (*7%2* da como resultado uno, el resto de siete entre dos)
++	Incremento
--	Decremento

```
var valor1=50;
var valor2=10;
var valor3=20;
var suma, resta, producto, division, resto;
var incremento, decremento;

suma=valor1+valor2; //suma vale 60
resta=valor1-valor2;  //resta vale 40
producto=valor1*valor2;  //producto vale 5000
division=valor1/valor3;  //division vale 2,5
resto=valor1%valor3; //resto vale 10

valor1++; //valor1 vale 51
valor1--; //valor1 vale 50

//Ejemplo de postincremento, primero se asigna y luego se incrementa
//la variable
incremento=valor1++; //incremento vale 50 y valor1 vale 51
decremento=valor1--; //decremento vale 51, valor1 vale 50

//Ejemplo de preincremento, primero se incrementa
//y luego se asigna la variable
incremento=++valor1; //incremento vale 51 y valor1 también
decremento=--valor1; //decremento y valor1 valen 50
```

operadores de asignación

Al operador de asignación (=) visto anteriormente se le pueden añadir símbolos para darle la posibilidad de asignar valores tras realizar una operación sobre una variable. Por ejemplo:

```
var x=18;
```

```
x+=9; //x vale ahora 27
```

La expresión x+=9 sirve para abreviar la expresión x=x+9. Es un operador de asignación y suma. La lista completa de operadores de asignación es:

Operador	Significado	Ejemplo	Equivalente a...
+=	Suma y asignación	*x+=5*	*x=x+5*
-=	Resta y asignación	*x-=5*	*x=x-5*
*=	Multiplicación y asignación	*x*=5*	*x=x*5*
/=	División y asignación	*x/=5*	*x=x/5*
%=	Resto y asignación	*x%=5*	*x=x%5*

textos, strings

uso básico de texto

El segundo gran tipo de datos en JavaScript es el texto; lo que comúnmente se conoce como **string** (cadenas de caracteres).

En JavaScript el texto se escribe entrecomillado; bien entre comillas dobles o bien entre comillas simples; es indiferente. Tan válido es:

```
var saludo="Hola";
```

como:

```
var saludo='Hola';
```

Dentro de las comillas se puede desear poner más comillas. No habrá problema si alternamos tipos de comilla. Ejemplos:

```
var frase="Ella dijo 'Hola' al llegar";
```

En el ejemplo la palabra Hola queremos que salga entrecomillada con comillas simples, por lo que todo el texto debe de estar delimitado por comillas dobles (de otro modo se interpretaría mal el texto).

secuencias de escape

JavaScript ha heredado numerosas características del lenguaje **C**. Una de ellas es la posibilidad de utilizar **secuencias de escape**. Las secuencias de escape son caracteres especiales que no se podrían escribir directamente en un string y se deben escribir de una forma muy concreta. Para escribir secuencias de escape se usa el carácter \ (**backslash**) o barra inversa.

Por ejemplo supongamos que queremos encerrar dentro de un texto entrecomillado con comillas dobles, las propias comillas dobles; si lo intentamos directamente ocurre un error:

```
var s="Ana dijo "hola"al llegar a la casa";
```

La palabra hola no se considera parte del texto al quedar fuera de las comillas. Sin embargo si indicamos las comillas mediante \", entonces:

```
var s="Ana dijo \"hola\"al llegar a la casa";
```

La combinación \" representa a las propias comillas. Aunque en este caso podríamos haber escrito las comillas de forma normal, si el texto hubiera estado delimitado por comillas simples, es una posibilidad interesante usarlas con secuencias de escape.

Además tenemos las siguientes secuencias de escape:

código de escape	Significado
\n	Nueva línea
\t	Tabulador
\\	El propio carácter de barra invers a
\"	Comillas dobles.
\'	Comillas simples
\ooo	Cada *c* es un número en sistema octal, de modo que la las tres cifras indican el código octal de la tabla ASCII correspondiente al carácter que se mostrará. Por ejemplo *\104* es el equivalente a la letra *D* mayúscula
\x*hh*	Igual que el anterior, pero ahora las cifras son hexadecimales. Por ejemplo \x44 es el equivalente al carácter D mayúscula
\u*hhhh*	Como el anterior, pero se pueden utilizar ahora cuatro cifras hexadecimales que tomarán el carácter correspondiente a ese código hexadecimal de la tabla Unicode. Por ejemplo el código *\u0444* se corresponde al símbolo *ф*

encadenamiento de texto

En JavaScript el operador + tiene dos propósitos, sirve para sumar y también para encadenar texto. Encadenar texto no es más que unirlo y es muy útil para escribir expresiones complejas. Ejemplo:

```
var nombre="Miguel";
document.write( "Hola "+nombre);
```

En la página web saldría escrito **Hola Miguel** porque el operador + une ambas expresiones. No usar dicho operador produciría un error porque el intérprete de JavaScript no sabría cómo ligar ambas expresiones.

booleanos

Los valores booleanos (o lógicos) sirven para indicar valores de verdad (**true**) o de falsedad (**false**). Sirven para almacenar resultados de condiciones lógicas.

En el caso más simple tomarían valores directos de verdadero o falso:

```
var b=true; //la variable b vale "verdadero"
```

Pero pueden tomar resultados de operaciones lógicas, como:

```
var b= (17>5); //La variable b sigue siendo verdadera
```

La variable *b* de este ejemplo vale lo mismo que en el anterior, pero en este caso ha tomado el valor *true* tras evaluar una operación lógica.

operadores lógicos

Los operadores lógicos son aquellos que obtienen un resultado de verdadero o falso.

Operador	Significado
>	Mayor que

Operador	Significado
<	Menor que
>=	Mayor o igual
<=	Menor o igual
==	Igual
!=	Distinto
===	Estrictamente igual.
!==	Estrictamente distinto
&&	"Y" lógico (*AND*)
||	"Ó" lógico (*OR*)
!	"NO" lógico (*NOT*)

Así por ejemplo:

```
var x=19, y=13;
if(x>y) {
    alert("Hola");
}
```

En el código anterior (que incorpora la instrucción *if* que se discute más adelante) sale por pantalla el texto *Hola* porque la condición *x>y* es cierta, ya que *x* es mayor que *y*.

En el caso de la igualdad, se comparar mediante el operador ==, ejemplo:

```
document.write(3=="3");
```

En la página web en la que incorporemos este código, aparecía la palabra **true**. Sin embargo:

```
document.write(3==="3");
```

Ahora escribe **false** porque ahora no son estrictamente iguales ya que *3* es un número y *"3"* un string. El operador === sólo devuelve true si ambos valores son iguales y del mismo tipo.

Los operadores && (*AND*), || (*OR*) y ! (*NOT*) permiten establecer condiciones más complejas. Por ejemplo:

```
var x=19, y=13, z= 7, a="20";
if(x>y && z<a) {
   alert("Hola");
}
```

Aparece la palabra "Hola", porque *x>y* y además *z<a*. Para que el operador && devuelve verdadera, las dos expresiones deben serlo. El operador || devuelve verdadero si cualquiera lo es.

valores verdaderos y valores falsos

Además de usar los valores **true** y **false** y los resultados de expresiones lógicas. Hay otra serie de reglas que hay que tener en cuenta al tomar los valores booleanos. Son:

- Los números cero (tanto en positivo como en negativo, -0) se toman como un valor falso

- Los números positivos o negativos se toman como un valor verdadero. Ejemplo:

```
var x=12;
if(x) {
  document.write("Verdadero");
}
else{
   document.write("Falso");
}
```

El ejemplo anterior escribe *Verdadero*

- Los textos se toman como valores verdaderos
- La cadena vacía ("") se toma como falsa:

```
var x="";
if(x) {
  document.write("Verdadero");
}
else{
   document.write("Falso"); //Escribe falso
}
```

- Los valores **Infinity** y **-Infinity** (resultado de operaciones como 1/0 por ejemplo) son verdaderos
- Los valores **NaN**, **undefined** y **null** se toman como falsos.
- Los objetos se toman como verdaderos

(5.3.3) constantes

Las constantes se usan como las variables; su particularidad es que no podemos cambiar el valor que las asignamos inicialmente. Por ello se definen con la palabra **const**, en lugar de la palabra **var**. Se aconseja que el nombre de las constantes tenga todas las letras en mayúsculas. Ejemplo:

```
const PI=3.141591
```

(5.3.4) objetos

En JavaScript hay sólo tres tipos de datos simples: **number**, **string** y **booleanos**. Pero hay un cuarto tipo de datos: el tipo **Object**, que vale para cualquier otro tipo de datos por complejo que sea.

Aunque más adelante en estos mismos apuntes se explica el uso de los objetos (o variables de tipo **Object**), en este apartado se explica la forma de trabajar con los tipos básicos, utilizándoles como objetos, ya que JavaScript permite esa posibilidad.

JavaScript, objetos y el objeto global

JavaScript es un lenguaje orientado a objetos. No posee la potencia de lenguajes como Java para gestionar objetos complejos, pero sí es un lenguaje con capacidad de manejar objetos.

De hecho en JavaScript se considera que todas las variables realmente son objetos; incluso en JavaScript hay un objeto global que es el propietario de las funciones y variables que creamos en nuestro código JavaScript. Así la función **typedef** (a la que se accede sin utilizar objeto alguno) se dice que es un método del objeto global, luego es accesible desde cualquier parte del código.

También los valores **undefined**, **Infinity** o **NaN** se consideran propiedades del objeto global; así como las funciones **isNaN**, la ya comentada **typedef** y otras muchas que posee el lenguaje. Parte del objeto global son los llamados objetos globales (equivalentes a las clases estáticas de Java) como **Math** (que se comenta más adelante).

Cuando declaramos una variable normal, realmente lo estamos haciendo en el objeto global y eso la hace accesible desde cualquier parte del código. Es decir, al objeto global le podemos añadir nuevas variables y funciones.

Esta forma de entender el código JavaScript tiene que ver con la llamada **Programación Orientada a Objetos (POO)** en la que los programas se crean definiendo objetos como un conjunto de propiedades (atributos) y métodos (funciones) que pueden realizar.

uso de los objetos JavaScript

Aunque más adelante veremos cómo crear objetos propios, cuando declaramos variables ya tenemos en realidad objetos que queremos usar.

En casi todos los lenguajes que manejan objetos (y eso incluye a JavaScript), podemos acceder a los métodos y propiedades de los objetos poniendo un punto y el nombre de la propiedad. Así por ejemplo:

```
var x="Este es un texto de prueba";
document.write(x.length); //Escribe 26
```

La propiedad **length** está disponible para todas las variables de tipo string; o sea para todos los ***objetos de clase String***, que sería la forma compatible con la POO.

En el caso de los métodos:

```
var x="Este es un texto de prueba";
document.write(x.toUpperCase());
//Escribe ESTE ES UN TEXTO DE PRUEBA
```

toUpperCase es un método de la clase String, por ello se ponen paréntesis. Algunos métodos requieren parámetros, valores para poder realizar su labor. Ejemplo:

```
var x="Este es un texto de prueba";
document.write(x.charAt(13));
/*Escribe la letra x que está en la posición 13, ya que empezamos a
contar desde la posición cero */
```

No sólo podemos utilizar métodos para las variables de tipo string, también los números y el resto de elementos los tienen. Ejemplo:

```
var x=1.23456789;
document.write(x.toFixed(3));
//Muestra 1.235 ya que redondea a tres decimales
```

métodos y propiedades de la clase string

Estos métodos y propiedades están disponibles para todas las variables de clase string, para acceder a ellos bastará poner el nombre de la variable seguida de un punto y luego poner el nombre del método o la propiedad. En el caso de los métodos se utiliza el nombre del método seguido de paréntesis en los que se indica el valor de los parámetros (si les hay). En la lista aparecen entre corchetes los parámetros que son opcionales .

propiedades

Propiedad	Resultado
length	Tamaño del texto. Ejemplo: `document.write("Hola".length);//Escribe 4`

métodos

Método	Resultado

charAt(índice)	Tamaño del texto. Ejemplo:
	`document.write("Hola".charAt(2));//Escribe l`
	Hay que tener en cuenta que el primer carácter tiene índice cero

charCodeAt(índice)	Devuelve el código Unicode del carácter situado en la posición indicada por el parámetro índice.

indexOf(texto [,inicio])	Devuelve la posición del texto indicado en la variable, empezando a buscar desde el lado derecho. Si aparece varias veces, se devuelve la primera posición en la que aparece. El segundo parámetro (inicio) nos permite empezar a buscar desde una posición concreta.
	Ejemplo:
	`var var1="Dónde esta la x, busca, busca";`
	`document.write(var1.indexOf("x"));`
	`//Escribe 14, posición del carácter "x"`
	En el caso de que el texto no se encuentre, devuelve -1

lastIndexOf(texto [,inicio])	Igual que el método anterior, pero ahora se devuelve la última posición en la que aparece el texto (o menos uno si no se encuentra).

