UNIDAD 4

LENGUAJES DE SCRIPT EN CLIENTE

1. INTRODUCCIÓN

Cuando hablamos de tecnologías empleadas en lenguajes de programación web podemos citar dos grupos básicos: **client-side** y **server-side**. Las tecnologías client-side son aquellas que son ejecutadas en el cliente, generalmente en el contexto del navegador web. Cuando los programas o tecnologías son ejecutadas o interpretadas por el servidor estamos hablando de programación server-side.

Cada tipo general de programación tiene su propio lugar y la mezcla es generalmente la mejor solución. Cuando hablamos de lenguajes de programación en clientes web, podemos distinguir dos variantes:

- Lenguajes que nos permiten dar formato y estilo a una página web (HTML, CSS, etc.).
- Lenguajes que nos permite aportar dinamismo a páginas web (lenguajes de scripting).

Vamos a profundizar en el lenguaje de scripting más utilizado en la actualidad, en el desarrollo de aplicaciones web en lado del cliente, JavaScript. Además, está soportado mayoritariamente por todas las plataformas.

JavaScript es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas.

Una página web dinámica es aquella que incorpora efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

Tabla comparativa de lenguajes de programación web cliente – servidor

Lado del Cliente (client-side)	Lado del servidor (server-side)
✓ Aplicaciones de Ayuda. ✓ Programas del API del navegador. ➧ Plug-ins de Netscape. ➧ Pepper para Chrome. ➧ Controles ActiveX. ➧ Applets de Java. ✓ Lenguajes de scripting. ➧ JavaScript. ➧ VBScript.	✓ Scripts y programas CGI. ✓ Programas API del servidor. ➧ Módulos de Apache. ➧ Extensiones ISAPI y filtros. ➧ Servlets de Java. ✓ Lenguajes de scripting. ➧ PHP. ➧ Active Server Pages (ASP/ASP.NET). ➧ JavaScript (Librería Node.js) ➧ Ruby (Ruby on Rails) ...

El lenguaje que vamos a estudiar ahora se llama JavaScript, pero quizás habrás oído otros nombres que te resulten similares como JScript (que es el nombre que le dio Microsoft a este lenguaje).

JavaScript se diseñó con una sintaxis similar al lenguaje C y aunque adopta nombres y convenciones del lenguaje Java, éste último no tiene relación con JavaScript ya que tienen semánticas y propósitos diferentes.

JavaScript fue desarrollado originariamente por Brendan Eich, de Netscape, con el nombre de Mocha, el cual se renombró posteriormente a LiveScript y quedó finalmente como JavaScript.

Hoy en día JavaScript es una marca registrada de Oracle Corporation, y es usado con licencia por los productos creados por Netscape Communications y entidades actuales, como la fundación Mozilla.

1.1. Herramientas y utilidades de programación

La mejor forma de aprender JavaScript es tecleando el código HTML y JavaScript en un simple documento de texto.

Para aprender JavaScript no se recomiendan editores del estilo WYSIWYG (What You See is What You Get) como Dreamweaver o FrontPage, ya que estas herramientas están más orientadas a la modificación de contenido y presentación, y nosotros nos vamos a centrar más en el código fuente de la página.

La idea es decantarse por **editores** que posean características que te faciliten la programación web, como, por ejemplo:

- Sintaxis con codificación de colores. Que resalte automáticamente en diferente color o tipos de letra los elementos del lenguaje tales como objetos, comentarios, funciones, variables, etc.
- Verificación de sintaxis. Que te marque los errores en la sintaxis del código que estás escribiendo.
- Que te permita diferenciar los comentarios del resto del código.
- Que genere automáticamente partes del código tales como bloques, estructuras, etc.
- Que disponga de utilidades adicionales, tales como cliente FTP para enviar tus ficheros automáticamente al servidor, etc.

Dependiendo del sistema operativo que utilices en tu ordenador dispones de múltiples opciones de editores. Cada una es perfectamente válida para poder programar en JavaScript. Algunos ejemplos de editores web gratuitos son:

- Para Windows tienes: Brackets, Sublime Text, Notepad++, Light Table, Aptana Studio, Bluefish, Eclipse, NetBeans, etc.
- Para Macintosh tienes: Aptana Studio, Light Table, Light Table, Bluefish, Eclipse, KompoZer, Nvu, etc.
- Para Linux tienes: Brackets, Sublime Text, Light Table, Geany, KompoZer, Amaya, Quanta Plus, Bluefish, codetch, etc.

Otro de los componentes obligatorio para aprender JavaScript es el **Navegador Web**. No es necesario que te conectes a Internet para comprobar tus scripts realizados con JavaScript. Puedes realizar dicha tarea sin conexión. Esto quiere decir que puedes aprender JavaScript y crear aplicaciones con un ordenador portátil y desde cualquier lugar sin necesitar Internet para ello.

Una recomendación muy interesante es el disponer de 2 o 3 tipos de navegadores diferentes, ya que así podrás comprobar la compatibilidad de tu página web y ver si tu código fuente de JavaScript se ejecuta correctamente en todos ellos.

Otro aspecto muy importante es la **validación**. Puedes ahorrarte muchas horas de comprobaciones simplemente asegurándote de que tu código HTML es válido. Si tu código HTML contiene imperfecciones, tienes muchas posibilidades de que tu JavaScript o CSS no funcionen de la manera esperada, ya que ambos dependen de los elementos HTML y sus atributos. Cuanto más te ajustes a las especificaciones del estándar, mejor resultado obtendrás entre los diferentes tipos de navegadores.

El consorcio W3C, el cuál diseño el lenguaje HTML, desarrolló un validador que te permitirá chequear si tu página web cumple las especificaciones indicadas por el elemento DOCTYPE que se incluye al principio de cada página web que realices. El Validador será tu amigo y te permitirá realizar unas páginas más consistentes.

La dirección del Validador W3C es: https://validator.w3.org/

1.2. Glosario básico

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

1.3. Sintaxis

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

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

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

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

2. SINTAXIS BÁSICA

2.1. Integración de código JavaScript con HTML

Los navegadores web te permiten varias opciones de inserción de código de JavaScript. Podremos insertar código usando las etiquetas <script> </script> y empleando un atributo type indicaremos qué tipo de lenguaje de script estamos utilizando:

Por ejemplo:

```
<script type="text/javascript">
// El código de JavaScript vendrá aquí.
</script>
```

Otra forma de integrar el código de JavaScript es incrustar un fichero externo que contenga el código de JavaScript. Ésta sería la forma más recomendable, ya que así se consigue una separación entre el código y la estructura de la página web y como ventajas adicionales podrás compartir código entre diferentes páginas, centralizar el código para la depuración de errores, tendrás mayor claridad en tus desarrollos, más modularidad, seguridad del código y conseguirás que las páginas carguen más rápido. La rapidez de carga de las páginas se consigue al tener el código de JavaScript en un fichero independiente, ya que si más de una página tiene que acceder a ese fichero lo cogerá automáticamente de la caché del navegador con lo que se acelerará la carga de la página.

Para ello tendremos que añadir a la etiqueta `script` el atributo `src`, con el nombre del fichero que contiene el código de JavaScript. Generalmente los ficheros que contienen texto de JavaScript tendrán la extensión `.js`.

Por ejemplo:

```
<script type="text/javascript" src="tucodigo.js"></script>
```

Si necesitas cargar más de un fichero .js repite la misma instrucción cambiando el nombre del fichero. Las etiquetas de <script> y </script> son obligatorias a la hora de incluir el fichero .js. **Atención:** no escribas ningún código de JavaScript entre esas etiquetas cuando uses el atributo src.

Para **referenciar el fichero origen .js** de JavaScript dependerá de la localización física de ese fichero. Por ejemplo, en la línea anterior el fichero tucodigo.js deberá estar en el mismo directorio que el fichero .html. Podrás enlazar fácilmente a otros ficheros de JavaScript localizados en directorios diferentes de tu servidor o de tu dominio. Si quieres hacer una referencia absoluta al fichero, la ruta tendrá que comenzar por http://, en otro caso tendrás que poner la ruta relativa dónde se encuentra tu fichero .js

Ejemplos:

```
<script type="text/javascript"
src="http://www.tudominio.com/ejemplo.js"></script>
<script type="text/javascript" src="../js/ejemplo.js"></script>
```

(el fichero ejemplo.js se encuentra en el directorio anterior (../) al actual, dentro de la carpeta js/)

Cuando alguien examine el código fuente de tu página web verá el enlace a tu fichero .js, en lugar de ver el código de JavaScript directamente. Ésto no quiere decir que tu código no sea inaccesible, ya que simplemente copiando la ruta de tu fichero .js y tecleándola en el navegador podremos descargar el fichero .js y ver todo el código de JavaScript. En otras palabras, nada de lo que tu navegador descargue para mostrar la página web podrá estar oculto de la vista de cualquier programador.

Algunos navegadores no disponen de soporte completo de JavaScript, otros navegadores permiten bloquearlo parcialmente e incluso algunos usuarios bloquean completamente el uso de JavaScript porque creen que así navegan de forma más segura.

En estos casos, es habitual que, si la página web requiere JavaScript para su correcto funcionamiento, se incluya un mensaje de aviso al usuario indicándole que debería activar JavaScript para disfrutar completamente de la página.