Método	Resultado

match(expresiónRegular)	Devuelve un array con todas las apariciones de la expresión regular en el texto o el valor null si no encaja la expresión en el texto). Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.match(/st/g));`
	`//Escribe por consola: [st,st,st]`
	`//ya que el texto st aparece tres veces`
	Si hubiéramos hecho:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.match(/st/));`
	Sólo nos sacaría la primera aparición (aunque en el array habría otro elemento para indicar la posición de esa expresión y otro más para indicar cuál era el texto original). Es decir sin usar la *g* devolvería:
	`[ 'st', index: 1, input: 'Esto es una estructura estática' ]`

replace(textoBusq,textoReem)	Busca en el string el texto indicado en el primer parámetro (textoBusq) y lo cambia por el segundo texto (textoReem). Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.replace("st","xxtt"));`
	`//Escribe: Exxtto es una estructura estática`

search(expresiónRegular)	Comprueba la expresión regular en el texto y devuelve la posición en la que se cumple dicha expresión en el texto. Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.search(/st/));`
	`//Escribe por consola: 1`
	`//La posición en la que aperece st por primera`
	`//vez`

Método	Resultado

slice(inicio [,fin])	Toma del texto los caracteres desde la posición indicada por **inicio**, hasta la posición **fin** (sin incluir esta posición). Si no se indica fin, se toma desde el inicio hasta el final.
	Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.slice(3,10);`
	`//Escribe por consola: o es un`
	El parámetro **fin** puede usar números negativos, en ese caso, éstos cuentan desde el final del texto. Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.slice(3,-5);`
	`//Escribe por consola: o es una estructura est`

split([textoDelim [,límite])	Divide el texto en un array de textos. Sin indicar parámetro alguno, cada elemento será un carácter del array. Si se indica el parámetro **textoDelim**, se usa como texto delimitador; es decir, se divide el texto en trozos separados por ese delimitador. Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.split(" "));`
	`//Escribe:`
	`// [ 'Esto', 'es', 'una', 'estructura', 'estática' ]`
	El segundo parámetro pone un límite tope de divisiones. Por ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.split(" ",3));`
	`//Escribe:`
	`// [ 'Esto', 'es', 'una']`
	El texto delimitador puede ser una expresión regular en lugar de un texto, eso permite dividir el texto en base a criterios más complejos.

Método	Resultado

substr(inicio [,tamaño])	Toma del texto el número de caracteres indicados por el parámetro tamaño, desde la posición indicada por inicio. Si no se indica tamaño, se toma desde el inicio hasta el final.
	Ejemplo:
	`var texto="Esto es una estructura estática";`
	`console.log(texto.substr(3,10);`
	`//Escribe por consola: o es una e`
	substr no funcionaba en Internet Explorer hasta la versión 9.

substring(inicio [,fin])	Igual que slice pero no admite usar números negativos.
toLowerCase()	Convierte el texto a minúsculas
toUpperCase()	Convierte el texto a mayúsculas

métodos HTML

Están pensados para añadir código HTML al texto. Retornan otra versión del texto (no modifican el original) que contiene el texto junto con las etiquetas añadidas. No se muestran en la siguiente tabla aquellos métodos que utilizan etiquetas HTML obsoletas.

Método	Resultado

anchor(texto)	Añade una etiqueta de tipo a al string y rellena el atributo name de la misma con el texto indicado. De esa forma tendríamos el código HTMl de un marcador de la página.
	`var texto="Apartado Principal";`
	`document.write(texto.anchor("marca1"));`
	`//Genera el código:`
	`//<a name="marca1">Apartado Principal</a>`

bold()	Rodea el string con etiquetas de negrita (de tipo b y no de tipo strong, que son las que aconsejan actualmente)
fixed()	Rodea el string con etiquetas de indicación de texto fijo (tt)
italics()	Rodea el string con etiquetas de cursiva (de tipo i y no de tipo em, que son las que aconsejan actualmente)

link(url)	Quizá la más interesante de esta sección, permite rodear al string de una etiqueta a con destino (es decir con el atributo href) apuntando a la URL indicada. Ejemplo:
	`var texto="Ir a mi página";`
	`document.write(texto.link("http://www.jorgesanchez.net"));`
	`//Resultado:`
	`// <a href="http://www.jorgesanchez.net">Ir a mi página</a>`

Método	Resultado
sub()	Devuelve el texto rodeado por la etiqueta de subíndice (sub)
sup()	Devuelve el texto rodeado por la etiqueta de superíndice (sup)

métodos estáticos

Los métodos estáticos son un tipo especial. No se pueden utilizar mediante variables de tipo String, sino que se utilizan poniendo el nombre de la clase String.

Método	Resultado
String.fromCharCode(listaCódigos)	Devuelve un texto resultado de convertir los códigos indicados en su carácter correspondiente. Ejemplo: `document.write(String.fromCharCode(72,111,108,97));` Escribe *Hola*

métodos y propiedades de la clase Number

La clase Number es la clase a la que pertenecen las variables numéricas de JavaScript. Como ocurría con los textos, las variables numéricas pueden ser tratadas como objetos que poseen métodos que permiten realizar interesantes tareas sobre los números. Se exponen a continuación.

Método	Resultado

toExponential([dígitos])	Convierte números a su formato de notación exponencial. El parámetro (opcional) dígitos permite indicar el máximo número de cifras que se usarán en dicha notación.
	`var número=234567.637462;`
	`console.log(número.toExponential(5));`
	`//Escribe por consola: 2.34568e+5`

toFixed([decimales])	Convierte el número de forma que use un máximo de decimales. Si no se indica parámetro alguno, se tomará el valor cero (sin decimales). Ejemplo:
	`var número=234567.637462;`
	`console.log(número.toFixed(3));`
	`//Escribe por consola: 234567.637`

toPrecision([decimales])	Equivalente al anterior, pero usando el formato interno de decimal de coma flotante (equivalente al tipo float del lenguaje C)
toString()	Convierte el número a string (a texto).

métodos de la clase Boolean

Las variables booleanas también se pueden tratar como objetos. En este caso sólo disponen del método toString() para poder convertirlas a forma de texto (string)

(5.3.5) expresiones regulares

Uno de los usos más habituales en la mayoría de lenguajes de programación, tiene que ver con expresiones regulares.

Se utilizan, sobre todo, para establecer un patrón que sirva para establecer condiciones avanzadas en los textos. Estas condiciones nos facilitan la tarea de búsqueda de textos dentro de otros textos, validación de textos o extracción avanzada de subcadenas.

Las expresiones regulares de JavaScript en realidad son objetos de tipo **RegExp** pero admiten ser usados de manera muy cómoda. En JavaScript el foprmasto de expresión regular es el del estándar **PCRE**, http://www.pcre.org/, (***Perl Compatible Regular Expressions***) , que es el formato compatible con el lenguaje Perl (famoso precisamente por sus expresiones regulares).

Las expresiones regulares se pueden crear delimitándolas entre dos barras de dividir. Entre ambas barras se coloca la expresión regular. Tras las barras se pueden indicar modificadores, la sintaxis sería esta:

```
var patrónExpresiónRegular=/expresion/modificadores
```

Los posibles modificadores son:

- **i** Hace que en la expresión regular no se distinga entre mayúsculas o minúsculas
- **g** Permite que la expresión regular busque todas las veces que la expresión regular encaja en el texto
- **m** Permite comprobar la expresión en modo multilínea.

Se pueden utilizar varios modificadores:

```
var reg=/[a-z]/gi
```

Esa expresión la cumplen los textos que contengan letras (mayúsculas o no) y además permite buscar todas las apariciones de letras (no sólo para en el primer encaje).

elementos de las expresiones regulares

El patrón de una expresión regular puede contener diversos símbolos. En los siguientes apartados se indican las posibilidades

símbolo	significado
c	Si c es un carácter cualquiera (por ejemplo *a*, *H*, *ñ*, etc.) indica, donde aparezca dentro de la expresión, que en esa posición debe aparecer dicho carácter para que la expresión sea válida.
cde	Siendo c, d, y e caracteres, indica que esos caracteres deben aparecer de esa manera en la expresión. Ejemplo, la expresión: `/hola/` la cumplen los textos que tengan dentro la palabra hola
(x)	Permite indicar un subpatrón dentro del paréntesis. Ayuda a formar expresiones regulares complejas.
.	Cualquier carácter. El punto indica que en esa posición puede ir cualquier carácter

símbolo	significado
^x	Comenzar por. Indica el String debe empezar por la expresión ***x***.
x$	Finalizar por. Indica que el String debe terminar con la expresión x. Ejemplo: /^Hola$/ Sólo cumplirán este patrón, los textos que exactamente contengan la palabra ***Hola***
x+	La expresión a la izquierda de este símbolo se puede repetir una o más veces
x*	la expresión a la izquierda de este símbolo se puede repetir cero o más veces
x?	El carácter a la izquierda de este símbolo se puede repetir cero o una veces
x?=y	El carácter x irá seguido del carácter y
x?!y	El carácter x no irá seguido del carácter y
x{n}	Significa que la expresión x aparecerá n veces, siendo n un número entero positivo.
x{n,}	Significa que la expresión x aparecerá n o más veces
x{m,n}	Significa que la expresión x aparecerá de m a n veces.
x|y	La barra indica que las expresiones x e y son opcionales, se puede cumplir una u otra.
c-d	Cumplen esa expresión los caracteres que, en orden ASCII, vayan del carácter c al carácter d. Por ejemplo a-z repsenta todas las letras minúsculas del alfabeto inglés.
[cde]	Opción, son válidos uno de estos caracteres: c, d ó e
[^x]	No es válido ninguno de los caracteres que cumplan la expresión x. Por ejemplo ***[^dft]*** indica que no son válidos las caracteres ***d***, ***f*** ó ***t***.
\d	Dígito, vale cualquier dígito numérico
\D	Todo menos dígito
\s	Espacio en blanco
\S	Cualquier carácter salvo el espacio en blanco
\w	***Word***, carácter válido. Es lo mismo que ***[a-zA-Z0-9]***
\W	Todo lo que no sea un carácter de tipo ***Word***. Equivalente a ***[^A-Za-z0-9]***
\0	Carácter nulo
\n	Carácter de nueva línea
\t	Carácter tabulador
\\	El propio símbolo \
\"	Comillas dobles
\'	Comillas simples
\c	Permite representar el carácter ***c*** cuando este sea un carácter que de otra manera no sea representable (como ***[***, ***]***, ***/***, \,…). Por ejemplo \\ es la forma de representar la propia barra invertida.

símbolo	significado
\ooo	Permite indicar un carácter Unicode mediante su código octal.
\xff	Permite indicar un carácter ASCII mediante su código hexadecimal.
\uffff	Permite indicar un carácter Unicode mediante su código hexadecimal.

Ejemplo de expresión que valida un código postal (formado por 5 números del 00000 al 52999). Los paréntesis ayudan a agrupar esta expresión:

```
var cp1="49345";
var cp2="53345";
var exp=/^((5[012])|([0-4][0-9]))([0-9]{3})$/;
console.log(exp.test(cp1)); //Escribe true
console.log(exp.test(cp2));//Escribe false
```

métodos de las expresiones regulares

Las variables a las que se asigna una expresión regular son en realidad objetos (clase Object) sobre los que podemos utilizar sus métodos. El más importante es **test** (como se puede apreciar en el código anterior) que devuelve verdadero si el texto que se le pasa cumple la expresión regular. La lista completa de métodos es:

método	significado
compile(expReg [,modif])	En el caso de que deseemos cambiar una variable que almacena una expresión regular, este método es indispensable para que dicha variable reconozca la nueva expresión. El segundo parámetro (modif) es un string que permite indicar los modificadores **i**, **g** o **gi** comentados anteriormente.
exec(texto)	Comprueba si el texto cumple la expresión regular; de ser así devuelve el primer texto que la cumple; si no, devuelve **null**
test(texto)	Devuelve verdadero si el texto cumple la expresión regular.

propiedades de las expresiones regulares

Permiten obtener información sobre las mismas.

propiedad	significado
global	Devuelve verdadero si el modificador global (**g**) forma parte de la expresión regular.
ignoreCase	Devuelve verdadero si el modificador de ignorar mayúsculas (**i**) forma parte de la expresión regular.
lastIndex	Posición del texto a partir de la cual se realizará la siguiente búsqueda
multiline	Devuelve verdadero si el modificador de multilínea (**m**) forma parte de la expresión regular.
source	Devuelve la expresión regular en forma de string

(5.4) mensajes

(5.4.1) alert

Como ya se ha explicado antes la función **alert** muestra un mensaje por pantalla. La forma y tipo de cuadro (incluso el título) que se utiliza para mostrar el mensaje, dependen del navegador. Ejemplos:

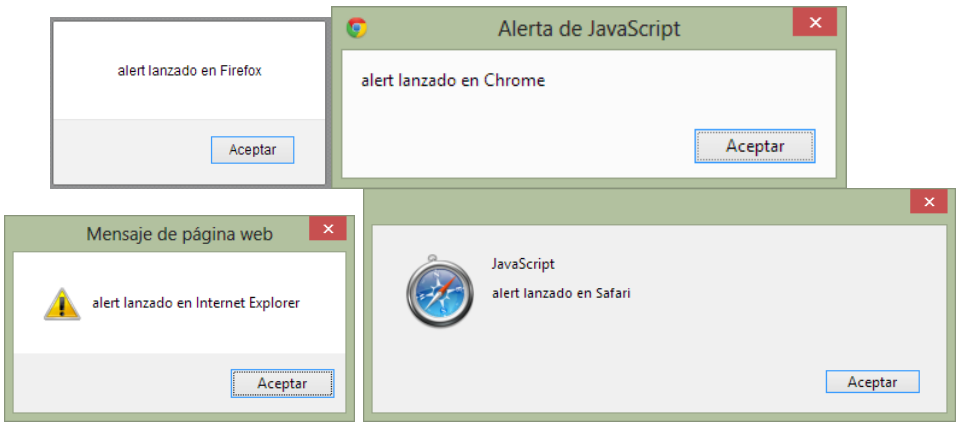


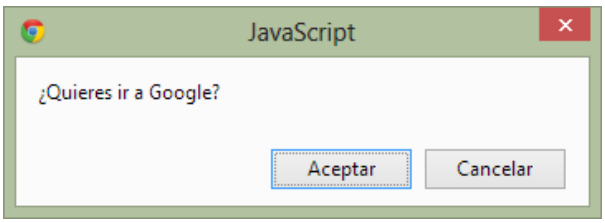
Ilustración 1, Diferentes cuadros alert según el navegador

(5.4.2) confirm

Permite mostrar un cuadro con un mensaje que se puede aceptar o cancelar. En el caso de que cancelemos el mismo la función **confirm** devuelve falso, si lo aceptamos devuelve verdadero. Ejemplo:

```
var resp=confirm("¿Quieres ir a Google?");
if(resp==true){
   location="http://www.google.es";
}
else{
   document.write("vale, nos quedamos aquí");
}
```

El cuadro que se mostraría sería (en el navegador Chrome):



Si aceptamos el cuadro la página cambiará por la del famoso buscador (gracias a la línea `location=http://www.google.es`), de otro modo se mostrará el texto ***vale, nos quedamos aquí***

(5.4.3) prompt

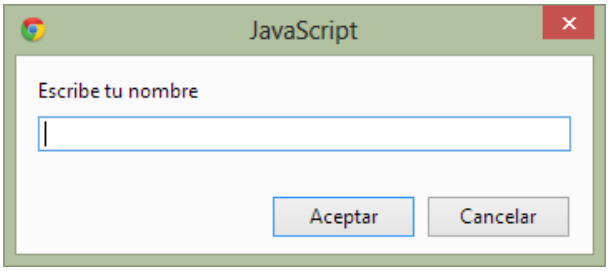
prompt es una función que muestra un cuadro de mensaje en el que podremos rellenar valores que serán devueltos por la función. De este modo esta función permite utilizar cuadros en los que el usuario puede introducir información.

El cuadro dispone también la posibilidad de cancelar,; en ese caso devolverá el valor **null** (y no ***false*** como en el caso de los cuadros **confirm**).

Ejemplo:

```
var nombre=prompt("Escribe tu nombre","");
if(nombre!=null){
   document.write("Hola "+nombre);
}
else{
   document.write("has cancelado el cuadro");
}
```

El cuadro que aparece (en el navegador Chrome) es:

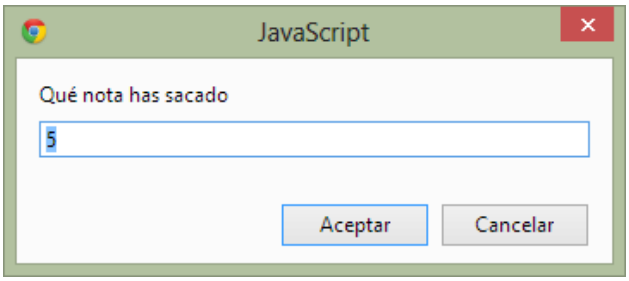


Si el usuario acepta el cuadro, se escribirá ***Hola*** seguido del texto que haya escrito en el cuadro. Si cancela saldrá el texto ***Has cancelado el cuadro***.

La función **prompt** tiene dos parámetros separados por comas (aunque el segundo es opcional). El segundo parámetro (que también es un texto) sirve para rellenar inicialmente el cuadro con el texto que indiquemos. Ejemplo:

```
var nota=prompt("Qué nota has sacado","5");
```

Mostrará el cuadro:



En él aparece ya escrito el número cinco.

Muchas veces en ese segundo parámetro se indica el texto vacío ("", comillas comillas) para que no aparezca ningún valor en el cuadro.

Otro detalle a tener en cuenta es que lo que devuelve el cuadro es un texto y que no hay manera de restringir lo que el usuario puede escribir a un tipo de datos concreto (es decir, no podemos hacer que el usuario puede meter sólo números), lo que es un gran problema.

(5.5) conversiones de datos

(5.5.1) diferencias del manejo de tipos en JavaScript respecto a los lenguajes clásicos

Lenguajes como **Java** son muy estrictos con los datos, las variables de un tipo son absolutamente incompatibles con variables de otros tipos.

Sin embargo en JavaScript las variables deciden en cada momento (en lo que se conoce como **tiempo de ejecución**) el tipo de datos que contienen. Así en JavaScript es posible hacer cosas como:

```
var x="Hola"; //x es un string
x=199; //Ahora es un número
```

Eso significa que hasta que no se ejecuta el programa no se sabe lo que va a ocupar en memoria y esa ocupación cambia constantemente (lo que hace de JavaScript un lenguaje poco eficiente, es decir menos veloz que lenguajes como C, por ejemplo). En Java o C se sabe de antemano lo que ocupan las variables: trabajan de forma más eficiente, pero son mucho menos libres para el programador.

Hay que tener en cuenta que JavaScript es un lenguaje pequeño pensado para ser utilizado en páginas web, por lo que no se requiere una gran potencia, pero sí una gran versatilidad.

(5.5.2) operador typeof y función isNaN

Eso tiene una gran desventaja, nunca sabremos con seguridad el tipo de datos que contiene una variable cuyo contenido le hayamos leído al usuario. El operador **typeof** nos ayuda:

```
var x="199";
document.write(typeof(x)); //Escribe string
x=199;
document.write(typeof(x)); //Escribe number
```

Pero para saber si un usuario ha introducido o no números, **typeof** no nos es de mucha ayuda porque el cuadro **prompt** por ejemplo siempre devuelve textos (strings).

En eso nos puede ayudar **isNaN**.Esta es una función que devuelve verdadero si el parámetro que recibe no es un número:

```
var x="199",y="hola";
document.write(isNaN(x)); //Escribe false
document.write(isNaN(y)); //Escribe true
```

(5.5.3) conversión implícita

Una de las capacidades más interesantes de JavaScript es la facilidad para convertir datos de diferente tipo. Así esta expresión:

```
document.write("13" *3);
```

Escribe 39 (resultado de multiplicar). Mientras que esta expresión:

```
document.write("Hola" *3);
```

Escribe NaN (no es un número) porque no puede convertir Hola a un número.

Pero en esta expresión puede sorprender el resultado:

```
document.write("5"+3);//Escribe 53
```

Aquí el operador + (más) no realiza la suma de los números, sino que encadena los mismos. Esto puede tener importantes problemas, por lo que a veces es necesario convertir los datos.

métodos y funciones de conversión

Para convertir datos disponemos de varias posibilidades:

- Utilizar métodos constructores de las clases String, Number y/o Boolean. A estas tres funciones (todas ellas comienzan con la letra mayúscula) se les pasa el dato a convertir. Ejemplo:

```
var x="5";
document.write(Number(x)+3); //Escribe 8
```

  Al convertir a Number (número) si el dato no es convertible, entonces devuelve el valor NaN indicando que el original no era numérico

- Utilizar los métodos globales parseInt (pasar a entero) o parseFloat (pasar a decimal). Son la forma habitual de conversión a número ya que es la más potente. En ambos casos tomarán los números que tenga el texto que deseamos convertir y si aparecen símbolos no numéricos, simplemente los ignoran. parseInt además ignora el punto decimal. Ejemplo:

```
var x="1234.567abcd12345";
document.write(parseInt(x)); //Escribe 1234
document.write(parseFloat(x));//Escribe 1234.567
```

  Tanto parseInt como parseFloat si no tienen manera de pasar el texto a número (porque no hay números a la izquierda), devolverán NaN.

(5.6) control del flujo del programa

(5.6.1) introducción

Hasta ahora las instrucciones que hemos visto, son instrucciones que se ejecutan secuencialmente; es decir, podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

Las instrucciones de control de flujo permiten alterar esta forma de ejecución. A partir de ahora habrá líneas en el código que se ejecutarán o no dependiendo de una condición.

Esa condición se construye utilizando lo que se conoce como **expresión lógica**. Una expresión lógica es cualquier tipo de expresión JavaScript que dé un resultado booleano (verdadero o falso).

Las expresiones lógicas se construyen por medio de variables booleanas o bien a través de los operadores relacionales (==, >, <,...) y lógicos (&&,||, !) vistos anteriormente (véase operadores lógicos, página 19).

(5.6.2) instrucción if

sentencia condicional simple

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutará ninguna expresión. Su sintaxis es:

```
if(expresión lógica) {
    instrucciones
    ...
}
```

Las llaves se requieren sólo si va a haber varias instrucciones. En otro caso se puede crear el if sin llaves, es decir:

```
if(expresión lógica) sentencia;
```

Ejemplo:

```
if(nota>=5){
    document.write("Aprobado");
    aprobados++;
}
```

La idea gráfica del funcionamiento del código anterior sería:

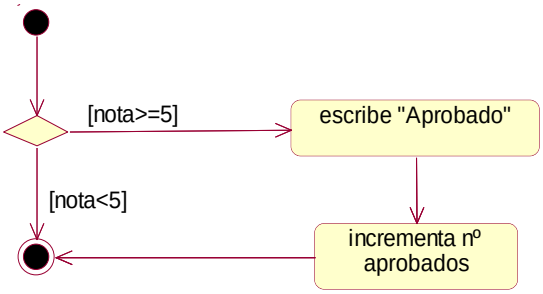


Ilustración 5-2, Diagrama de actividad de la instrucción *if* simple

sentencia condicional compuesta

Es igual que la anterior, sólo que se añade un apartado **else** que contiene instrucciones que se ejecutarán si la expresión evaluada por el **if** es falsa. Sintaxis:

```
if(expresión lógica){
    instrucciones
    ....
}
else {
    instrucciones
    ...
}
```

Como en el caso anterior, las llaves son necesarias sólo si se ejecuta más de una sentencia. Ejemplo de sentencia **if-else**:

```
if(nota>=5){
    document.write("Aprobado");
    aprobados++;
}
else {
    document.write("Suspenso");
    suspensos++;
}
```

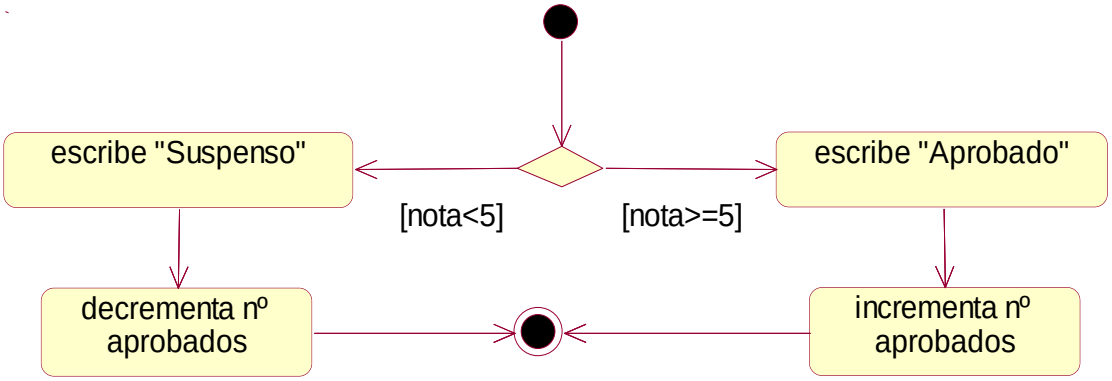


Ilustración 5-3, Diagrama de actividad de una instrucción *if* doble

anidación

Dentro de una sentencia **if** se puede colocar otra sentencia **if**. A esto se le llama **anidación** y permite crear programas donde se valoren expresiones complejas. La nueva sentencia puede ir tanto en la parte *if* como en la parte *else*.

Las anidaciones se utilizan muchísimo al programar. Sólo hay que tener en cuenta que siempre se debe cerrar primero el último *if* que se abrió. Es muy importante también tabular el código correctamente para que las anidaciones sean legibles.

El código podría ser:

```
if (x==1) {
    instrucciones
    ...
}
else {
    if(x==2) {
            instrucciones
            ...
    }
    else {
            if(x==3) {
                    instrucciones
                    ...
            }
    }
}
```

Una forma más legible de escribir ese mismo código sería:

```
if (x==1) {
    instrucciones que se ejecutan si x vale 1
    ...
}
else if (x==2) {
    instrucciones que se ejecutan si x no vale 1 y vale 2
    ...
}
else if (x==3) {
    instrucciones que se ejecutan si x no vale ni 1 ni 2 y vale 3
    ...
}
else{
    instrucciones que se ejecutan si x no vale ni 1 ni 2 ni 3
}
```

dando lugar a la llamada instrucción **if-else-if**.

(5.6.3) instrucción while

bucle while

La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten mientras se cumpla una determinada condición. Los bucles **while** agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa.

La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while

Sintaxis:

```
while (expresión lógica) {
    sentencias que se ejecutan si la condición es verdadera
}
```

El programa se ejecuta siguiendo estos pasos:

(1) Se evalúa la expresión lógica

(2) Si la expresión es verdadera ejecuta las sentencias, sino el programa abandona la sentencia **while**

(3) Tras ejecutar las sentencias, volvemos al paso 1

Ejemplo (escribir números del 1 al 100):

```
var  i=1;
while (i<=100){
    document.write(i);
    i++;
}
```

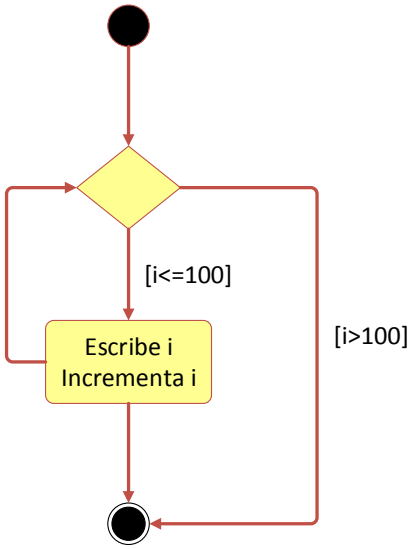


Ilustración 5-4, Diagrama UML de actividad del bucle while

bucles con contador

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles están controlados por un contador (o incluso más de uno). El contador es una variable que va variando su valor (de uno en uno, de dos en dos,… o como queramos) en cada vuelta del bucle. Cuando el contador alcanza un límite determinado, entonces el bucle termina.