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

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

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

2.2. Entrada y Salida

2.2.1. Entrada de datos

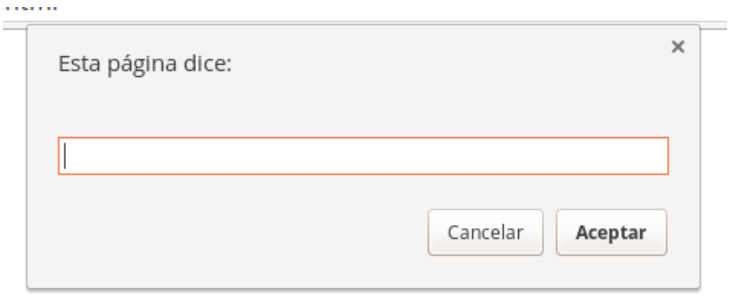
Para la entrada de datos por teclado tenemos la **función prompt**. Cada vez que necesitamos ingresar un dato con esta función aparece una ventana donde cargamos el valor.

El método `prompt()` sirve para que el usuario ingrese todo tipo de datos: cadenas de texto (o *strings*), números enteros, números decimales, etc. Este método se puede usar de tres maneras distintas (y las tres maneras dependen de la cantidad de parámetros que utilizaremos).

La primera forma de usar **`prompt()`** es sin parámetros.

```
var edad = prompt();
```

Y su resultado en el navegador sería lo siguiente:

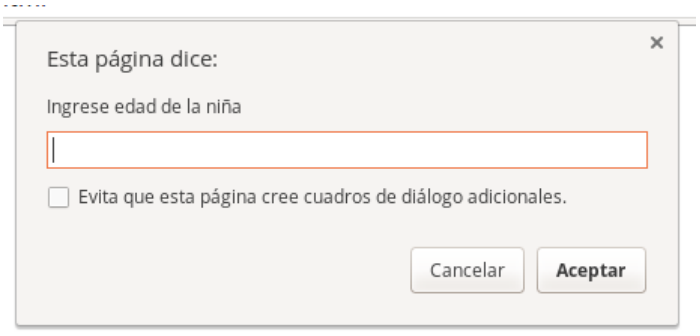


Entonces, lo que hace `prompt()` es mostrarnos una ventana emergente que nos abre un espacio donde podemos ingresar información. Y como anteriormente habíamos guardado el `prompt()` dentro de la variable 'edad', lo que ingresemos dentro del `prompt()` quedará almacenado en esa variable.

La segunda es agregando un parámetro dentro de los paréntesis. Esta información que le pasaremos es el mensaje que queremos que se vea arriba del campo a rellenar, es decir, podemos poner una indicación o pregunta, para que el usuario complete esa información.

```
var edad = prompt('Ingrese edad de la niña');
```

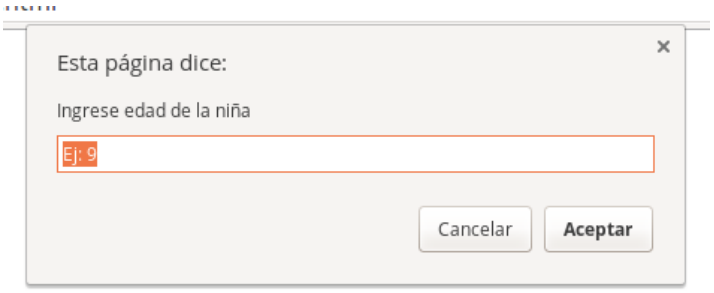
Habiendo ingresado el parámetro, la ventana ahora se verá de la siguiente forma:



Por último, para terminar con el método `prompt()`, si le ingresamos dos parámetros, el primero lo toma como **el contexto de la información a completar** y el segundo como un **valor por defecto**.

Esto se vería así:

```
var edad = prompt('Ingrese edad de la niña', 'Ej: 9');
```



2.2.2. Salida de datos

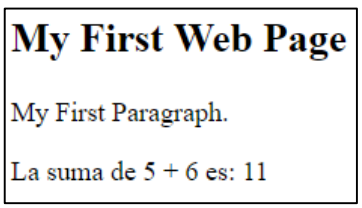
JavaScript puede "mostrar" datos de diferentes maneras:

- Escribir en un elemento HTML, usando **innerHTML**

Para acceder a un elemento HTML, JavaScript puede usar el método **`document.getElementById(id).`** El atributo **id** define el elemento HTML. La propiedad **`innerHTML`** define el contenido HTML

```
<!DOCTYPE html>
<html>
<body>

<h2>My First Web Page</h2>
<p>My First Paragraph.</p>

<p id="solucion"></p>

<script>
var resultado = 5 + 6;
document.getElementById("solucion").innerHTML = "La suma de 5 + 6 es: " +resultado;
</script>

</body>
</html>
```

Y el resultado es:



- Escribiendo en la salida HTML usando **document.write()**

El método `document.write()` solo debe usarse para pruebas, ya que después de cargar un documento HTML, eliminará todo el HTML existente.

```
<!DOCTYPE html>
<html>
<body>

<h2>My First Web Page</h2>
<p>My first paragraph.</p>

<button type="button" onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

Y el resultado es:

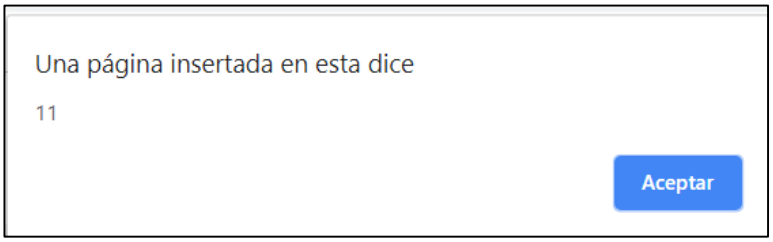
Al pulsar “Try it”, el resultado ahora es: 11

- Escribir en un cuadro de alerta, usando **window.alert ()**

Utiliza un cuadro de alerta para mostrar los datos.

```
<!DOCTYPE html>
<html>
<body>

<h2>My First Web Page</h2>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>

</body>
</html>
```

Y muestra una ventana de alerta:



- Escribiendo en la consola del navegador, utilizando **console.log ()**

Para propósitos de depuración.

2.3. Espacios en blanco

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

```
var that = this;
```

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

2.4. Comentarios

Los comentarios son sentencias que el intérprete de JavaScript ignora. Sin embargo, estas sentencias permiten a los desarrolladores dejar notas sobre cómo funcionan las cosas en sus scripts.

Los comentarios ocupan espacio dentro del código de JavaScript, por lo que cuando alguien se descargue el código necesitará más o menos tiempo, dependiendo del tamaño de vuestro fichero. Aunque esto pueda ser un problema, es muy recomendable el que documentes tu código lo máximo posible, ya que esto te proporcionará muchas más ventajas que inconvenientes.

JavaScript permite dos estilos de comentarios. Un estilo consiste en dos barras inclinadas hacia la derecha (sin espacios entre ellas), y es muy útil para comentar una línea sencilla. JavaScript ignorará cualquier carácter a la derecha de esas barras inclinadas en la misma línea, incluso si aparecen en el medio de una línea.

Ejemplos de comentarios de una única línea:

```
// Este es un comentario de una línea
var nombre="Marta" // Otro comentario sobre esta línea
// Podemos dejar, por ejemplo
//
// una línea en medio en blanco
```

Para comentarios más largos, por ejemplo, de una sección del documento, podemos emplear en lugar de las dos barras inclinadas el /* para comenzar la sección de comentarios, y */ para cerrar la sección de comentarios.

Por ejemplo:

```
/* Ésta es una sección de comentarios
en el código de JavaScript */
```

O también:

```
/* ---------------------------------
función imprimir()
Imprime el listado de alumnos en orden alfabético
-----------------------------------*/
function imprimir()
{
   // Líneas de código JavaScript
}
```

2.5. Variables

La forma más conveniente de trabajar con datos en un script, es asignando primeramente los datos a una variable. Es incluso más fácil pensar que una variable es un cajón que almacena información. El tiempo que dicha información permanecerá almacenada, dependerá de muchos factores. En el momento que el navegador limpia la ventana o marco, cualquier variable conocida será eliminada.

Dispones de dos maneras de crear variables en JavaScript: una forma es usar la palabra reservada var seguida del nombre de la variable. Por ejemplo, para declarar una variable edad, el código de JavaScript será:

```
var edad;
```

Otra forma consiste en crear la variable, y asignarle un valor directamente durante la creación:

```
var edad = 38;
```

o bien, podríamos hacer:

```
var edad;
edad = 38; // Ya que no estamos obligados a declarar la variable
var altura, peso, edad; // Para declarar más de una variable.
```

La palabra reservada **var** se usa para la declaración o inicialización de la variable en el documento.

Una variable de JavaScript podrá almacenar diferentes tipos de valores, y una de las ventajas que tenemos con JavaScript es que no tendremos que decirle de qué tipo es una variable u otra.

A la hora de dar nombres a las variables, tendremos que poner nombres que realmente describan el contenido de la variable. No podremos usar palabras reservadas, ni símbolos de puntuación en el medio de la variable, ni la variable podrá contener espacios en blanco. Los nombres de las variables han de construirse con caracteres alfanuméricos y el carácter subrayado (_). No podremos utilizar caracteres raros como el signo +, un espacio, % , $, etc. en los nombres de variables, y estos nombres no podrán comenzar con un número.

Si queremos nombrar variables con dos palabras, tendremos que separarlas con el símbolo "_" o bien diferenciando las palabras con una mayúscula, por ejemplo:

```
var mi_peso;
var miPeso;// Esta opción es más recomendable
```

Deberemos tener cuidado también en no utilizar nombres reservados como variables. Por ejemplo, no podremos llamar a nuestra variable con el nombre de return o for.

2.6. Tipos de datos

Las variables en JavaScript podrán contener cualquier tipo de dato. A continuación, se muestran los tipos de datos soportados en JavaScript:

Tabla Tipos soportados por JavaScript

Tipo	Ejemplo	Descripción
cadena	“Hola Mundo”	Una serie de caracteres dentro de comillas dobles.
número	9.45	Un número sin comillas dobles.
boolean	True	Un valor verdadero o falso
null	Null	Desprovisto de contenido.
object		Es un objeto software que se define por sus propiedades y métodos.
function		La definición de una función

Para crear variables de **tipo cadena**, simplemente tenemos que poner el texto entre comillas. Si tenemos una cadena que sólo contiene caracteres numéricos, para JavaScript eso será una cadena, independientemente del contenido que tenga. Dentro de las cadenas podemos usar caracteres de escape (como saltos de línea con \n, comillas dobles \”, tabuladores \t, etc.). Cuando sumamos cadenas en JavaScript lo que hacemos es concatenar esas cadenas (poner una a continuación de la otra).

Los **tipos de datos booleano**, son un true o un false. Y se usan mucho para tomar decisiones. Los únicos valores que podemos poner en las variables booleanas son true y false. Se muestra un ejemplo de uso de una variable booleana dentro de una sentencia if, modificando la comparación usando == y ===.

Existen otros tipos de datos especiales que son de **tipo Objeto**, por ejemplo, los arrays. Un array es una variable con un montón de casillas a las que se accede a través de un índice.

El programa final quedará así:

```
<script>
  // tipo de datos numéricos
  // pueden ser enteros o números con decimales
  var miEntero = 33;
  var miDecimales = 2.5;
  var comaFlotante = 2344.983338;
  var numeral = 0.573;
  // pueden tener notación científica
  var numCientifico = 2.9e3;
  var otroNumCientifico = 2e-3;
  //alert(otroNumCientifico);
  // podemos escribir números en otras bases
  var numBase10 = 2200;
  var numBase8 = 0234;
  var numBase16 = 0x2A9F;
  //alert(numBase16);
  // tipo de datos cadena de caracteres
  var miCadena = "Hola! esto es una cadena!";
  var otraCadena = "2323232323"; //parece un numérico pero es cadena
  // caracteres de escape en cadenas
  var cadenaConSaltoDeLinea = "linea1\nLínea2\nLínea3";
  var cadenaConComillas = "cadena „con comillas? \"comillas dobles\"";
  var cadenaNum = "11";
  var sumaCadenaConcatenacion = otraCadena + cadenaNum;
  //alert(sumaCadenaConcatenacion);
  // tipos de datos booleano
  var miBooleano = true;
  var falso = false;
  if (niBooleano){
  //alert ("era true");
  }
  else{
   //alert("era false");
  }
  var booleano = (23==="23");
  //alerts(booleano)
  //esos eran los tipos de datos principales
  //pero existen otros tipos especiales que veremos más adelante
  //arrays
  var miArray = (2,5,7);
  //objetos
  var miObjeto = {
  propiedad: 23,
  otracosa: "hola"
  }
  //operador typeof para conocer un tipo
  alert("El tipo de miEntero es: " + typeof(miEntero));
  alert("El tipo de miCadena es: " + typeof(miCadena));
  alert("El tipo de miEntero es: " + typeof(miBooleano));
  alert("El tipo de miEntero es: " + typeof(miArray));
  alert("El tipo de miEntero es: " + typeof(miObjeto));
</script>
```

2.7. Conversión entre tipos de datos

Aunque los tipos de datos en JavaScript son muy sencillos, a veces se podrá encontrar con casos en los que las operaciones no se realizan correctamente, y eso es debido a la conversión de tipos de datos. JavaScript intenta realizar la mejor conversión cuando realiza esas operaciones, pero a veces no es el tipo de conversión que a nos interesaría.

Por ejemplo, cuando intentamos sumar dos números:

```
4 + 5 // resultado = 9
```

Si uno de esos números está en formato de cadena de texto, JavaScript lo que hará es intentar convertir el otro número a una cadena y los concatenará, por ejemplo:

```
4 + "5" // resultado = "45"
```

Otro ejemplo podría ser:

```
4 + 5 + "6" // resultado = "96"
```

Esto puede resultar ilógico, pero sí que tiene su lógica. La expresión se evalúa de izquierda a derecha.

La primera operación funciona correctamente devolviendo el valor de 9 pero al intentar sumarle una cadena de texto "6" JavaScript lo que hace es convertir ese número a una cadena de texto y se lo concatenará al comienzo del "6".

Para convertir cadenas a números dispones de las funciones: **`parseInt()` y `parseFloat()`.**

Por ejemplo:

```
parseInt("34") // resultado = 34
parseInt("89.76") // resultado = 89
parseFloat devolverá un entero o un número real según el caso:
parseFloat("34") // resultado = 34
parseFloat("89.76") // resultado = 89.76
4 + 5 + parseInt("6") // resultado = 15
```

Si lo que desea es realizar la conversión de números a cadenas, es mucho más sencillo, ya que simplemente tendrás que concatenar una cadena vacía al principio, y de esta forma el número será convertido a su cadena equivalente:

```
("" + 3400) // resultado = "3400"
("" + 3400).length // resultado = 4
```

En el segundo ejemplo podemos ver la gran potencia de la evaluación de expresiones. Los paréntesis fuerzan la conversión del número a una cadena. Una cadena de texto en JavaScript tiene una propiedad asociada con ella que es la longitud (length), la cuál te devolverá en este caso el número 4, indicando que hay 4 caracteres en esa cadena "3400". La longitud de una cadena es un número, no una cadena.

3. OPERADORES

JavaScript es un lenguaje rico en operadores: símbolos y palabras que realizan operaciones sobre uno o varios valores, para obtener un nuevo valor.

Cualquier valor sobre el cual se realiza una acción (indicada por el operador), se denomina un operando. Una expresión puede contener un operando y un operador (denominado operador unario), como por ejemplo en b++, o bien dos operandos, separados por un operador (denominado operador binario), como por ejemplo en a + b.

Tabla Categorías de los operadores en javaScript

Tipo	Qué realizan
Comparación	Comparan valores de dos operandos, devolviendo el resultado de true o false. == != === !== > >= < <=
Aritméticos	Unen dos operandos produciendo un único valor, resultado de una operación aritmética. * - + / % ++ -- +valor -valor
Asignación	Asigna el valor a la derecha de la expresión a la variable de la izquierda. = += -= *= /= %= etc.
Boolean	Realizan operaciones booleanas aritméticas sobre uno o dos operandos booleanos. && || !
Objeto	Evalúan la herencia y capacidades de un objeto. . [] () delete in instanceof new this
Miscelánea	Operadores con comportamiento especial. , ¿: typeof void

3.1. Operadores de comparación

Tabla. Operadores de comparación

Sintaxis	Nombre	Tipos de	Resultados
==	Igualdad	Todos	Boolean
!=	Distinto	Todos	Boolean
===	Igualdad estricta	Todos	Boolean
!==	Desigualdad estricta	Todos	Boolean
>	Mayor que	Todos	Boolean
>=	Mayor o igual que	Todos	Boolean
<=	Menor o igual que	Todos	Boolean

Por ejemplo:

```
30 == 30 // true
30 == 30.0 // true
5 != 8 // true
9 > 13 // false
7.29 <= 7.28 // false
```

También podríamos comparar cadenas a este nivel:

```
"Marta" == "Marta" // true
"Marta" == "marta" // false
"Marta" > "marta" // false
"Mark" < "Marta" // true
```

Para poder comparar cadenas, JavaScript convierte cada carácter de la cadena de un string, en su correspondiente valor ASCII. Cada letra, comenzando con la primera del operando de la izquierda, se compara con su correspondiente letra en el operando de la derecha. Los valores ASCII de las letras mayúsculas, son más pequeños que sus correspondientes en minúscula, por esa razón "Marta" no es mayor que "marta". En JavaScript hay que tener muy en cuenta la sensibilidad a mayúsculas y minúsculas.

Si por ejemplo comparamos un número con su cadena correspondiente:

```
"123" == 123 // true
```

JavaScript cuando realiza esta comparación, convierte la cadena en su número correspondiente y luego realiza la comparación. También dispones de otra opción, que consiste en convertir mediante las funciones `parseInt() o parseFloat()` el operando correspondiente:

```
parseInt("123") == 123 // true
```

Los operadores === y !== comparan tanto el dato como el tipo de dato. El operador === sólo devolverá true, cuando los dos operandos son del mismo tipo de datos (por ejemplo, ambos son números) y tienen el mismo valor.

3.2. Operadores aritméticos

Tabla. Operadores de aritméticos

Sintaxis	Nombre	Tipos de Operando	Resultados
+	Más.	integer, `float`, `string`.	integer, `float`, `string`.
-	Menos.	integer, `float`.	integer, `float`.
*	Multiplicación.	integer, `float`.	integer, `float`.
/	División.	integer, `float`.	integer, `float`.
%	Módulo.	integer, `float`.	integer, `float`.
++	Incremento.	integer, `float`.	integer, `float`.
--	Decremento.	integer, `float`.	integer, `float`.
+valor	Positivo.	integer, `float`, `string`.	integer, `float`.
-valor	Negativo.	integer, `float`, `string`.	integer, `float`.

Algunos ejemplos:

```
var a = 10; // Inicializamos a al valor 10
var z = 0; // Inicializamos z al valor 0
z = a; // a es igual a 10, por lo tanto, z es igual a 10.
z = ++a; // el valor de a se incrementa justo antes de ser asignado a
         //z, por lo que a es 11 y z valdrá 11.
z = a++; // se asigna el valor de a (11) a z y luego se incrementa el
         //valor de a (pasa a ser 12).
z = a++; // a vale 12 antes de la asignación, por lo que z es igual a
         //12; una vez hecha la asignación a valdrá 13.
```

Otros ejemplos:

```
var x = 2;
var y = 8;
var z = -x; // z es igual a -2, pero x sigue siendo igual a 2.
z = - (x + y); // z es igual a -10, x es igual a 2 e y es igual a 8.
z = -x + y; // z igual a 6, pero x es igual a 2 e y igual a 8.
```

3.3. Operadores de asignación

Un operador de asignación asigna un valor a su operando izquierdo basándose en el valor de su operando derecho. El operador básico de asignación es el igual (=), el cual asigna el valor de su operador derecho a su operador izquierdo. Esto es, x=y asigna el valor de y a x.

Tabla. Operadores de asignación

Sintaxis	Nombre	Ejemplo	Significado operandos
=	Asignación	x=y	X=y
+=	Sumar un valor	x+=y	x=x+y
-=	Substraer un valor	x-=y	x=x-y
*=	Multiplicar un valor	x*=y	x=x*y
%=	Módulo de un valor	x%=y	x=x%y
<<=	Desplazar bits a la izquierda	x<<=y	x=x<<y
>>=	Desplazar bits a la derecha	x>=y	x=x>y
&=	Operación AND bit a bit	x&=y	x=x&y
!=	Operación OR bit a bit	x|=y	x=x|y
^=	Operación XOR bit a bit	x^=y	x=x^y
[]=	Desectructuración de asignaciones	[a,b]=[c,d]	a=c,b=d

3.4. Operadores booleanos

Debido a que parte de la programación tiene un gran componente de lógica, es por ello, que los operadores booleanos juegan un gran papel.

Los operadores booleanos te van a permitir evaluar expresiones, devolviendo como resultado true (verdadero) o false (falso).

Tabla. Operadores de asignación

Sintaxis	Nombre	Operandos	Resultados
&&	AND.	Boolean.	Boolean.
||	OR.	Boolean.	Boolean.
!	Not.	Boolean.	Boolean.

Ejemplos:

```
!true // resultado = false
!(10 > 5) // resultado = false
!(10 < 5) // resultado = true
!("gato" == "pato") // resultado = true
```

```
5 > 1 && 50 > 10 // resultado = true
5 > 1 && 50 < 10 // resultado = false
5 < 1 && 50 > 10 // resultado = false
5 < 1 && 50 < 10 // resultado = false
```

Tabla de verdad del operador AND

Operando Izquierdo	Operador AND	Operando Derecho	Resultado
True	&&	True	True
True	&&	False	False
False	&&	True	False
False	&&	False	False

Tabla de verdad del operador OR

Operando Izquierdo	Operador OR	Operando Derecho	Resultado
True	||	True	True
True	||	False	True
False	||	True	True
False	||	False	False

Ejemplos:

```
5 > 1 || 50 > 10 // resultado = true
5 > 1 || 50 < 10 // resultado = true
5 < 1 || 50 > 10 // resultado = true
5 < 1 || 50 < 10 // resultado = false
```

3.5. Operadores misceláneos

- **El operador coma ,**

Este operador, indica una serie de expresiones que van a ser evaluadas en secuencia, de izquierda a derecha. La mayor parte de las veces, este operador se usa para combinar múltiples declaraciones e inicializaciones de variables en una única línea.
Por ejemplo:

```
var nombre, direccion, apellidos, edad;
```

Otra situación en la que podemos usar este operador coma, es dentro de la expresión loop. En el siguiente ejemplo inicializamos dos variables de tipo contador, y las incrementamos en diferentes porcentajes. Cuando comienza el bucle, ambas variables se inicializan a 0 y a cada paso del bucle una de ellas se incrementa en 1, mientras que la otra se incrementa en 10.

```
for (var i=0, j=0; i < 125; i++, j+10)
{
      // más instrucciones aquí dentro
}
```

- **? : (operador condicional)**

Este operador condicional es la forma reducida de la expresión if …. else. La sintaxis formal para este operador condicional es:

```
condicion ? expresión si se cumple la condición: expresión si no se
cumple;
```

Si usamos esta expresión con un operador de asignación:

```
var = condicion ? expresión si se cumple la condición: expresión si no
se cumple;
```

Ejemplo:

```
var a,b;
a = 3; b = 5;
var h = a > b ? a : b; // a h se le asignará el valor 5;
```

- **typeof (devuelve el tipo de valor de una variable o expresión).**

Este operador unario se usa para identificar cuando una variable o expresión es de alguno de los siguientes tipos: number, string, boolean, object, function o undefined.

Ejemplo:

```
if (typeof miVariable == "number")
{
      miVariable = parseInt(miVariable);
}
```

4. ESTRUCTURAS DE CONTROL

4.1. Estructura if

La decisión más simple que podemos tomar en un programa, es la de seguir una rama determinada si una determinada condición es true.

Sintaxis:

```
if (condición) // entre paréntesis irá la condición que se evaluará a
true o false.
{
       // instrucciones a ejecutar si se cumple la condición
}
```

Ejemplo:

```
if (miEdad >30)
{
       alert("Ya eres una persona adulta");
}
```

4.2. Estructura if…else

En este tipo de construcción, podemos gestionar que haremos cuando se cumpla y cuando no se cumpla una determinada condición.

Sintaxis:

```
if (condición){
     // entre paréntesis irá la condición que se evaluará a true o false.
     // instrucciones a ejecutar si se cumple la condición
}
else
{
     // instrucciones a ejecutar si no se cumple la condición
}
```

Ejemplo:

```
if (miEdad >30){
       alert("Ya eres una persona adulta.");
}
else{
       alert("Eres una persona joven.");
}
```

4.3. Estructura switch

En este tipo de construcción, se evalúa la expresión, y en función del valor que adopte, se ejecuta algunas de las alternativas prevista. Si la expresión no equivale a los patrones establecidos en los case, se ejecuta las sentencias establecidas por defecto. El break se utiliza para que una vez que no se intente entrar en otras alternativas.

Sintaxis:

```
switch(Expresión)
{
       case Constante1:
            Bloque de sentencias1;
            break;
       case Constante2:
            Bloque de sentencias2;
            break;
            ...
       case ConstanteN:
            Bloque de sentenciasN;
            break;
       default:
       Bloque de sentencias por defecto;
}
```

4.4. Estructura for

Este tipo de bucle te deja repetir un bloque de instrucciones un número limitado de veces.

Sintaxis:

```
for (expresión inicial; condición; incremento)
{
   // Instrucciones a ejecutar dentro del bucle.
}
```

Ejemplo:

```
for (var i=1; i<=20; i++)
{
   // Instrucciones que se ejecutarán 20 veces.
}
```

4.5. Estructura while

Este tipo de bucles se utilizan cuando queremos repetir la ejecución de unas sentencias un número indefinido de veces, siempre que se cumpla una condición. Es más sencillo de comprender que el bucle FOR, ya que no incorpora en la misma línea la inicialización de las variables, su condición para seguir ejecutándose y su actualización. Sólo se indica, como veremos a continuación, la condición que se tiene que cumplir para que se realice una iteración o repetición.

Sintaxis:

```
while (condición)
{
       // Instrucciones a ejecutar dentro del bucle.
}
```

Ejemplo:

```
var i=0;
while (i <=10){
    // Instrucciones a ejecutar dentro del bucle hasta que i sea mayor
    // que 10 y no se cumpla
    // la condición.
    i++;
}
```

5. FUNCIONES

Una función es la definición de un conjunto de acciones pre-programadas. Las funciones se llaman a través de eventos o bien mediante comandos desde un script.

Siempre que sea posible, se deben diseñar funciones para poder reutilizarlas en otras aplicaciones, de esta forma, las funciones se convertirán en pequeños bloques constructivos que te permitirán ir más rápido en el desarrollo de nuevos programas.

La **sintaxis** formal de una función es la siguiente:

```
function nombreFunción ( [parámetro1]....[parámetroN] )
{
// instrucciones
}
```

Si nuestra función va a devolver algún valor emplearemos la palabra reservada return, para hacerlo.

Ejemplo:

```
function nombreFunción ( [parámetro1]....[parámetroN] )
{
    // instrucciones
    return valor;
}
```

Los nombres que se pueden asignar a una función, tendrán las mismas restricciones que tienen los elementos HTML y las variables en JavaScript. Se deben asignar un nombre que realmente la identifique, o que indique qué tipo de acción realiza. Se puede usar palabras compuestas como chequearMail o calcularFecha, las funciones suelen llevar un verbo, puesto que las funciones son elementos que realizan acciones.

Una recomendación, es que las funciones sean muy específicas, es decir que no realicen tareas adicionales a las inicialmente propuestas en esa función.

Para realizar una llamada a una función lo podemos hacer con:

```
nombreFuncion();// Esta llamada ejecutaría las instrucciones
                //programadas dentro de la función.
```

Otro ejemplo de uso de una función en una asignación:

```
variable=nombreFuncion(); // En este caso la función devolvería
                          //un valor que se asigna a la variable
```

Las funciones en JavaScript también son objetos, y como tal tienen métodos y propiedades. Un método, aplicable a cualquier función puede ser `toString()`, el cual nos devolverá el código fuente de esa función.

5.1. Parámetros

Cuando se realiza una llamada a una función, muchas veces es necesario pasar parámetros (también conocidos como argumentos). Para pasar parámetros a una función, tendremos que escribir dichos parámetros entre paréntesis y separados por comas. Veamos el siguiente ejemplo:

```
function saludar(a,b)
{
      alert("Hola " + a + " y "+ b +".");
}
```

Si llamamos a esa función desde el código:

```
saludar("Martin","Silvia");
//Mostraría una alerta con el texto: Hola Martin y Silvia.
```

Otro ejemplo de función que devuelve un valor:

```
function devolverMayor(a,b) {
       if (a > b) then
            return a;
       else
            return b;
}
```

Ejemplo de utilización de la función anterior:

```
document.write ("El número mayor entre 35 y 21 es el: " +
devolverMayor(35,21) + ".");
```

5.2. Ámbito de las variables

Las variables que se definen fuera de las funciones se llaman **variables globales**. Las variables que se definen dentro de las funciones, con la palabra reservada var, se llaman **variables locales**.

Ejemplo de variables locales y globales:

```
/* Uso de variables locales y globales no muy recomendable,
ya que estamos empleando el mismonombre de variable en global y en
local.*/
var chica = "Aurora"; // variable global
var perros = "Lacky, Samba y Ronda"; // variable global
function demo()
{
   // Definimos una variable local (es obligatorio para
   // las variables locales usar var)
   // Esta variable local tendrá el mismo nombre que otra variable
   // global pero con distinto contenido.
   // Si no usáramos var estaríamos modificando la variable global chica.
   var chica = "Raquel"; // variable local
   document.write( "<br/>"+ perros +" no pertenecen a "+ chica +".");
}
// Llamamos a la función para que use las variables locales.
demo();// Utilizamos las variables globales definidas al comienzo.
document.write("<br/>" + perros + " pertenecen a " + chica + ".");
```

6. OBJETOS

6.1. Propiedades y métodos

Un objeto es una estructura que tiene propiedades y métodos.

Las propiedades son como los adjetivos en el lenguaje, es decir que expresan una cualidad del objeto, así por ejemplo las instrucciones dadas en lenguaje CSS se consideran propiedades, ya que modifican el aspecto de la página. Los atributos de las etiquetas HTML son considerados también propiedades.

Los métodos son funcionalidades, es decir funciones o modos de operar asociados a un objeto. Se comportan igual que una función. Un método es algo que se puede hacer con el objeto. Es por eso que llevan siempre un paréntesis detrás, en el que se le pueden pasar varios valores para que opere con ellos (los parámetros o argumentos).

```
var person = {
  firstName:"John",
  lastName:"Doe",
  age:50,
  eyeColor:"blue"
};
```

Los pares **`nombre: valores`** en objetos JavaScript se llaman **propiedades**:

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Se puede acceder a las propiedades de los objetos de dos maneras:

- `objectName.propertyName`

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

<p>There are two different ways to access an object property.</p>

<p>You can use person.property or person["property"].</p>

<p id="demo"></p>

<script>
// Create an object:
var person = {
  firstName: "John",
  lastName : "Doe",
  id     :  5566
};
// Display some data from the object:
document.getElementById("demo").innerHTML =
person.firstName + " " + person.lastName;
</script>

</body>
</html>
```

- `objectName["propertyName"]`

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

<p>There are two different ways to access an object property.</p>

<p>You can use person.property or person["property"].</p>

<p id="demo"></p>

<script>
// Create an object:
var person = {
  firstName: "John",
  lastName : "Doe",
  id     :  5566
};
// Display some data from the object:
document.getElementById("demo").innerHTML =
person["firstName"] + " " + person["lastName"];
</script>

</body>
</html>
```

Los **métodos** se almacenan en propiedades como **definiciones de funciones**.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

En una definición de función, **`this`** se refiere al "propietario" de la función.

En el ejemplo anterior, **`this`** es el **objeto de persona** que "posee" la función **fullName**. En otras palabras, **this.firstName** significa la propiedad **firstName** de **este objeto** .

Se accede a un método de objeto con la siguiente sintaxis:

- `objectName.methodName()`

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

<p>An object method is a function definition, stored as a property value.</p>

<p id="demo"></p>

<script>
// Create an object:
var person = {
  firstName: "John",
  lastName : "Doe",
  id     : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>
</body>
</html>
```

Si accede a un método **sin** los paréntesis (), devolverá la **definición de** la **función**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

<p>If you access an object method without (), it will return the function definition:
</p>

<p id="demo"></p>

<script>
// Create an object:
var person = {
  firstName: "John",
  lastName : "Doe",
  id     : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};

// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName;
</script>

</body>
</html>
```

6.2. Creación de objetos

Con JavaScript, puedes definir y crear tus propios objetos. Hay diferentes formas de crear nuevos objetos:

- **Utilizando un objeto literal:** Un objeto literal es una lista de pares nombre: valor (como age: 50) dentro de llaves {}.

```
<!DOCTYPE html>
<html>
<body>

<p>Creating a JavaScript Object.</p>

<p id="demo"></p>

<script>
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```

- **Con la palabra clave new**

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```

- **Definiendo un constructor de objetos y creando objetos del tipo construido.**

6.3. Objetos predefinidos por JavaScript

En JavaScript, casi "todo" es un objeto.

- Los booleanos pueden ser objetos (si se definen con **new**)
- Los números pueden ser objetos (si se definen con **new**)
- Las cadenas pueden ser objetos (si se definen con **new**)
- Las fechas son siempre objetos.
- Las matemáticas son siempre objetos.
- Las expresiones regulares son siempre objetos.
- Los arrays son siempre objetos.
- Las funciones son siempre objetos.
- Los objetos son siempre objetos.

Todos los valores de JavaScript, excepto los primitivos (string, number, boolean, null, undefined), son objetos.

6.4. Operadores de objeto

El siguiente grupo de operadores se relaciona directamente con objetos y tipos de datos. La mayor parte de ellos fueron implementados a partir de las primeras versiones de JavaScript, por lo que puede haber algún tipo de incompatibilidad con navegadores antiguos.

- **. (punto)**. El operador punto, indica que el objeto a su izquierda tiene o contiene el recurso a su derecha, como por ejemplo: objeto.propiedad y objeto.método().

Ejemplo con un objeto nativo de JavaScript:

```
var s = new String('rafa');
var longitud = s.length;
var pos = s.indexOf("fa"); // resultado: pos = 2
```

- **[ ]** (corchetes para enumerar miembros de un objeto).

Por ejemplo, cuando creamos un array:

```
var a = ["Santiago","Coruña", "Lugo"];
```

Enumerar un elemento de un array:

```
a[1] = "Coruña";
```

Enumerar una propiedad de un objeto:

```
a["color"]= "azul";
```

- **delete** (para eliminar un elemento de una colección).

Por ejemplo, si consideramos:

```
var oceanos = new Array("Atlantico", "Pacifico", "Indico","Artico");
```

Podríamos hacer:

```
delete oceanos[2];
```

Esto eliminaría el tercer elemento del array ("Indico"), pero la longitud del array no cambiaría. Si intentamos referenciar esa posición oceanos[2] obtendríamos undefined.

- **in** (para inspeccionar métodos o propiedades de un objeto).

El operando a la izquierda del operador, es una cadena referente a la propiedad o método (simplemente el nombre del método sin paréntesis); el operando a la derecha del operador, es el objeto que estamos inspeccionando. Si el objeto conoce la propiedad o método, la expresión devolverá true.

Ejemplo:

```
"write" in document
```

o también

```
"defaultView" in document
```

- **instanceof** (para comprobar si un objeto es una instancia de un objeto nativo de JavaScript).

Ejemplo:

```
a = new Array(1,2,3);
a instanceof Array; // devolverá true.
```

- **new** (para acceder a los constructores de objetos incorporados en el núcleo de JavaScript).

Ejemplo:

```
var hoy = new Date();
// creará el objeto hoy de tipo Date() empleando el constructor por
// defecto de dicho objeto.
```

- **this** (para hacer referencia al propio objeto en el que estamos localizados).

Ejemplo:

```
nombre.onchange = validateInput;
function validateInput(evt)
{
   var valorDeInput = this.value;
   // Este this hace referencia al objeto nombre que estamos validando.
}
```

7. OBJETO ARRAYS

En programación, un array se define como una colección ordenada de datos. Un array es como si fuera una tabla que contiene datos, o también como si fuera una hoja de cálculo.

Por ejemplo, si tenemos las siguientes variables:

```
var coche1="Seat";
var coche2="BMW";
var coche3="Audi";
var coche4="Toyota";
```

Este ejemplo sería un buen candidato a convertirlo en un array, ya que permitiría introducir más marcas de coche, sin que tengas que crear nuevas variables para ello.

7.1. Creación de un array

Para crear un objeto array, podremos hacerlos de dos formas:

a) Definiendo el array de forma literal.

```
var coches = ["Saab", "BMW", "Audi", "Toyota"];
```

b) Usando el constructor `new Array()`.

```
var coches = new Array("Saab", "BMW", "Audi", "Toyota");
```

Por simplicidad, legibilidad y velocidad de ejecución, se utiliza la opción a).

Para establecer un número de posiciones para un array, sería:

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

7.2. Introducir datos en un array.

Introducir datos en un array, es tan simple como crear una serie de sentencias de asignación, una por cada elemento del array. Ejemplo de un array que contiene el nombre de los planetas del sistemasolar:

```
sistemaSolar = new Array();
sistemaSolar[0] = "Mercurio";
sistemaSolar[1] = "Venus";
sistemaSolar[2] = "Tierra";
sistemaSolar[3] = "Marte";
sistemaSolar[4] = "Jupiter";
sistemaSolar[5] = "Saturno";
sistemaSolar[6] = "Urano";
sistemaSolar[7] = "Neptuno";
```

```
sistemaSolar=[];
sistemaSolar[0] = "Mercurio";
sistemaSolar[1] = "Venus";
sistemaSolar[2] = "Tierra";
sistemaSolar[3] = "Marte";
sistemaSolar[4] = "Jupiter";
sistemaSolar[5] = "Saturno";
sistemaSolar[6] = "Urano";
sistemaSolar[7] = "Neptuno";
```

Esta forma es un poco tediosa a la hora de escribir el código, pero una vez que las posiciones del array están cubiertas con los datos, acceder a esa información nos resultará muy fácil:

```
unPlaneta=sistemaSolar[2];//almacenará en un Planeta la cadena
"Tierra".
```

Otra forma de crear el array puede ser mediante el constructor. En lugar de escribir cada sentencia de asignación para cada elemento, lo podemos hacer creando lo que se denomina un "array denso", aportando al constructor array(), los datos a cubrir separados por comas:

```
sistemaSolar = new array ("Mercurio", "Venus", "Tierra", "Marte",
"Jupiter", "Saturno","Urano", "Neptuno");
```

El término de "**array denso**" quiere decir que los datos están empaquetados dentro del array, sin espacios y comenzando en la posición 0. Otra forma permitida a partir de la versión de JavaScript 1.2+, sería aquella en la que no se emplea el constructor y se definen los arrays de forma literal:

```
sistemaSolar = ["Mercurio","Venus","Tierra","Marte","Jupiter","Saturno",
"Urano", "Neptuno"];
```

Y para terminar vamos a ver otra forma de creación de un **array mixto** (o también denominado objeto literal), en el que las posiciones son referenciadas con índices de tipo texto o números, mezclándolos de forma aleatoria.