En todos los bucles de contador necesitamos saber:

(1) Lo que vale la variable contadora al principio. Antes de entrar en el bucle

(2) Lo que varía (lo que se incrementa o decrementa) el contador en cada vuelta del bucle.

(3) Las acciones a realizar en cada vuelta del bucle

(4) El valor final del contador. En cuanto se rebase el bucle termina. Dicho valor se pone como condición del bucle, pero a la inversa; es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita y no para que termine.

Ejemplo:

```
var i=10; /*Valor inicial del contador, empieza valiendo 10
            (por supuesto i debería estar declarada como entera, int) */

while (i<=200){  /* condición del bucle, mientras i sea menor de
                    200, el bucle se repetirá, cuando i rebase este
                          valor, el bucle termina */

   document.write(i+"<br>); /*acciones que ocurren en cada vuelta del
                          bucle, en este caso simplemente escribe el valor
                          del contador y generamos el salto de línea*/

   i+=10; /* Variación del contador, en este caso cuenta de 10 en 10*/
}
/* Al final el bucle escribe:
10
20
30
...
y así hasta 200
*/
```

bucles de centinela

Es el segundo tipo de bucle básico. Una condición lógica llamada centinela, que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.

Esa expresión lógica a cumplir es lo que se conoce como centinela y normalmente la suele realizar una variable booleana.

Ejemplo:

```
var   salir=false; /* En este caso el centinela es una variable
                          booleana que inicialmente vale falso */
var n;
while(salir==false){ /* Condición de repetición: que salir siga siendo
                          falso. Ese es el centinela.
                          También se podía haber escrito simplemente:
                          while(!salir)
                          */
   n= parseInt(Math.random()*5+1; // Lo que se repite en el
   document.write(i+"<br>);  // bucle: calcular un número
                          aletorio de 1 a 500 y escribirlo */
   salir=(i%7==0); /* El centinela vale verdadero si el número es
                          múltiplo de 7, cuando lo sea, saldremos*/
```

Comparando los bucles de centinela con los de contador, podemos señalar estos puntos:

- Los bucles de contador se repiten un número concreto de veces, los bucles de centinela no
- Un bucle de contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque, si está bien programado, alguna vez lo alcanzará)
- Un bucle de contador está relacionado con la programación de algoritmos basados en series.

Un bucle podría ser incluso mixto: de centinela y de contador. Por ejemplo imaginar un programa que escriba números de uno a 500 y se repita hasta que llegue un múltiplo de 7, pero que como mucho se repite cinco veces. Sería:

```
var  salir = false;  //centinela
var n;
var i=1; //contador
while (salir == false && i<=5) {
   n = parseInt(Math.random() * 500 + 1);
   document.write(n+"<br>");
   i++;
   salir = (n % 7 == 0);
}
console.log("Último número "+n)
```

(5.6.4) bucle do...while

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir el bucle al menos se ejecuta una vez. Es decir los pasos son:

(1) Ejecutar sentencias

(2) Evaluar expresión lógica

(3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while

Sintaxis:

```
do {
    instrucciones
} while (expresión lógica)
```

Ejemplo (contar de uno a 1000):

```
int i=0;
do {
    i++;
    document.write(i);
} while (i<1000);
```

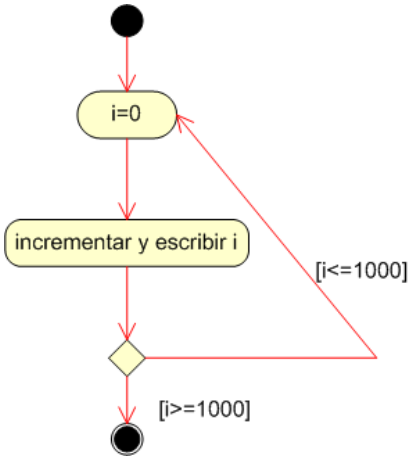


Ilustración 5-5, Diagrama de actividad del bucle do..while

Se utiliza cuando sabemos al menos que las sentencias del bucle se van a repetir una vez (en un bucle **while** puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio).

De hecho cualquier sentencia **do..while** se puede convertir en while. El ejemplo anterior se puede escribir usando la instrucción while, así:

```
int i=0;
i++;
console.write(i);
while (i<1000) {
    i++;
    console.write(i);
}
```

(5.6.5) bucle for

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

```
for(inicialización;condición;incremento){
    sentencias
}
```

Las sentencias se ejecutan mientras la condición sea verdadera. Además antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir el funcionamiento es:

(1) Se ejecuta la instrucción de inicialización

(2) Se comprueba la condición

(3) Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque *for*

(4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Ejemplo (contar números del 1 al 1000):

```
for(var i=1;i<=1000;i++){
    console.write(i+"<br>");
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle **while:**

```
var i=1; /*sentencia de inicialización*/
while(i<=1000) { /*condición*/
    console.write(i+"<br>");
    i++; /*incremento*/
}
```

Como se ha podido observar, es posible declarar la variable contadora dentro del propio bucle for. De hecho es la forma habitual de declarar un contador. De esta manera se crea una variable que muere en cuanto el bucle acaba.

(5.7) arrays

(5.7.1) qué es un array

Todos los lenguajes de programación disponen de un tipo de variable que es capaz de manejar conjuntos de datos. Es lo que se conoce comúnmente como arrays de datos. También se las llama listas, vectores o arreglos; pero todos ellos son nombres que tienen connotaciones que pueden confundirse con otros tipos de datos, por ello es más popular el nombre sin traducir al castellano: array.

La idea es solucionar un problema habitual al programar. Por ejemplo, supongamos que deseamos almacenar 25 notas de alumnos que luego se utilizarán en el código JavaScript. Eso es tremendamente pesado de programar. Manejar esos datos significaría estar continuamente manejando 25 variables. Los arrays permiten manejar los 25 datos bajo un mismo nombre. Ese nombre es el de la variable de tipo array (mejor dicho: el objeto de tipo array) que aglutina a todos los elementos, pero también nos permite la capacidad de acceder individualmente a cada elemento.

Los arrays son una colección de datos al que se le pone un nombre (por ejemplo ***nota***). Para acceder a un dato individual de la colección hay que utilizar su posición. La posición es un número entero, normalmente se le llama **índice**; por ejemplo ***nota[4]*** es el nombre que recibe el quinto elemento de la sucesión de notas. La razón por la que ***nota[4]*** se refiere al quinto elemento y no al cuarto es porque el primer elemento tiene índice cero.

Esto, con algunos matices, funciona igual en casi cualquier lenguaje. Sin embargo en JavaScript los arrays son **objetos**. Es decir no hay un tipo de datos array, si utilizamos **typeof** para averiguar el tipo de datos de un array, el resultado será la palabra **object**.

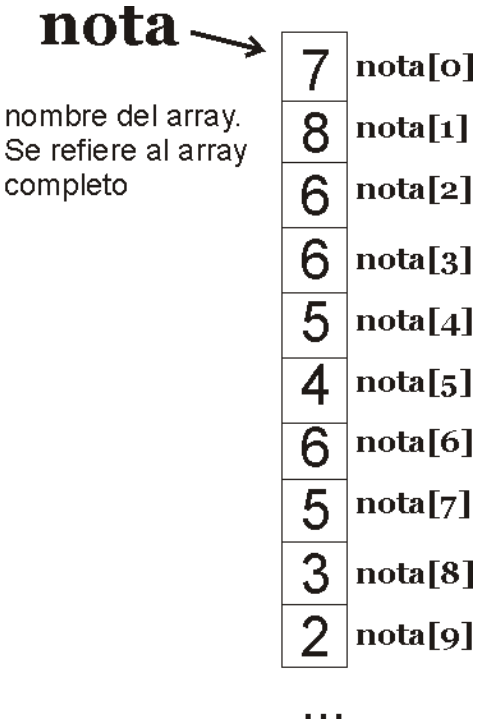


Ilustración 6, Ejemplo de array de notas

En algunos lenguajes como C, el tamaño del array se indica de antemano y no se puede cambiar; se habla entonces de arrays estáticos. Los arrays de JavaScript son totalmente dinámicos, pueden modificarse en tiempo de ejecución. Además los arrays de lenguajes como C o Java contienen datos del mismo tipo, mientras que en JavaScript los datos de un array pueden ser diversos.

(5.7.2) creación y uso básico de arrays en JavaScript

declaración y asignación de valores

Hay muchas formas de declarar un array. Por ejemplo si deseamos declarar un array vacío, la forma habitual es:

```
var a=[];
```

Los corchetes vacíos tras la asignación, significan array vacío. Si deseáramos indicar valores, se hace:

```
var nota=[7,8,6,6,5,4,6,5,3,2];
```

En esta caso la variable *nota* es un array de 10 elementos, todos números.

tipo de un array

Como se ha comentado anteriormente, este código:

```
document.write(typeof(nota));
```

escribe **object**. En JavaScript sólo hay cuatro tipos de datos (**number**, **boolean**, **string** y **object**), se considera que un array es un objeto (como ocurre con las expresiones regulares).

acceder a un elemento de un array

Si un array tiene valores, podemos acceder a uno de esos valore si individuales mediante el uso del índice dentro de un corchete. Por ejemplo (si nota es el *array* anterior):

```
document.write(nota[4]); //Escribe 5
```

elementos indefinidos

Es posible incluso declarar arrays y asignarles valores de esta forma:

```
var nota=[7,6,,5,,,8,9,,8];
```

Se permite no rellenar todos los valores, las comas que se quedan sin valor se tienen en cuenta. Así por ejemplo si con el array anterior, escribimos:

```
document.write(nota[2]);
```

El resultado es **undefined**, indicando que ese elemento está indefinido. Es más este código:

```
document.write(nota[2000]);
```

devuelve también **undefined** (y no error). De hecho se permite incluso códigos como:

```
var nombre=[];
nombre[3]="Fernando";
nombre[5]="Alicia";
document.write(nombre[3]);//Escribe Fernando
document.write(nombre[4]); //Escribe undefined
```

Directamente usamos índices y los no utilizados, son elementos indefinidos.

tamaño de un array

Puesto que un array es un objeto, tiene métodos y propiedades que podemos utilizar. La propiedad **length** sirve para obtener el tamaño de un array. Ejemplo:

```
var nota=[7,8,6,6,5,4,6,5,3,2];
document.write(nota.length); //Escribe 10
```

En el caso de arrays con elementos indefinidos, hay que tener en cuenta que **length** cuenta todos los elementos (y eso incluye a los indefinidos). Ejemplo:

```
var nota=[7,6,,5,,,8,9,,8];
document.write(nota.length); //Escribe 10, cuenta los indefinidos
```

Otro ejemplo:

```
var nombre=[];
nombre[3]="Fernando";
nombre[5]="Alicia";
document.write(nota.length); //Escribe 6
```

En este último ejemplo el resultado de **length** es seis, porque el último índice utilizado es el 5 (sexto elemento). De manera que los índices 5 y 3 están definidos, pero el 0,1,2 y 4 pasan a estar indefinidos.

arrays heterogéneos

Es perfectamente posible que un array contenga elementos de diferente tipo. Por ejemplo:

```
var a=[3,4,"Hola",true,Math.random()];
```

Ese tipo de array es el llamado heterogéneo y es muy habitual en JavaScript. Incluso podemos definir arrays de este tipo.

```
var b=[3,4,"Hola",[99,55,33]];
```

El elemento con índice 3 del array *b*, es otro array. De modo que esta instrucción es totalmente posible:

```
document.write(b[3][1]);//Escribe 55
```

En el ejemplo anterior, *b[3][1]* hace referencia al segundo elemento del cuarto elemento de *b*. Las posibilidades de arrays en JavaScript son espectaculares.

definir arrays de manera formal

Hay una forma de definir arrays más coherente con la programación orientada objetos que será más del gusto de los programadores conocedores del lenguaje Java. Se trata de usar la función constructora de objetos de clase Array (ya que un array es un objeto).

Los nombres de clase se ponen con la primera inicial en mayúsculas (no estamos obligados, pero es un normal muy importante a cumplir y que se cumple escrupulosamente en las clases estándares de JavaScript)), por eso es *Array* y no *array*.

Ejemplos de arrays definidos de esta forma son:

```
var a=new Array();
var b=new Array(7,8,6,6,5,4,6,5,3,2);
var d=new Array(3,4,"Hola",[99,55,33]);
var e=new Array(3,4,"Hola",new Array(99,55,33));
```

Sin embargo esta otra:

```
var c=new Array(7,6,,5,,,8,9,,8);
```

Da error, porque con new Array no se pueden dejar elementos sin defiunir.

(5.7.3) añadir y quitar elementos a los arrays

añadir al final

Es muy habitual añadir nuevos elementos a un array después del último elemento. La forma manual de hacerlo es:

```
a[a.length]=nuevoValor;
```

Si *a* es un array, *a.length* devuelve el tamaño del array; en ese índice colocaríamos el nuevo elemento.

Una forma más directa de hacer lo mismo (usando métodos de los arrays) es usar el método push:

```
var a=new Array(1,2,3,4,5);
a.push(6);
console.log(a);
//Escribe por consola: [ 1, 2, 3, 4, 5, 6 ]
```

añadir al principio

Otra función, nos permite añadir elementos al principio (en el índice cero) y desplazar el resto de elementos. Se trata de unshift:

```
var a=new Array(1,2,3,4,5);
a.unshift(6);
console.log(a);
//Escribe por consola: [ 6,1, 2, 3, 4, 5 ]
```

quitar último elemento

En este caso es la función pop la encargada:

```
var a=new Array(1,2,3,4,5);
a.pop();
console.log(a);
//Escribe por consola: [ 1, 2, 3, 4]
```

quitar primer elemento

Lo hace la función shift:

```
var a=new Array(1,2,3,4,5);
a.shift();
console.log(a);
//Escribe por consola: [  2, 3, 4,5]
```

quitar un elemento de una posición concreta

Otra posibilidad es eliminar un elemento cuya posición conocemos. El hueco que deja ese elemento lo ocupan los elementos a su derecha que se desplazarán hacia el nuevo hueco. El método encargado es delete al que se le indica el índice del elemento a eliminar:

```
var a=new Array(1,2,3,4,5);
delete a[1];
console.log(a);
//Escribe por consola: [  2, 3, 4,5]
```

(5.7.4) recorrer arrays

recorrido con un bucle de contador

La forma clásica de recorrer un array, es utilizar un bucle **for** en el que un contador permite recorrer cada elemento del array.

Por ejemplo supongamos que queremos recorrer el array llamado notas:

```
var nota=[2,3,4,5,6,7,3,4,2,5,6,7,8];
for(var i=0;i<nota.length;i++){
    document.write(nota[i]+" ");//Escriba la nota y deja un espacio
}
```

El resultado:

```
2 3 4 5 6 7 3 4 2 5 6 7 8
```

Pero si el array tiene elementos sin definir:

```
var nota=[2,3,,,6,,3,,,5,,,8];
for(var i=0;i<nota.length;i++){
    document.write(nota[i]+" ");//Escriba la nota y deja un espacio
}
```

Ahora el resultado es:

```
2 3 undefined undefined 6 undefined 3 undefined undefined 5 undefined undefined 8
```

es decir, los elementos **undefined** aparecen. Para saltarnos esos elementos, debemos cambiar el código (añadiendo una instrucción **if**):

```
var nota=[2,3,,,6,,3,,,5,,,8];
for(var i=0;i<nota.length;i++){
  if(nota[i]!=undefined)
     document.write(nota[i]+" ");//Escribe la nota y deja un espacio
}
```

ahora saldrán solo los números: 2 3 6 3 5 8

recorrido con bucle for..in

Existe una variante de bucle for, pensada para recorrer arrays donde sólo se muestran los elementos definidos del mismo (independientemente de su tamaño). Dicho bucle recorre cada elemento del array de modo que el índice del mismo se almacena en un contador.

La forma de este bucle es:

```
for(var índice in array){
    ....//instrucciones que se ejecutan por cada elemento recorrido
}
```

Ejemplo:

```
var nota=[2,3,,,6,,3,,,5,,,8];
for(var i in nota){
    document.write(nota[i]+" ");//Escribe la nota y deja un espacio
}
```

es una forma más sencilla que la comentada en el apartado anterior de mostrar sólo los elementos definidos en el array. Escribiría: 2 3 6 3 5 8

recorrido con función forEach

En las versiones modernas de JavaScript existe una forma de recorrer arrays que procede de los lenguajes como Python o Ruby. Se trata de usar el método forEach de los arrays.

Dicho método exige el uso de una función (que normalmente es anónima, se explican más adelante; es lo que se conoce como una función callback) y eso es lo que complica un poco su uso. Pero cuando se conocen las funciones, se convierte en una forma muy interesante de recorrer.

A la función interna se le pueden (no es obligatorio) indicar dos parámetros; el primero es el nombre de la variable que almacena cada elemento a medida que se recorrer, y el segundo el índice que dicho elemento tienen en el array. La sintaxis habitual es:

```
arrayARecorrer.forEach(function(elemento,índice){
    ...//código que se ejecuta por cada elemento del array
});
```

Ejemplo:

```
var nota=[2,3,,,6,,3,,,5,,,8];
nota.forEach(function(elemento,indice){
    document.write("La nota nº "+indice+" es "+elemento+"<br>");
});
```

Lo que resulta de este código es:

```
La nota nº 0 es 2
La nota nº 1 es 3
La nota nº 4 es 6
La nota nº 6 es 3
La nota nº 9 es 5
La nota nº 12 es 8
```

(5.7.5) comprobación de si una variable es un array

El operador instanceof de JavaScript nos permite averiguar la clase de un determinado objeto. Puesto que los arrays son objetos de clase Array, podemos usarle para averiguar si una variable es un array. Ejemplo:

```
var a=[1,2,3,4,5,6,7,8,9];
var b="Hola";
document.write(a instanceof Array); //Escribe true
document.write (b instanceof Array); //Escribe false
```

(5.7.6) métodos y propiedades de los arrays

Los arrays poseen numerosos métodos que facilitan su manejo. Se describen a continuación

propiedades

propiedad	significado
length	Devuelve el tamaño del array

métodos

propiedad	significado

concat(listaDeArrays)	Encadena una serie de arrays. Ejemplo:
	`var a=[1,2,3];`
	`var b=[4,5];`
	`var c=[6,7,8,9,10];`
	`var d=a.concat(b,c);`
	`console.log(d);`
	Por consola aparecería: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
	El array original (*a*, en este caso) no se modifica.

indexOf(elemento [,inicio])	Busca el elemento en el array y devuelve la posición. Si dicho elemento no está en el array el resultado será el valor -1. El segundo parámetro (opcional) permite buscar a partir de una posición concreta.

join([separador])	Convierte el array a string (texto) usando una coma para separar cada elemento del array o el carácter separador que se indique. Ejemplo:
	`var a=[23,"Hola",true,23.4,67];`
	`var s= a.join("-");`
	`document.write(s);`
	Escribe: 23-Hola-true-23.4-67

lastIndexOf(elemento [,inicio])	Método equivalente a indexOf, salvo que en este caso se comienza a buscar desde el final del array.

propiedad	significado
pop()	Elimina el último elemento del array (y le devuelve como resultado)
push(listaElementos)	Añade los elementos indicados al final del array
reverse()	Invierte los elementos del array (el último pasa a ser el primero, el penúltimo el segundo,…)
shift()	Elimina el primer elemento del array (y le devuelve como resultado)

slice(inicio[,fin])	Toma una porción del array. Toma desde el índice indicado por el elemento **inicio**, hasta el índice indicado por **fin** (sin incluir dicho elemento)l Si no se indica **fin**, se toma hasta el final del array.
	
	`var a=[1,2,3,4,5,6,7,8,9];`
	`var b= a.slice(2,5);`
	`console.log(b);`
	
	Por consola sale: [3, 4, 5]

sort([funciónDeOrdenación])	Ordena el array. El orden que toma es el de la tabla ASCII. Es decir, ordena bien el texto en inglés (aunque distingue en mayúsculas y minúsculas). Para otras ordenaciones se puede utilizar una función personal para explicar cómo ordenar (esa función tiene dos parámetros y devuelve un valores positivos para indicar si el primero es mayor que el segundo, cero para indicar que son iguales y negativos para indicar que el mayor es el segundo). Ejemplo para ordenar un array numérico: `array.sort(function(a,b){return a-b;});`

propiedad	significado

splice(inicio,número[,listaEltos])	Elimina del array elementos. Los elementos eliminados son los que van desde la posición indicada por inicio, y elimina el número de elementos indicados.
	
	El tercer parámetro, que es opcional, permite indicar una serie de elementos con los que se sustituirán los eliminados. Ejemplo:
	
	`var a=[1,2,3,4,5,6,7,8,9];`
	`a.splice(3,2);`
	`console.log(a); //escribe: [ 1, 2, 3, 6, 7, 8, 9 ]`
	
	Otro ejemplo:
	`var a=[1,2,3,4,5,6,7,8,9];`
	`a.splice(3,2,11,12,13,14,15,16);`
	`console.log(a);`
	`//escribe: [ 1, 2, 3, 11, 12, 13, 14, 15, 16, 6, 7, 8, 9 ]`

toString()	Convierte el array en forma de string (texto). Separa cada elemento con una coma.
unshift(listaElementos)	Añade los elementos indicados al inicio del array

(5.8) funciones

(5.8.1) introducción

En la programación clásica, las funciones fueron la base de la llamada **programación modular** que se llamada así porque permitía la posibilidad de que un programa se dividiera en un conjunto de módulos cada uno de los cuales se programaba de manera independiente.

Esta idea persiste actualmente y de hecho todos los lenguajes actuales poseen algún mecanismo de modularidad. En el caso de JavaScript (aparte de la creación de clases y objetos), se pueden crear funciones para conseguir dicha modularidad.

Una función es código JavaScript que realiza una determinada tarea a partir de unos datos de entrada. Dicha tarea por lo general consiste en devolver un determinado resultado a partir de los datos de entrada, o bien, realizar una acción aunque no devuelve ningún resultado.

Es decir, la idea es la de las funciones matemáticas. La ventaja es que la función puede ser invocada una y otra vez y que una misma función nos puede ayudar en diferentes programas, bastará con incluir el archivo en el que almacenemos las funciones.

(5.8.2) crear funciones

Las funciones requieren de los siguientes elementos:

- Un **identificador** (nombre) de la función. Que cumple las mismas reglas que los identificadores de variables. Además las funciones usan identificadores con letras en minúsculas.
- Una serie de **parámetros**, que son variables locales a la función que permiten almacenar los valores necesarios para que la función realice su labor. Se pueden crear funciones incluso sin parámetros.
- Un valor que se devuelve a través de la instrucción **return**. Aunque hay funciones (lo que en otros lenguajes se conoce como **procedimiento**) que no usan dicha instrucción (es decir no retornan ningún valor).
- Las instrucciones de la función, que irán entre las llaves de la misma. El llamado **cuerpo** de la función.

La sintaxis de creación de funciones en JavaScript es:

```
function nombre([listaParámetros]){
    ...cuerpo de la función
}
```

Ejemplo:

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

Esta función devuelve el cuadrado de un número, no es muy necesaria, pero ilustra cómo crear funciones. Una función más útil sería:

```
function escribirArray(a){
  if(Array.isArray(a)){
   document.write("[");
    for(i in a){
       document.write(a[i]+" ")
    }
   document.write("]");
  }
  else{
     document.write("No es un array");
  }
}
```

En este caso la función permite escribir en una página el contenido de un array. Esta función no devuelve ningún resultado.

Es recomendable declarar las funciones en un archivo externo si prevemos que nos van a ser útiles en diferentes páginas o si sólo se utilizan en el archivo actual, declararlas en el apartado de cabecera (**head**) de la página web.

(5.8.3) uso de funciones. invocar funciones

Para utilizar funciones basta con invocarlas enviando los parámetros con los valores necesarios para que la función realice su trabajo. Ejemplo (suponiendo que estamos utilizando las funciones anteriores):

```
var v1=9, v2=2.25, v3="Prueba";
var a=[2,4,6,7,9];
document.write(cuadrado(9)); //Escribe 81
document.write(cuadrado(v1)); //Escribe 81
document.write(cuadrado(v2)); //Escribe 5.0625
document.write(cuadrado(v3)); //Escribe NaN
document.write(cuadrado(v1)+cuadrado(3)); //Escribe 90
document.write(cuadrado(cuadrado(2))); //Escribe 16
escribirArray(a); //Escribe [2 4 6 7 9]
escribirArray(a.reverse());//Escribe [9 7 6 4 2]
escribirArray([3,4,cuadrado(3)]); //Escribe [3 4 9]
```

(5.8.4) modificar variables dentro de las funciones. paso por valor y por referencia

El paso por valor y pro referencia es uno de los temas que todo programador de lenguajes clásicos conoce. La cuestión procede de esta duda:

```
function f(x){
    x=3;
}
var v=9;
f(v);
document.write(v); //Escribe 9 y no 3
```

Aunque el parámetro ***x*** toma el valor de la variable ***v***, lo que hace es tomar una copia del mismo. Y por ello aunque modificamos su valor, sólo modificamos el valor de esa copia. La variable ***v*** original no se ha tocado y por ello seguirá valiendo ***9***. A esta forma de manejar los parámetros, se la llama **paso por valor**, porque el parámetro recoge el valor de la variable. En JavaScript todas las variables básicas (**boolean**, **number** y **string**) se pasan por valor.

Sin embargo en este otro ejemplo:

```
function f(x){
    x[2]=3;
}
var v=[1,1,1,1,1];
f(v);
console.log(v);  //Por consola sale: [1,1,3,1,1]
```

La variable ***v*** (un array) si se ha modificado. Es decir ***x***, no recibe una copia sino que recibe una referencia a la variable original. Dicho de otra forma: cambiar ***x*** es cambiar

v. Eso es un **paso por referencia**. En JavaScript todos los objetos, es decir variables de tipo **object** (los arrays son ***object***) se pasan por referencia.

Este funcionamiento es idéntico al del lenguaje **Java**.

(5.8.5) parámetros indefinidos

En JavaScript cuando se invoca a una función y no se da valor a todos los parámetros, los parámetros que no se han utilizado, toman el valor **undefined**. Es decir en cierto modo los parámetros (a diferencia de los lenguajes formales como **C** o **Java**) son opcionales.

Ejemplo:

```
function cuadrado(n){
    return n*n;
}
document.write(cuadrado());
//Escribe NaN porque n es undefined y ese valor no se puede multiplicar
```

Esto puede dar lugar a problemas por eso a veces se indica un valor en los parámetros para el caso en el que no se indique valor. Ejemplo:

```
function cuadrado(n){
    if(n==undefined) n=2;
    return n*n;
}
document.write(cuadrado()); //Escribe 4
```

(5.8.6) listas de argumentos

Las funciones de JavaScript permiten una libertad que no es nada común con otros lenguajes. Así podemos invocar a la función **escribirArray** de esta forma:

```
var array1=[1,2,3,4,5];
var array2=[6,7,8,9];
escribirArray(array1,array2); //Escribe [1 2 3 4 5]
```

Aunque se invoca a ***escribirArray*** con un segundo parámetro, no se le hace caso. Sólo se usa el primero. Pero lo interesante es que no ocurre ningún error.

Lo cierto es que incluso podríamos acceder a ese segundo parámetro a través de un array que se puede utilizar en todas las funciones y que se llama **arguments**. En dicho array el primer elemento es el primer parámetro que se pase a la función, el siguiente es el segundo y así sucesivamente. Al ser un array, el uso de **arguments.length** nos permite saber cuántos parámetros se han utilizado para invocar a la función.

Eso permite realizar funciones que trabajan con listas de argumentos. Por ejemplo supongamos que deseamos crear una función llamada ***escribirArrays***, que en lugar de escribir un array, es capaz de escribir una lista. El código sería (este código utiliza la función ***escribirArray*** anteriormente comentada que sirve para escribir por pantalla el contenido de un array

Ejemplo:

```
function escribirArrays(){
  //Bucle que recorre todos los argumentos
  for(var i=0;i<arguments.length;i++){
    if(Array.isArray(arguments[i])){
      escribirArray(arguments[i]);
      document.write("<br>");
    }
  }
}
var array1=[1,2,3,4,5];
var array2=[6,7,8,9];
escribirArray(array1,array2);
/* Escribe:
   [1 2 3 4 5]
   [6 7 8 9]
*/
```

(5.8.7) variables asignadas a funciones

En JavaScript se puede hacer que una variable represente a una función. Ejemplo:

```
var v=cuadrado; //la variable representa a la función cuadrado
document.write(v(3)); //Escribe 9, el cuadrado de 3
```

Lo que abre esta funcionalidad del lenguaje JavaScript es difícil de ver en este punto; pero a medida que se profundiza en el lenguaje, más interesantes son estas variables.

Lo curioso es que si escribimos este código:

```
document.write(typeof v); //Escribe function
```

v ya no es una variable, es una función. Esta idea admite que podamos utilizar funciones como parámetros de otras funciones, funciones que devuelven funciones,…

(5.8.8) funciones anónimas

JavaScript admite también un tipo especial de función llamada anónima. Estas funciones no tienen nombre y, por lo tanto no pueden ser invocadas. Son funciones de un solo uso; pero se utilizan mucho (y cada vez más) en JavaScript.

Un ejemplo de uso:

```
var f=function(x) {return x*x};
document.write(f(9)); //Escribe 81
```

Otro uso habitual es el que estas funciones aparezcan como parámetros en otras funciones preparadas para ello. Es el caso de la función forEach de los arrays, que admiten como parámetro una función que además puede tener dos parámetros, uno para recoger cada elemento del array que se recorre (a medida que se recorre) y el segundo el índice de cada elemento.

Ejemplo:

```
var a=[1,2,3,4,5,6];
a.forEach(function(elemento,índice){
  a[índice]*=2;
});
```

Cada elemento del array doblará su valor.

(5.9) objetos

(5.9.1) programación orientada a objetos en JavaScript

JavaScript admite el uso de objetos. No es un lenguaje completamente orientado a objetos (no cumple las reglas fundamentales de este tipo de programación) pero sí es un lenguaje **basado en objetos**. Estos significa que JavaScript maneja objetos. La cuestión es ¿qué es un objeto?

Un objeto es un tipo de variable compleja que posee tanto datos (llamados **propiedades**) como funciones (llamadas **métodos**). Las propiedades contienen la información de un objeto, los métodos permiten definir las acciones que podemos realizar con el objeto.

(5.9.2) acceso a propiedades y métodos

Si tenemos un determinado objeto, acceder a sus propiedades y métodos consiste en utilizar el operador punto (.), de manera que las propiedades se acceden en la forma:

```
objeto.propiedad
```

Cambiar el valor de una propiedad es posible por ejemplo así:

```
punto.x=9;
```

ahí hemos cambiado el valor de la propiedad ***x*** del objeto ***punto***.

Para utilizar métodos, la sintaxis es:

```
objeto.método([parámetros])
```

Ejemplo:

```
punto.sumar(2);
```

(5.9.3) objetos literales

Los objetos más sencillos que podemos crear en JavaScript son los llamados **literales** o **instancias directas**. Son objetos que se utilizan directamente sin indicar cómo funcionan.

Se pueden definir directamente sus propiedades sobre la marcha. Por ejemplo:

```
var punto=new Object(); //punto es un objeto, por ahora vacío
punto.x=5; //Definimos la propiedad x y le damos valor
punto.y=punto.x*2; //Definidmos la propiedad y que vale el doble de x (10)
```

Otra posibilidad (la más utilizada actualmente) es utilizar la notación de objetos conocida como **JSON** (***JavaScript Object Notation***) que pone entre llaves las propiedades y los valores del objeto. La sintaxis es:

```
{
    propiedad1 : valor1,
    propiedad2 : valor2,
    ...
}
```

Ejemplo (equivalente al anterior):

```
var punto={
    x : 5,
    y : 10
}
document.write(punto.y); //Escribe 10
```

Otro ejemplo más complejo sería:

```
var libro = {
    título : "Manual de UFOlogía",
    "n-serie" : "187C2", //Las propiedades pueden llevar espacios, guiones,
                         //etc. por lo que en esos casos se usan comillas
    autores : [ "Pedro Martínez","Ana León"], //Esta propiedad es un array
    editorial : { //La editorial es otro objeto
            nombre : "Grajal S.A.",
            pais : "España"
    }
};

document.write(libro.título); //Escribe: Manual de UFOLogía
document.write (libro.editorial.nombre); //Escribe Grajal S.A.
```

Como se ve en el ejemplo, con esta notación es posible dar nombres de propiedades con símbolos prohibidos en los identificadores (guiones, espacios en blanco,...). Es lo que le ocurre a la propiedad *"n-serie"*. Lo malo es que no es válida la notación:

```
document.write(libro."n-serie"); //¡¡¡¡Errror!!!
```

Acceder a este tipo de propiedades implica utilizar la notación de los arrays asociativos. Son arrays donde el índice no es un número sino el nombre de una propiedad. Es decir dentro de corchetes se indica la propiedad entre comillas. Ejemplo:

```
document.write(libro["n-serie"]); //Escribe 187C2
```

(5.9.4) definir objetos mediante constructores

En todos los lenguajes orientados a objetos, los constructores son funciones que permiten crear objetos. Esta forma, por lo tanto, de crear objetos será más del agrado de los programadores experimentados en la programación orientada a objetos.

La idea es que ahora no creamos directamente un objeto sino una **clase** un tipo de objeto. Crear un objeto a partir de su clase implica usar la acción **new**.

En estas funciones es muy habitual usar la expresión **this**. Esta palabra reservada del lenguaje representa al objeto actual y a través de ella podremos acceder a las propiedades y métodos del objeto actual.

En los nombres de las clases que declaramos a través de su constructor, la primera letra del nombre se pone en mayúsculas. No es una regla obligatoria, pero sí muy recomendable.

Ejemplo:

```
function Punto(x,y){
   this.x=x;
   this.y=y;
}

var p=new Punto(2,3);
document.write(p.x+","+ p.y); //Escribe 2,3
```

En el ejemplo anterior, la función *Punto* es una función constructora de objetos de clase *Punto*. Para definir un objeto concreto (un *Punto* concreto), es decir una instancia de la clase Punto; se utiliza el operador new.

la expresión *this.x* nos permite acceder a la propiedad x del objeto que estamos creando. Lo mismo ocurre con *this.y*. Sin embargo *x* e *y*, sin la palabra **this** es el nombre que se ha dado a los parámetros necesarios para crear un punto.

(5.9.5) definir métodos

Hay dos formas de definir métodos.

definir métodos en objetos literales

En la notación JSON se admite definir funciones, gracias a la facilidad ya comentada de que las variables (y por tanto las propiedades) se puedan asignar a funciones (normalmente anónimas).

Por lo tanto en objetos literales podemos definir también funciones. Ejemplo:

```
var punto={
   x : 3,
   y : 4,
   escribir : function(){
      console.log("("+this.x+","+this.y+")");
   }
};

punto.escribir();//Escribe (3,4)
```

También es posible sin usar JSON (continuando el ejemplo anterior):

```
punto.sumar=function(v){
  this.x+=v;
  this.y+=v;
}
punto.sumar(4);
punto.escribir();//Escribe (7,8)
```

definir métodos en el constructor de clase

Si lo que deseamos es una clase de objetos **Punto**, entonces no nos interesa definir los métodos en la forma anterior. En este caso se define la función en el constructor de clase.

Ejemplo (equivalente al anterior):

```
function Punto(x,y){
  this.x=x;
  this.y=y;
  this.escribir=function(){
    document.write ("("+this.x+","+this.y+")");

  }
  this.sumar=function(v){
    this.x+=v;
    this.y+=v;
  }
}

var p=new Punto(2,3);
p.escribir(); //Escribe 2,3
p.sumar(2);
p.escribir(); //Escribe (4,5)
```

(5.9.6) recorrer las propiedades de un objeto

Puesto que los objetos pueden ser tratados como arrays asociativos, disponemos del bucle **for..in** visto en el apartado de los arrays para recorrer las propiedades de un array. La sintaxis sería:

```
for(propiedad in objeto){
    ...instrucciones que se ejecutan por cada propiedad
}
```

Ejemplo (usando el objeto libro visto anteriormente):

```
for(prop in libro){
    console.log("Propiedad: "+prop+" valor: "+libro[prop]);
}
```

Si disponemos de una vista de la consola de JavaScript, veríamos:

```
Propiedad: título, valor: Manual de UFOlogía
Propiedad: n-serie, valor: 187C2
Propiedad: autores, valor: Pedro Martínez,Ana León
Propiedad: editorial, valor: [object Object]
Propiedad: escribir, valor: function (){
    console.log(this.título+"; Autores:"+this.autores[0]+","+this.autores[1]);
  }
```

Como se observa, también se recorren los métodos.

(5.9.7) operador instanceof

Heredado de C++., JavaScript posee el operador **instanceof**. Sirve para saber si un objeto pertenece a una clase concreta. La sintaxis del operador es:

```
objeto instanceof clase
```

si el objeto pertenece a la clase indicada devolverá true. Por ejemplo:

```
document.write(p instanceof Punto); //Devuelve true si p es un Punto
```

(5.9.8) objetos predefinidos

Se trata de objetos y clases ya creadas que están disponible para su uso. Math

Math es el nombre de un objeto global predefinido y utilizable en cualquier código JavaScript que permite realizar cálculos matemáticos

Math

Realmente es lo que en Java se conoce como **clase estática.** las clases estáticas son clases que no permiten crear objetos a partir de ellas, sino que directamente poseen métodos y propiedades ya disponibles.

En la práctica se manejan como cualquier objeto, la diferencia es la letra mayúscula en el primer carácter del nombre del objeto.

constantes de Math

Permiten usar en el código valores de constantes matemáticas por ejemplo tenemos a Math.PI, que representa el valor de Pi. La lista completa es:

constante	significado
E	Valor matemático *e*
LN10	Logaritmo neperiano de 10
LN2	Logaritmo neperiano de 2
LOG10E	Logaritmo decimal de *e*
LOG2E	Logaritmo binario de *e*
PI	La constante π (Pi)
SQRT1_2	Resultado de la división de uno entre la raíz cuadrada de dos
SQRT2	Raíz cuadrada de 2

métodos de Math

método	significado
abs(n)	Calcula el valor absoluto de *n*.
acos(n)	Calcula el arco coseno de *n*
asin(n)	Calcula el arco seno de *n*
atan(n)	Calcula el arco tangente de *n*
ceil(n)	Redondea el número *n* (si es decimal) a su valor superior. Por ejemplo si el número es el *2.3* se redondea a *3*
cos(n)	Coseno de *n*
exp(n)	*e* elevado a *n*: e^n
floor(n)	Redondea el número *n* (si es decimal) a su valor inferior. Por ejemplo si el número es el *2.8* se redondea a *2*
log(n)	Calcula el logaritmo decimal de *n*
max(a,b)	*a* y *b* deben de ser dos números y esta función devuelve el mayor de ellos.
min(a,b)	*a* y *b* deben de ser dos números y esta función devuelve el menor de ellos.
pow(a,b)	Potencia. Devuelve el resultado de a^b
random()	Devuelve un número aleatorio, decimal entre cero y uno-
round(n)	Redondea n a su entero más próximo. *round(2.3)* devuelve *2* y *round(2.5)* devuelve *3*
sin(n)	Devuelve el seno de *n*
sqrt(n)	Raíz cuadrada de *n*
tan(n)	Tangente de *n*

Date

Es el nombre de la clase preparada para manejar fechas. Crear un objeto de fecha es usar el constructor de Date (que tiene varios en realidad). Por ejemplo:

```
var hoy=new Date();
document.write(hoy);
```

Escribiría algo como:

```
Mon May 27 2013 02:45:14 GMT+0200 (Hora de verano romance) ;
```

Podemos crear objetos de fecha creándoles con un valor concreto de fecha. Se usaría la sintaxis:

```
new Date(año,mes,día,hora,minutos,segundos,ms);
```

Ejemplo:

```
var d=new Date(2013,5,27,18,12,0,0);
document.write(d);
```

Muestra:

```
Thu Jun 27 2013 18:12:00 GMT+0200 (Hora de verano romance) ;
```

Hay una tercera opción que es crear una fecha a partir de un número que simboliza el número de milisegundos transcurridos desde el 1 de enero de 1970. Ejemplo:

```
var d2=new Date(1000);
document.write(d2);
```

Saldría:

```
Thu Jan 01 1970 01:00:01 GMT+0100 (Hora estándar romance) ;
```

métodos

Usan el nombre de un objeto para modificar o mostrar sus valores.

Ejemplo:

```
var d=new Date();
document.write(d.getFullYear()); //Escribe 2013, año de la fecha d
```

método	significado
getDate()	Devuelve el día del mes de la fecha (de 1 a 31)
getUTCDate()	Devuelve el día del mes de la fecha universal
getDay()	Obtiene el día de la semana de la fecha (de 0 a 6)
getUTCDay()	Obtiene el día de la semana de la fecha universal
getFullYear()	Obtiene el año (con cuatro cifras)
getUTCFullYear()	Obtiene el año (con cuatro cifras). La diferencia es que UTC usa la fecha global (lo que corresponda al meridiano de Greenwich.
getHours()	La hora de la fecha (número que va de 0 a 23)
getUTCHours()	Formato universal de la anterior (la hora va de cero a 23)
getMiliseconds()	Milisegundos en la fecha actual
getUTCMiliseconds()	Milisegundos en la fecha actual pasada al formato universal
getMinutes()	Minutos
getUTCMinutes()	Minutos en el formato universal
getMonth()	Número de mes; de 0, enero, a 11, diciembre,
getUTCMonth()	Número de mes en formato universal
getSeconds()	Segundos
getUTCSeconds()	Segundos en formato universal
getTime()	Valor en milisegundos de la fecha. Número de segundos transcurridos desde el 1 de enero de 1970, respecto a esa fecha
getTimezoneOffset()	Minutos de diferencia sobre la hora universal (la del meridiano de Greenwich)
setDate(día)	Modifica el día del mes de la fecha
setUTCDate(día)	Versión universal
setFullYear(año)	Modifica el año de la fecha
setUTCFullYear(año)	Versión universal

método	significado
setHours(hora)	Modifica la hora
setUTCHours(hora)	Versión universal
setMilliseconds(ms)	Modifica los milisegundos
setUTCMilliseconds(ms)	Versión universal
setMinutes(minutos)	Modifica los minutos
setUTCMinutes(minutos)	Versión universal
setMonth(mes)	Modifica el número de mes
setUTCMonth(mes)	Versión universal
setSeconds(segundos)	Modifica los segundos
setUTCSeconds(segundos)	Versión universal
setTime(milisegundos)	Modifica la fecha haciendo que valga la fecha correspondiente a aplicar el número de milisegundos indicados, contados a partir del 1 de enero de 1970
toDateString()	Muestra la fecha en un formato más humano de lectura
toGMTString()	Igual que la anterior, pero antes de mostrarla convierte la fecha a la correspondiente según el meridiano de Greenwich

toISOString()	Muestra la fecha en formato ISO: ***yyyy-mm-ddThh:mm:ss.sssZ***
	Ejemplo:
	`var d=new Date();`
	`document.write(d.toISOString());`
	Sale (si son las 1:18:40,268 segundos del día 27 de mayo de 2013 en el meridiano de Greenwich)
	`2013-05-27T01:18:40.268Z`

toJSON()	Muestra la fecha en formato JSON. Obtiene lo mismo que la anterior.
toLocaleDateString()	Muestra la fecha (sin la hora) en formato de texto usando la configuración local.
toLocaleString()	Muestra la fecha y hora en formato de texto usando la configuración local.
toTimeString()	Muestra la hora (sin la fecha) en formato de texto usando la configuración local.
toString()	Muestra la fecha en formato de texto usando la configuración habitual de JavaScript
toUTCString()	Versión en formato universal de la anterior.

métodos estáticos de Date

Los métodos se usan como con **Math**, es decir usan **Date.nombreMétodo**, lo que se conoce como método estático. Lista:

método	significado

método	significado
now()	Fecha actual en milisegundos desde el día 1 de enero de 1970
parse(objetoFecha)	Obtiene una representación en forma de texto de la fecha.
UTC(año,mes,día, horas, minutos, segundos,ms)	Consigue, de la fecha indicada, la forma equivalente en la que tendremos los milisegundos transcurridos desde el 1 de enero de 1970.

(5.10) manejo de eventos HTML

(5.10.1) introducción

El lenguaje HTML dispone de la posibilidad de asociar comportamientos de usuario en el documento a código JavaScript. Es lo que se conoce como el manejo de eventos del documento.

Los eventos son sucesos que ocurren en el documento por lo general por la acción del usuario. Por ejemplo cuando el usuario hace clic a la etiqueta o cuando pulsa una tecla o cuando se ha terminado de cargar el elemento.

Hay eventos comunes a casi todas las etiquetas (hacer clic, mover el cursor encima), pero otros son de etiquetas concretas, como que se active el control de formulario en el que vamos a escribir.

(5.10.2) manejo de eventos

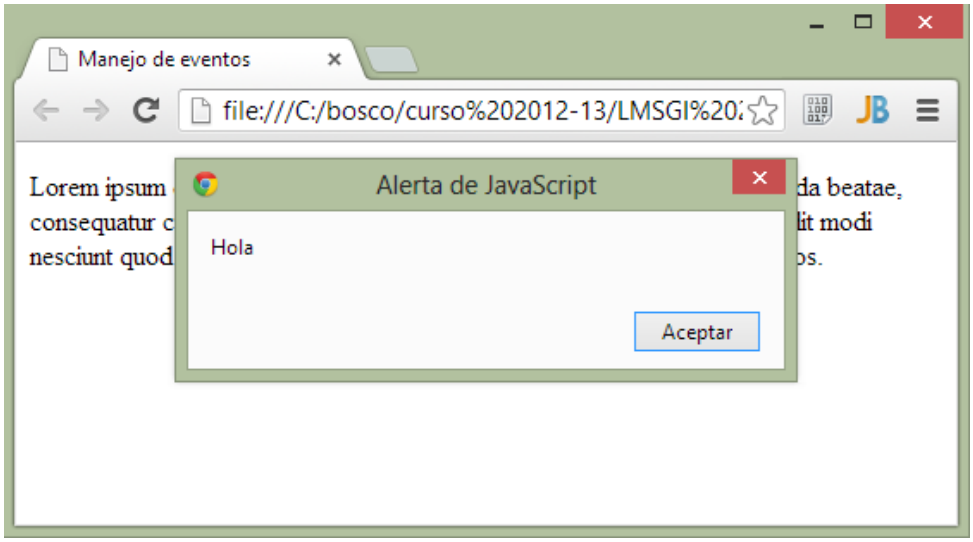
Supongamos que deseamos que ocurra algo cuando el usuario haga clic en un determinado párrafo. Lo que hay que hacer es asociar el evento **onclick** a un código JavaScript.

Casi siempre se asocia el evento a una función de JavaScript que habremos construido previamente.

Ejemplo (al hacer clic en el párrafo aparece el mensaje "Hola"):

```
<!doctype html>
<html lang="es">
<head>
   <meta charset="UTF-8">
   <title>Manejo de eventos</title>
</head>
<body>
<p onclick="alert('Hola');">Lorem ipsum dolor sit amet, consectetur adipisicing elit.
   Aperiam assumenda beatae, consequatur cumque debitis
   delectus et explicabo, fugiat fugit harum impedit modi
   nesciunt quod quos reprehenderit saepe temporibus voluptates?
   Dignissimos.</p>
</body>
</html>
```

Al hacer clic en el párrafo ocurriría esto:



(5.10.3) lista de eventos

provocados por el uso del ratón

evento	ocurre cuando...
onclick	el usuario hace clic en el elemento
ondblclick	el usuario hace doble clic en el elemento
onmousedown	el usuario pulsa el ratón sobre el elemento. Al soltar se produce onmouseup
onmouseup	el usuario suelta el ratón tras haberle pulsado (onmousedown)
onmouseover	el usuario mueve el ratón sobre el elemento
onmouseover	el usuario mueve el ratón hacia fuera del elemento

provocados por el teclado

evento	ocurre cuando...
onkeypress	el usuario pulsa una tecla
onkeydown	el usuario está pulsando una tecla
onkeyup	el usuario suelta la tecla que estaba pulsando

provocados por la acción en el propio navegador

evento	ocurre cuando...
onabort	se cancela la carga del elemento
onerror	si la carga del elemento no se ha hecho correctamente
onload	cuando el elemento se ha terminado de cargar
onresize	el tamaño del documento cambia (porque el usuario modifica el tamaño de la ventana)
onscroll	cuando el usuario se desplaza por el elemento

evento	ocurre cuando...
onunload	cuando se abandona la carga de la página (porque el usuario se va a otra o cierra la ventana)

eventos de formulario

Presentes sólo en los controles de formulario

evento	ocurre cuando...
onblur	cuando el control del formulario pierde el foco (el usuario abandona el cuadro de texto, casilla de verificación,...)
onchange	se ha modificado el contenido del cuadro
onfocus	el control del formulario obtiene el foco
onreset	se restablece (mediante botón *reset*) el contenido del formulario
onselect	cuando se selecciona texto dentro del control del formulario
onsubmit	el usuario pulsa el botón de enviar

(5.11) DOM

(5.11.1) introducción

DOM es la abreviatura de **Document Object Model**, el modelo de objetos del documento. En definitiva es la forma que dispone JavaScript para acceder a los elementos de una página web; esta forma de acceder se basa en objetos JavaScript.

El DOM entiende que una página es un árbol de elementos cuya raíz es el documento. De esta forma, la página:

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Documento HTML</title>
</head>
<body>
 <section id="s1">
   <h1>Título</h1>
   <p>Ejemplo de documento <strong>DOM</strong>Visión en árbol</p>
 </section>
 <section id="s2">
   <p>Aquí sólo hay texto</p>
 </section>
</body>
</html>
```

Se puede entender de esta forma:

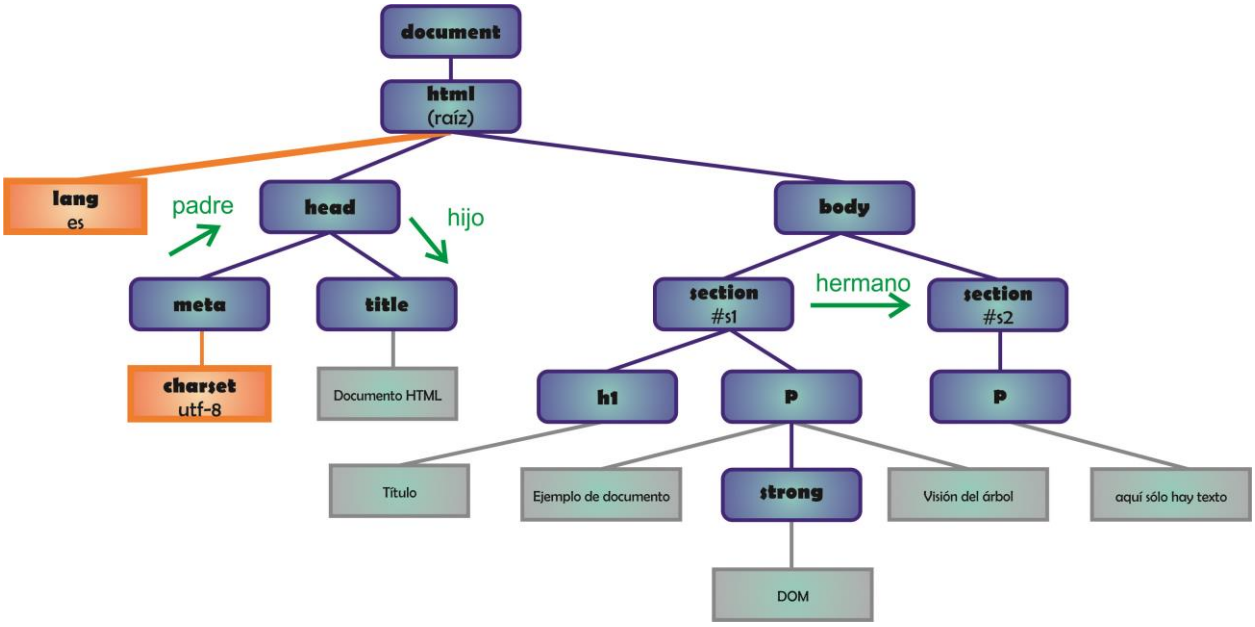


Ilustración 7, Esquema del DOM del documento anterior. En azul, los elementos, en naranja los atributos y en gris los textos. Se marcan en verde algunas relaciones entre los nodos

En la ilustración anterior, se observa el árbol de objetos que compone el documento anterior. Todo objeto tiene un padre (**parent**). Hay distintos tipos de objetos: elementos, atributos,.. además incluso hay diferentes tipos de objetos elementos (objetos de tabla, de formulario,…).

Esto permite manipular cada parte de una página manejando el objeto que la representa; acceder por ejemplo al texto de un elemento es acceder a la propiedad del objeto que representa a ese elemento y que contiene el texto de dicho elemento.

(5.11.2) objeto document

Como se observa en el esquema anterior, todos los objetos del DOM parten de un objeto superior que representa al propio documento. Se trata del objeto llamado **document**. Este objeto nos permite utilizar diversos métodos para trabajar con el documento.

(5.11.3) selección de objetos del DOM

Ahora la cuestión está en cómo seleccionar un objeto concreto del DOM. Ese permitirá asignar, por ejemplo, dicho objeto a una variable que nos permitirá manipular las propiedades del objeto gracias a los métodos que dicho objeto nos brinda. Podremos cambiar sus propiedades de formato (incluso todo su CSS), contenido, etc.