Por ejemplo:

```
var datos={ "numero": 42, "mes" : "Junio", "hola":"mundo", 69:"96" };

document.write("<br/>"+datos["numero"]+ " -- " + datos["mes"]+ " -- "
+ datos["hola"]+ " -- "+ datos[69]+ "<br/>");
```

7.3. Recorrido de un array.

Existen múltiples formas de recorrer un array para mostrar sus datos. Veamos algunos ejemplos con el array del sistema Solar:

```
var sistemaSolar = new Array();
sistemaSolar[0] = "Mercurio";
sistemaSolar[1] = "Venus";
sistemaSolar[2] = "Tierra";
sistemaSolar[3] = "Marte";
sistemaSolar[4] = "Jupiter";
sistemaSolar[5] = "Saturno";
sistemaSolar[6] = "Urano";
sistemaSolar[7] = "Neptuno";
```

Empleando un bucle for, por ejemplo:

```
for (i=0;i<sistemaSolar.length;i++)
document.write(sistemaSolar[i] + "<br/>");
```

Empleando un bucle while, por ejemplo:

```
var i=0;
while (i<sistemaSolar.length)
{
  document.write(sistemaSolar[i] + "<br/>");
  i++;
}
```

7.4. Borrado de elementos en un array

Para borrar cualquier dato almacenado en un elemento del array, se hace ajustando su valor a **null** o a una cadena vacía "".

Hasta que apareció el operador `delete` en las versiones más modernas de navegadores, no se podía eliminar completamente una posición del array.

Al borrar un elemento del array, se eliminará su índice en la lista de índices del array, pero no se reducirá la longitud del array. Por ejemplo, en las siguientes instrucciones:

```
elarray.length; // resultado: 8
delete elarray[5];
elarray.length; // resultado: 8
elarray[5]; // resultado: undefined
```

El proceso de borrar una entrada del array no libera la memoria ocupada por esos datos necesariamente.

Si consideramos el siguiente array denso:

```
var oceanos = new array("Atlantico","Pacifico","Artico","Indico");
```

Esta clase de array asigna automáticamente índices numéricos a sus entradas, para poder acceder posteriormente a los datos, como por ejemplo con un bucle:

```
for (var i=0; i<oceanos.length; i++)
{
  if (oceanos[i] == "Atlantico")
  {
    // instrucciones a realizar..
  }
}
```

Si ejecutamos la instrucción:

```
delete oceanos[2];
```

Se producirán los siguientes cambios: en primer lugar, el tercer elemento del array ("Artico"), será eliminado del mismo, pero la longitud del array seguirá siendo la misma, y el array quedará tal y como:

```
oceanos[0] = "Atlantico";
oceanos[1] = "Pacifico";
oceanos[3] = "Indico";
```

7.5. Propiedades y métodos

Vamos a ver a continuación, las propiedades y métodos que podemos usar con cualquier array que utilicemos en JavaScript.

Tabla. Propiedades del objeto array

Propiedad	Descripción
constructor	Devuelve la función que creó el prototipo del objeto array.
length	Ajusta o devuelve el número de elementos de un array.
prototype	Te permite añadir propiedades y métodos a un objeto.

Tabla. Métodos del objeto array

Métodos	Descripción
concat()	Une dos o más arrays, y devuelve una copia de los arrays unidos.
join()	Une todos los elementos de un array en una cadena de texto.
pop()	Elimina el último elemento de un array y devuelve la nueva longitud.
reverse()	Invierte el orden de los elementos en un array.
shift()	Elimina el primer elemento de un array, y devuelve ese elemento.
slice()	Selecciona una parte de un array y devuelve el nuevo array.
sort()	Ordena los elementos de un array.
splice()	Añada/elimina elementos de un array.
toString()	Convierte un array a una cadena y devuelve el resultado.
unshift()	Añade nuevos elementos al comienzo de un array, y devuelve la nueva longitud.
valueOf()	Devuelve un valor primitivo de un array.

Ejemplo de uso algunos métodos del objeto array:

- **Método reverse():**

```
var frutas = ["Plátano", "Naranja", "Manzana", "Melocotón"];
document.write(frutas.reverse());
// Imprimirá: Melocotón,Manzana,Naranja,Plátano
```

- **Método slice():**

```
var frutas = ["Plátano", "Naranja", "Manzana", "Melocotón"];
document.write(frutas.slice(0,1) + "<br />");
// imprimirá: Plátano
document.write(frutas.slice(1) + "<br />");
// imprimirá: Naranja,Manzana,Melocotón
document.write(frutas.slice(-2) + "<br />");
// imprimirá: Manzana, Melocotón
document.write(frutas + "<br />");
// imprimirá: Plátano,Naranja,Manzana,Melocotón
```

7.6. Funciones útiles para arrays

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8. OTROS OBJETOS PREDEFINIDOS

El **Modelo de Objetos del Documento (DOM)**, permite ver el mismo documento de otra manera, describiendo el contenido del documento como un conjunto de objetos, sobre los que un programa de Javascript puede interactuar.

Según el **W3C**, el Modelo de Objetos del Documento es una interfaz de programación de aplicaciones (**API**), para documentos válidos HTML y bien construidos XML. Define la estructura lógica de los documentos, y el modo en el que se acceden y se manipulan.

Definimos como objeto, una entidad con una serie de propiedades que definen su estado, y unos métodos (funciones), que actúan sobre esas propiedades.

La forma de acceder a una propiedad de un objeto es la siguiente:

```
nombreobjeto.propiedad
```

La forma de acceder a un método de un objeto es la siguiente:

```
nombreobjeto.metodo( [parámetros opcionales] )
```

También podemos referenciar a una propiedad de un objeto, por su índice en la creación. Los índices comienzan por 0.

En esta sección vamos a echar una ojeada a objetos que son nativos en JavaScript, esto es, aquello que JavaScript nos da, listos para su utilización en nuestra aplicación.

8.1. Objeto String

Una cadena (string) consta de uno o más caracteres de texto, rodeados de comillas simples o dobles; da igual cuales usemos ya que se considerará una cadena de todas formas, pero en algunos casos resulta más cómodo el uso de unas u otras. Por ejemplo, si queremos meter el siguiente texto dentro de una cadena de JavaScript:

```
<input type="checkbox" name="coche" />Audi A6
```

Podremos emplear las comillas dobles o simples:

```
var cadena = '<input type="checkbox" name="coche" />Audi A6';
var cadena = "<input type='checkbox' name='coche' />Audi A6";
```

Si queremos emplear comillas dobles al principio y fin de la cadena, y que en el contenido aparezcan también comillas dobles, tendríamos que escaparlas con \", por ejemplo:

```
var cadena = "<input type=\"checkbox\" name=\"coche\" />Audi A6";
```

Cuando estamos hablando de cadenas muy largas, podríamos concatenarlas con +=, por ejemplo:

```
var nuevoDocumento = "";
nuevoDocumento += "<!DOCTYPE html>"; nuevoDocumento += "<html>";
nuevoDocumento += "<head>";
nuevoDocumento += '<meta http-equiv="content-type"';
nuevoDocumento += ' content="text/html;charset=utf-8">';
```

Si queremos concatenar el contenido de una variable dentro de una cadena de texto emplearemos el símbolo + :

```
nombreEquipo=prompt("Introduce el nombre de tu equipo favorito:","");
var mensaje= "El " + nombreEquipo + " ha sido el campeón de la Copa!";
alert(mensaje);
```

Caracteres especiales o caracteres de escape.

La forma en la que se crean las cadenas en JavaScript, hace que cuando tengamos que emplear ciertos caracteres especiales en una cadena de texto, tengamos que escaparlos, empleando el símbolo \ seguido del carácter.

Vemos aquí un listado de los caracteres especiales o de escape en JavaScript:

Símbolos	Explicación
\”	Comillas dobles
\’	Comilla simple
\\	Barra inclinada
\b	Retroceso
\t	Tabulador
\n	Nueva línea
\r	Salto de línea
\f	Avance de página

Para crear un objeto String lo podremos hacer de la siguiente forma:

```
var miCadena = new String("texto de la cadena");
```

O también se podría hacer:

```
var miCadena = "texto de la cadena";
```

Es decir, cada vez que tengamos una cadena de texto, en realidad es un objeto String que tiene **propiedades y métodos**:

```
cadena.propiedad;
cadena.metodo( [parámetros] );
```

Ejemplos de uso:

```
var cadena="El parapente es un deporte de riesgo medio";
document.write("La longitud de la cadena es: "+ cadena.length + "<br/>");
document.write(cadena.toLowerCase()+ "<br/>");
document.write(cadena.charAt(3)+ "<br/>");
document.write(cadena.indexOf('pente')+ "<br/>");
document.write(cadena.substring(3,16)+ "<br/>");
```

Tabla Propiedad del objeto String

Propiedad	Descripción
constructor	Devuelve la función que ha creado el prototipo del objeto string
length	Contiene la longitud de la cadena
prototype	Te permite añadir propiedades y métodos a un objeto

Tabla Métodos del objeto String

Método	Descripción
charAt()	Devuelve el carácter especificado por la posición que se indica entre paréntesis.
charCodeAt()	Devuelve el Unicode del carácter especificadopor la posición que se indica entre paréntesis.
concat()	Une una o más cadenas y devuelve el resultado de esa unión
fromCharCode()	Convierte valores Unicode a caracteres
indexOf()	Devuelve la posición de la primera ocurrencia del carácter buscado en la cadena
lastIndexOf()	Devuelve la posición de la última ocurrencia del carácter buscado en la cadena.
fromCharCode()	Convierte valores Unicode a caracteres
match()	Busca una coincidencia entre una expresión regular y una cadena, y devuelve la posición de la coincidencia
replace()	Busca una coincidencia entre una subcadena (o expresión regular) y una cadena, y sustituye a la subcadena encontrada con una nueva subcadena
search()	Busca una coincidencia entre una expression regular y una cadena, y devuelve las coincidencias
slice()	Extrae una parte de una cadena y devuelve una nueva cadena
split()	Divide una cadena dentro de un array de subcadenas
substr()	Extrae los caracteres de una cadena, empezando en la posición de inicio especificado, y el número especificado de caracteres.
substring()	Extrae los caracteres de una cadena, entre dos índices especificados.
toLowerCase()	Convierte una cadena a minúsculas
toUpperCase()	Convierte una cadena a mayúsculas
valueOf()	Devuelve el valor primitive de un objeto String

8.1.1. Funciones útiles para cadenas de texto

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

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

- **+,** se emplea para concatenar varias cadenas de texto

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **`indexOf(caracter)`**, calcula la posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si el carácter se incluye varias veces dentro de la cadena de texto, se devuelve su primera posición empezando a buscar desde la izquierda. Si la cadena no contiene el carácter, la función devuelve el valor -1:

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

Su función análoga es lastIndexOf():

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.2. Objeto Math

El objeto Math no es un constructor (no nos permitirá por lo tanto crear o instanciar nuevos objetos que sean de tipo Math), por lo que, para llamar a sus propiedades y métodos, lo haremos anteponiendo Math a la propiedad o el método. Por ejemplo:

```
var x = Math.PI; // Devuelve el número PI.
var y = Math.sqrt(16); // Devuelve la raíz cuadrada de 16.
```

Ejemplos de uso:

```
document.write(Math.cos(3) + "<br />"); document.write(Math.asin(0) +
"<br />");
document.write(Math.max(0,150,30,20,38) + "<br />");
document.write(Math.pow(7,2) + "<br />");
document.write(Math.round(0.49) + "<br />");
```

8.3. Objeto Number

El objeto Number se usa muy raramente, ya que, para la mayor parte de los casos, JavaScript satisface las necesidades del día a día con los valores numéricos que almacenamos en variables. Pero el objeto Number contiene alguna información y capacidades muy interesantes para programadores más experimentados.

Lo primero, es que el objeto Number contiene propiedades que nos indican el rango de números soportados en el lenguaje. El número más alto es 1.79E + 308; el número más bajo es **2.22E-308**.

Cualquier número mayor que el número más alto, será considerado como infinito positivo, y si es más pequeño que el número más bajo, será considerado infinito negativo.

Los números y sus valores están definidos internamente en JavaScript, como valores de doble precisión y de 64 bits.

El objeto **Number**, es un objeto envoltorio para valores numéricos primitivos. Los objetos **Number** son creados con **new Number().**

Tabla Propiedades del objeto Number

Propiedad	Descripción
constructor	Devuelve la función que creó el objeto Number
MAX_VALUE	Devuelve el número más alto disponible en JavaScript
MIN_VALUE	Devuelve el Devuelve el número más pequeño disponible en JavaScript
NEGATIVE_INFINITY	Representa a infinito negativo (overflow)
POSITIVE _INFINITY	Representa a infinito positivo (se devuelve en caso de overflow)
prototype	Permite añadir nuestras propias propiedades y métodos a un objeto

Tabla Métodos del objeto Number

Método	Descripción
toExponential(x)	Convierte un número a su notación exponencial.
toFixed(x)	Formatea un número con x dígitos decimales después del punto decimal
toPrecision (x)	Formatea un número a la longitud x
toString()	Convierte un objeto Number en una cadena. Si se pone 2 como parámetro se mostrará el número en binario. Si se pone 8 como parámetro se mostrará el número en octal. Si se pone 16 como parámetro se mostrará el número en hexadecimal.
valueOf()	Devuelve el valor primitivo de un objeto Number

Algunos ejemplos de uso:

```
var num = new Number(13.3714);
document.write(num.toPrecision(3)+"<br />");
document.write(num.toFixed(1)+"<br />");
document.write(num.toString(2)+"<br />");
document.write(num.toString(8)+"<br />");
document.write(num.toString(16)+"<br />");
document.write(Number.MIN_VALUE);
document.write(Number.MAX_VALUE);
```

8.3.1. Funciones útiles para números

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

```
var numero1 = 0;
var numero2 = 0;
console.log(numero1/numero2); // se muestra el valor NaN
```

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

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

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

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

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

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

8.4. Objeto Boolean

El objeto Boolean se utiliza para convertir un valor no Booleano, a un valor Booleano (**true** o **false**).

Tabla Propiedades del objeto Boolean

Propiedad	Descripción
constructor	Devuelve la función que creó el objeto Boolean
prototype	Te permite añadir propiedades y métodos a un objeto

Tabla Métodos del objeto Boolean

Método	Descripción
toString()	Convierte un objeto Boolean en una cadena y devuelve el resultado
valueOf()	Devuelve el valor primitivo de un objeto Boolean

Algunos ejemplos de uso:

```
var bool = new Boolean(1);
document.write(bool.toString());
document.write(bool.valueOf());
```

La clase Boolean es una clase nativa de JavaScript que nos permite crear valores booleanos.

Una de sus posibles utilidades es la de conseguir valores boleanos a partir de datos de cualquier otro tipo. No obstante, al igual que ocurría con la clase Number, es muy probable que no la llegues a utilizar nunca.

Dependiendo de lo que reciba el constructor de la clase Boolean el valor del objeto boleano que se crea será verdadero o falso, de la siguiente manera:

1. Se inicializa a false cuando no pasas ningún valor al constructor, o si pasas una cadena vacía, el número 0 o la palabra false sin comillas.
2. Se inicializa a true cuando recibe cualquier valor entrecomillado o cualquier número distinto de 0.

Se puede comprender el funcionamiento de este objeto fácilmente si examinamos unos ejemplos.

```
var b1 = new Boolean()
document.write(b1 + "<br/>")
//muestra false
var b2 = new Boolean("")
document.write(b2 + "<br/>")
//muestra false
var b25 = new Boolean(false)
document.write(b25 + "<br/>")
//muestra false
var b3 = new Boolean(0)
document.write(b3 + "<br/>")
//muestra false
var b35 = new Boolean("0")
document.write(b35 + "<br/>")
//muestra true
var b4 = new Boolean(3)
document.write(b4 + "<br/>")
//muestra true
var b5 = new Boolean("Hola")
document.write(b5 + "<br/>")
//muestra true
```

8.5. Objeto Date

El objeto Date se utiliza para trabajar con fechas y horas. Los objetos **Date** se crean con **`new Date()`.**

Hay 4 formas de instanciar (crear un objeto de tipo Date):

```
var d = new Date();
var d = new Date(milisegundos);
var d = new Date(cadena de Fecha);
var d = new Date(año, mes, día, horas, minutos, segundos, milisegundos);
// (el mes comienza en 0, Enero sería 0, Febrero 1, etc.)
```

Tabla Propiedades del objeto Date

Propiedad	Descripción
constructor	Devuelve la función que creó el objeto Date
prototype	Te permite añadir propiedades y métodos a un objeto

Tabla Métodos del objeto Date

Método	Descripción
getDate()	Devuelve e día del mes (de 1-31)
getDay ()	Devuelve el día de la semana (de 0-6)
getFullYear()	Devuelve e año {4 dígitos)
getHours{)	Devuelve la hora (de 0-23)
getMilliseconds	Devuelve los milisegundos (de 0-999)
getMinutes()	Devuelve los minutos (de 0-59)
getMonth()	Devuelve el mes (de 0-11)
getSeconds()	Devuelve los segundos (de 0-59)
getTime()	Devuelve los milisegundos desde media noche del 1 de Enero de 1970
getTimezoneOffset()	Devuelve la diferencia de tiempo entre GMT y la hora local, en minutos
getUTCDate()	Devuelve e día del mes en base a la hora UTC (de 1-31)
getUTCDay()	Devuelve el día de la semana en base a la hora UTC (de 0-6)
getUTCFullYear()	Devuelve el año en base a ¡a hora UTC (4 dígitos)
getDate {)	Ajusta el día del mes del objeto (de 1-31)
setFullYear()	Ajusta el año del objeto (4 dígitos)
setHours()	Ajusta la hora del objeto (de 0-23)

Algunos ejemplos de uso:

```
var d = new Date();
document.write(d.getFullYear());
document.write(d.getMonth());
document.write(d.getUTCDay());
var d2 = new Date(5,28,2011,22,58,00);
d2.setMonth(0);
d.setFullYear(2020);
```

8.6. Expresiones Regulares

Las expresiones regulares son patrones de búsqueda, que se pueden utilizar para encontrar texto que coincida con el patrón especificado.