Se comentan a continuación las técnicas para seleccionar elementos del documento.

selección por ID

Es la forma de seleccionar elementos más utilizada. Todas las etiquetas HTML disponen del atributo **id** en el cual podemos almacenar el valor de dicho elemento. Este valor además es único, por lo que este modo de selección permite seleccionar un elemento concreto.

La forma de hacerlo es utilizar el método **getElementById** disponible en el objeto **document**. Por ejemplo:

```
var seccion1=document.getElementById("s1");
```

selección por nombre

El atributo **name** se ideó con el mismo propósito que **id**, pero no tiene su potencia. Puede haber dos elementos con el mismo **name** y además la W3C no recomienda su uso.

Hoy en día está utilizada en los controles de formulario (porque comunican su información con los servidores a través de este atributo).

document posee el método **getElementByName** al que se le pasa el nombre del elemento a seleccionar y permite seleccionar dicho elemento.

selección por tipo de etiqueta

En este caso se seleccionan los elementos del tipo indicado (**p**, **h1**, **img**, **a**, etc.). Lo normal al seleccionar de esta forma es que se seleccionen varios elementos (todos los que sean de ese tipo) por lo tanto lo que obtendremos no será un solo objeto sino un **array de objetos**.

El método para realizar esta forma de selección es **getElementsByTagName**. Ejemplo:

```
var parrafos= document.getElementsByTagName("p");
alert("Número de párrafos del document: "+parrafos.length);
```

seleccionar por la clase CSS

Otra opción (similar a la anterior) es la posibilidad de seleccionar los elementos que pertenecen a una determinada clase CSS. Es decir seleccionar aquellos elementos cuyo contenido del atributo **class** es el que indiquemos.

El método que lo hace es **getElementsByClassName**. Ejemplo de uso:

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Documento HTML</title>
  <style type="text/css">
    .rojo{
      color:red;
    }
  </style>
  <script type="text/javascript">
    function f1(){
      var elementos= document.getElementsByClassName("rojo");
      alert("Número de párrafos rojos: "+elementos.length);

    }
  </script>
</head>
<body onload="f1()">
```

```
<p class="rojo">Lorem ipsum dolor sit amet, consectetur adipisicing elit. Consequatur dicta nisi non quam quibusdam! Aspernatur aut
   cumque dolorum eligendi facere in, inventore laudantium neque, nesciunt nobis nostrum voluptates. In, nam.</p>

<p>Blanditiis cumque dolorum id impedit molestias quae, ut? Alias amet aut fuga laboriosam magnam odit omnis optio
   quaerat qui. Alias architecto esse incidunt maiores neque pariatur quam, sed suscipit vero?</p>

<p class="rojo">Ad alias culpa cupiditate dolorem doloribus eos est fugiat iusto laborum laudantium magnam maxime modi natus non
   pariatur perspiciatis praesentium repellat, repudiandae soluta tempora tempore unde veniam vitae voluptatem
   voluptatum?</p>

<p class="rojo">A accusantium aliquam atque, beatae debitis doloribus eaque eligendi, excepturi fuga fugiat harum in incidunt ipsa
   iste nemo nesciunt nulla possimus quaerat ratione similique suscipit voluptatem voluptatum? Beatae, quis, totam?</p>

<p>Aliquid aperiam blanditiis culpa, cumque debitis, delectus doloremque dolorum eligendi id illo maiores minima modi
   molestiae molestias, nam non nostrum placeat saepe soluta sunt suscipit tempore vel vero vitae voluptatum.</p>

<p>Ab animi aperiam atque consectetur consequatur cumque debitis deserunt dolorem eaque excepturi fuga harum ipsum iure
   maxime molestias natus, nobis nostrum optio perspiciatis, qui recusandae repellat repellendus sapiente vitae
   voluptas!</p>
</body>
</html>
```

Al ejecutar este código (observar el código JavaScript) tras la carga de la página aparece un cuadro en el que se nos informa de que hay tres párrafos rojos.

El problema de este método es que hay unos cuantos navegadores (especialmente los antiguos) que no le reconocen. Pasa con alguno de los métodos comentados anteriormente, pero en este caso es más agudo el problema (especialmente en internet Explorer)

usar selectores CSS

El lenguaje CSS (especialmente CSS3) aporta numerosos mecanismos de selección avanzada de elementos. Es conveniente conocer esa sintaxis porque es muy potente.

El método de document que permite seleccionar de esta forma es querySelectorAll. Lo malo es que muchos navegadores no reconocen este método.

Ejemplos de ella son:

ejemplo de selector	selecciona…
document.querySelectorAll("#seccion1")	El elemento con el identificador #seccion1
document.querySelectorAll("p[lang='es']")	Los párrafos que usen el atributo ***lang*** con valor ***es***.
document.querySelectorAll("section p")	Los párrafos dentro de secciones
document.querySelectorAll("p.rojo")	Párrafos de clase ***rojo***
document.querySelectorAll((".rojo")	Todos los elementos de clase rojo
document.querySelectorAll("h1,h2,h3")	Títulos de tipo **h1**, **h2** o **h3**

Para conocer todos los posibles selectores hay que estudiar con detalle los selectores de CSS.

uso de los elementos como nodos

En el árbol de los elementos de una página, podemos entender que los elementos, atributos e incluso el texto son nodos del árbol y que entre los nodos del árbol tenemos padres (**parent**), hijos (**children**) y hermanos (**siblings**).

Esta forma de ver los elementos nos permite acceder a propiedades mediante las cuales, a partir de un elemento podemos seleccionar otros que tengan relación de parentesco con él.

propiedad	uso
attributes	Selecciona todos los atributos del elemento. Devuelve un array de objetos atributos
childNodes	Devuelve un array con todos los elementos que cuelgan del actual
firstChild	Selecciona el primer elemento hijo del actual
lastChild	Selecciona el último elemento hijo del actual
nextSibling	Selecciona el siguiente hermano siguiendo el árbol DOM
nodeName	Obtiene el nombre del nodo. Si es un elemento devuelve su tipo, si es un atributo su nombre.
nodeType	Devuelve el tipo de nodo. Si lo usamos sobre un elemento devuelve 1, si es un atributo 2, si es un texto 3. Las constantes pertenecientes a los objetos de nodo: **ELEMENT_NODE**, **ATTRIBUTE_NODE** y **TEXT_NODE** están asociadas a esos valores.
parentNode	Obtiene el padre del nodo al que se aplica este método
previousSibling	Obtiene el nodo del mismo nivel, anterior en el árbol DOM
textContent	Obtiene el contenido (sólo el texto) del nodo, sea del tipo que sea. También permite modificarle.

this

Se ha comentado anteriormente, pero hay que tener en cuenta que la palabra **this**, selecciona al objeto actual. Eso permite hacer código como este:

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejemplo de innerHTML</title>
  <script type="text/javascript">
    function mostrarContenido(e){
      alert("El contenido del elemento es "+ e.textContent);
    }
  </script>
</head>
<body>
  <p onclick="mostrarContenido(this);">Párrafo nº 1</p>
  <p onclick="mostrarContenido(this);">Párrafo nº 2</p>
</body>
</html>
```

Al hacer clic en el primer párrafo se mostrará el texto *"Párrafo nº 1"*, en el segundo mostrará *"Párrafo nº 2"*

(5.11.4) modificar los elementos HTML

Lógicamente la razón para desear seleccionar elementos HTML, es para poder modificar u observar sus propiedades. Tras seleccionar un elemento HTML podemos realizar numerosas operaciones con él, las más importantes se comentan a continuación.

modificar el contenido

Se entiende que el contenido de un elemento HTML es más contenido HTML. Es decir que puede contener otros elementos HTML y también texto.

La propiedad que se usa habitualmente para modificar el contenido de un elemento es **innerHTML**. Con ella podemos examinar desde JavaScript el contenido de un elemento, pero también podemos modificar su contenido (incluido el código HTML).

Ejemplo:

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejemplo de innerHTML</title>
  <script type="text/javascript">
    function colorearParrafo(parrafo){
      var contenidoAnterior=parrafo.innerHTML;
      parrafo.innerHTML="<div style='color:red'>"+
          contenidoAnterior+
          "</div>";
    }
  </script>
</head>
<body>
  <p onclick="colorearParrafo(this);">Lorem ipsum dolor sit amet, consectetur
    adipisicing elit. Adipisci, aperiam cumque dolore ea earum eum perspiciatis
quas ratione
    repellat soluta veniam veritatis voluptas? At doloribus incidunt neque tenetur?
    Aut, dolorum?</p>
  <p onclick="colorearParrafo(this);">Amet autem dolorem ea, eos iusto
molestiae nihil nobis
    veritatis? Consequuntur, deserunt enim error eveniet ex explicabo illum ipsam
molestias
    nesciunt odit placeat possimus quae, quasi qui quod unde vitae?</p>

  <p onclick="colorearParrafo(this);">Atque commodi corporis cumque fugiat
illo ipsa laboriosam repellendus,
    sunt veritatis voluptate! Adipiscia liquid aspernatur, at autem dolorem
    expedita facilis hic labore necessitatibus quibusdam ratione sint
    voluptates. At, atque voluptatibus?
  </p>
</body>
</html>
```

cambiar el valor de un atributo

En este caso es muy sencillo. Basta con considerar que los atributos son propiedades de los elementos. La forma de modificar un atributo es:

```
elemento.atributo=valor;
```

Ejemplo:

```
imagen.src="boton2.jpg";
```

Si *imagen* es un elemento de tipo IMG, estamos cambiando su contenido.

cambiando el CSS de un elemento

En realidad se trata del mismo caso que el anterior. Sólo que la propiedad **style** (referida al código CSS de un elemento). Así podemos cambiar el código JavaScript del ejemplo visto en el ejemplo de código del apartado ***modificar el contenido*** visto anteriormente, para que quede de esta forma:

```
function colorearParrafo(parrafo){
        parrafo.style.color="red";
}
```

A través de la propiedad **style** podemos acceder a la mayoría de propiedades CSS. Lo que abre numerosas posibilidades para dar mayor dinamismo a la página web.

manejar todas las propiedades

No todos los atributos de un elemento son accesibles mediante las propiedades anteriores. Por ello hay disponible un método muy interesante para los elementos HTML. Se trata de **setAttribute**. La sintaxis de este método es:

```
elemento.setAttribute(stringAtributo, stringValor);
```

Es decir, se le indica como texto (luego entre comillas), el atributo a modificar y, también entre comillas, el nuevo valor. Ejemplo:

```
parrafo.setAttribute("class","rojo");
```

Lo cual aplicaría la clase CSS ***rojo*** (si existe) al objeto (se supone que asociado a un elemento HTML) ***parrafo***.

Si lo que deseamos es simplemente obtener el valor de un atributo, entonces el método es **getAttribute**, que sólo tiene el primer parámetro y devuelve el valor de ese atributo. Ejemplo:

```
alert(parrafo.getAttribute("class"));
```

Mostrará en un cuadro la clase de ese elemento.

usar propiedades de eventos en los elementos

Los propios eventos son atributos de los elementos (como ya se ha comentado) y eso permite una forma muy potente de trabajar con eventos. Así la propiedad **onclick** de un elemento permite asignar una función a este evento.

Esto ayuda a simplificar el código. Por ejemplo:

```
<script type="text/javascript">
 var parrafos=document.getElementsByTagName("p");
 for(i in parrafos){
    parrafos[i].onclick=function(){this.style.color="red"};
 }
</script>
```

Este potente script, permite colorear a cualquier párrafo de color rojo encuanto le hagamos click (sin tener que definir el evento onclick en cada uno de ellos).

añadir contenido en una posición concreta

Volviendo a considerar a los elementos de una página web como nos de un árbol, hay diversos métodos de los nodos que nos permiten añadir y quitar contenido de la página web basándonos en la estructura en árbol del DOM. Son:

método	significado
appendChild(nodo)	Hace que el nodo indicado se coloque como último hijo del elemento
cloneNode(nodo)	Duplica el nodo indicado
compareDocumentPosition(nodo)	Compara las posiciones relativas de los nodos
hasAttributes()	Devuelve verdadero si el elemento tiene atributos definidos
hasChildNodes()	Devuelve verdadero si el elemento tiene hijos
insertBefore(nodo)	Añade el nodo antes del primer hijo del elemento
isEqualNode(nodo)	Devuelve verdadero si dos nodos son iguales
isSameNode(nodo)	Devuelve verdadero si los dos nodos son iguales
normalize()	Borrar los nodos de texto vacíos y une los nodos de texto adyacentes
removeChild(nodo)	Borra el nodo hijo indicado y le devuelve como resultado (si el nodo no existe, devuelve null)
replaceChild(nodoViejo,nodoNuevo)	cambia el nodo viejo por el nuevo nodo dentro de los hijos del elemento

crear nuevos nodos en el árbol

Muchos de los métodos anteriores tienen sentido si creamos elementos nuevos. Para ello el objeto document dispone del método createElement, al que se le pasa (como texto entrecomillado el tipo de elemento que estamos creando. Ejemplo:

```
document.createElement("p");//crea un nuevo párrafo
```

Aún más, disponemos del método createTextNode para crear nuevos nodos de tipo texto. Así este código:

```
var parrafo=document.createElement("p");//crea un nuevo párrafo
var texto=document.createTextNode("Rosa, rosae");
parrafo.appendChild(texto); //Añade texto al párrafo
document.getElementByTagName("body")[0].appendChild(parrafo);
//Añade el párrafo al final del body
```

En el ejemplo anterior, se consigue que al final del documento haya un párrafo que diga *Rosa, rosae*