Ejemplo de búsqueda de una cadena de texto sin usar expresiones regulares:

```
var texto = "La linea de alta velocidad llegará pronto a toda España,";
var subcadena = "velocidad";
var i = texto.indexOf(subcadena); // devuelve 17, índice de donde se encuentra la subcadena
if (i != -1) // correcto, se ha encontrado la subcadena
```

Este código funciona porque estamos buscando una subcadena de texto exacta. ¿Pero qué pasaría si hiciéramos una búsqueda más general? Por ejemplo, si quisiéramos buscar la cadena “car” en textos como “cartón”, “bicarbonato”, “practicar”, ¿...?

Cuando estamos buscando cadenas que cumplen un patrón en lugar de una cadena exacta, necesitaremos usar expresiones regulares. Podrías intentar hacerlo con funciones de **String**, pero al final, es mucho más sencillo hacerlo con expresiones regulares, aunque la sintaxis de las mismas es un poco extraña y no necesariamente muy amigable.

En JavaScript las expresiones regulares se gestionan a través del objeto **RegExp**.

Para crear un literal del tipo **RegExp**, tendrás que usar la siguiente sintaxis:

```
var expresion = /expresión regular/;
```

La expresión regular está contenida entre la barras ***/***, y fíjate que no lleva comillas. Las comillas sólo se pondrán en la expresión regular, cuando formen parte del patrón en si mismo.

Las expresiones regulares están hechas de caracteres, solos o en combinación con caracteres especiales, que se proporcionarán para búsquedas más complejas. Por ejemplo, lo siguiente es una expresión regular que realiza una búsqueda que contenga las palabras *Aloe Vera*, en ese orden y separadas por uno o más espacios en medio:

```
var expresion=/Aloe\s+Vera/;
```

Los caracteres especiales en este ejemplo son, la barra invertida (****), que tiene dos efectos: o bien se utiliza con un carácter regular, para indicar que se trata de un carácter especial, o se usa con un carácter especial, tales como el signo más (**+**), para indicar que el carácter debe ser tratado literalmente. En este caso, la barra invertida se utiliza con "**s**", que transforma la letra s en un carácter especial indicando un espacio en blanco, un tabulador, un salto de línea, etc. El símbolo **+** indica que el carácter anterior puede aparecer una o más veces.

Caracteres especiales utilizados en Expresiones Regulares

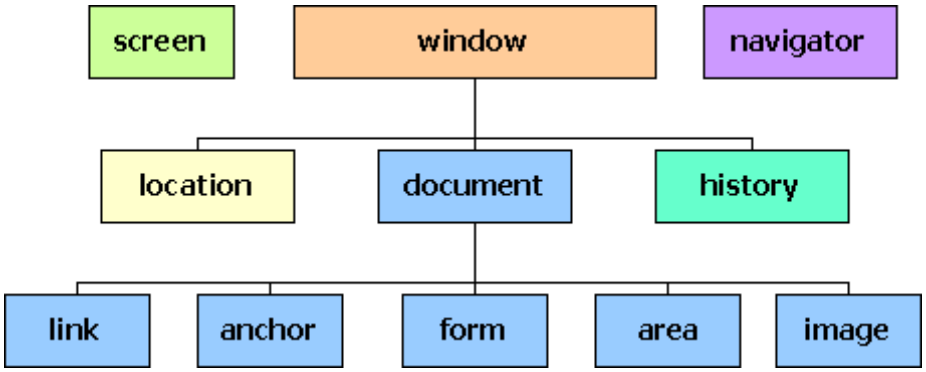
Carácter	Coincidencias	Patrón	Ejemplo de cadena
^	Al inicio de una cadena	**/^Esto/**	Coincidencia en "Esto es...".
$	Al final de la cadena	**/final$/**	Coincidencia en "Esto es el final".
*****	Coincide 0 o más veces	**/se*/**	Que la "e" aparezca 0 o más veces: "seeee" y también "se".
?	Coincide 0 o 1 vez	**/ap?**	Que la p aparezca 0 o 1 vez: "apple" y "and".
+	Coincide 1 o más veces	**/ap+/**	Que la "p" aparezca 1 o más veces: "apple" pero no "and".
{n}	Coincide exactamente n veces	**/ap{2}/**	Que la "p" aparezca exactamente 2 veces: "apple" pero no "apabullante".
{n,}	Coincide n o más veces	**/ap{2,}/**	Que la "p" aparezca 2 o más veces: "apple" y "appple" pero no en "apabullante".
{n,m}	Coincide al menos n, y máximo m veces	**/ap{2,4}/**	Que la "p" aparezca al menos 2 veces y como máximo 4 veces: "apppppple" (encontrará 4 "p").
.	Cualquier carácter excepto nueva línea	**/a.e/**	Que aparezca cualquier carácter, excepto nueva línea entre la a y la e: "ape" y "axe".
[...]	Cualquier carácter entre corchetes	**/a[px]e/**	Que aparezca alguno de los caracteres "p" o "x" entre la a y la e: "ape", "axe", pero no "ale".
[^...]	Cualquier carácter excepto los que están entre corchetes	**/a[^px]/**	Que aparezca cualquier carácter excepto la "p" o la "x" después de la letra a: "ale", pero no "axe" o "ape".
\b	Coincide con el inicio de una palabra	**/\bno/**	Que "no" esté al comienzo de una palabra: "novedad".

Carácter	Coincidencias	Patrón	Ejemplo de cadena
\B	Coincide al final de una palabra	**/\Bno/**	Que "no" esté al final de una palabra: "este invierno" ("no" de "invierno").
\d	Dígitos del 0 al 9	**/\d{3}/**	Que aparezcan exactamente 3 dígitos: "Ahora en 456".
\D	Cualquier carácter que no sea un dígito	**/\D{2,4}/**	Que aparezcan mínimo 2 y máximo 4 caracteres que no sean dígitos: encontrará la cadena "Ahor" en "Ahora en 456".
\w	Coincide con caracteres del tipo (letras, dígitos, subrayados)	**/\w/**	Que aparezca un carácter (letra, dígito o subrayado): "J" en "JavaScript".
\W	Coincide con caracteres que no sean (letras, dígitos, subrayados)	**/\W/**	Que aparezca un carácter (que no sea letra, dígito o subrayado): "%" en "100%".
	Coincide con una nueva línea		
\s	Coincide con un espacio en blanco		
\S	Coincide con un carácter que no es un espacio en blanco		
	Un tabulador		
(x)	Capturando paréntesis		Recuerda los caracteres.
\r	Un retorno de carro		
?=n	Cualquier cadena que está seguida por la cadena n indicada después del igual.	**/la(?= mundo)**	Hola mundo mundial.

9. INTERACCIÓN CON EL NAVEGADOR. OBJETOS PREDEFINIDOS ASOCIADOS

Los navegadores ofrecen al programador multitud de características en forma de un modelo jerárquico. Esta jerarquía es lo que se llama modelo de objetos del navegador y mediante ella se pueden controlar características propias del navegador desde qué mensaje mostrar en la barra de estado hasta la creación de nuevas páginas con el aspecto deseado.

La jerarquía de dichos objetos toma la siguiente forma:



Árbol de objetos HTML DOM

9.1. Objeto window

Un objeto window también se podrá referenciar mediante la palabra self, cuando estamos haciendo la referencia desde el propio documento contenido en esa ventana:

```
self.nombrePropiedad
self.nombreMétodo([parámetros] )
```

Podremos usar cualquiera de las dos referencias anteriores, pero intentaremos dejar la palabra reservada self, para scripts más complejos en los que tengamos múltiples marcos y ventanas.

Debido a que el objeto window siempre estará presente cuando ejecutemos nuestros scripts, podremos omitirlo, en referencias a los objetos dentro de esa ventana. Así que, si escribimos:

```
nombrePropiedad
nombreMétodo( [parámetros] )
```

Se trata del objeto más alto en la jerarquía del navegador (navigator es un objeto independiente de todos en la jerarquía), pues todos los componentes de una página web están situados dentro de una ventana. El objeto window hace referencia a la ventana actual. Veamos a continuación sus propiedades y sus métodos.

Tabla Propiedades del objeto window

Propiedad	Descripción
closed	Devuelve un valor Boolean indicando cuando una ventana ha sido cerrada o no
defaultStatus	Ajusta o devuelve el valor por defecto de la barra de estado de una ventana
document	Devuelve el objeto document para la ventana
frames	Devuelve un array de todos los marcos (incluidos iframes) de la ventana actual
history	Devuelve el objeto history de la ventana
length	Devuelve el número de frames (incluyendo iframes) que hay en dentro de una ventana
location	Devuelve la Localización del objeto Ventana (URL del fichero)
name	Ajusta o devuelve el nombre de una ventana
navigator	Devuelve el objeto navigator de una ventana
opener	Devuelve la referencia a la ventana que abrió la ventana actual
parent	Devuelve la ventana padre de la Ventana actual
self	Devuelve la ventana actual
status	Ajusta el texto de la barra de estado de una ventana

Tabla Métodos del objeto window

Método	Descripción
alert()	Muestra una ventana emergente de alerta y un botón de aceptar
blur()	Elimina el foco de la ventana actual
clearInterval ()	Resetea el cronómetro ajustado
setInterval()	Llama a una función o evalúa una expresión en un intervalo especificado (en milisegundos)
close()	Cierra la ventana actual
confirm()	Muestra una ventana emergente con un mensaje, un botón de aceptar y un botón de cancelar
focus()	Coloca el foco en la ventana actual
open()	Abre una nueva ventana de navegación
prompt()	Muestra una ventana de diálogo para introducir datos

Existen otras propiedades y métodos como **innerHeight, innerWidth, outerHeight, outerWidth, pageXOffset, pageYOffset, personalbar, scrollbars, back(), find(["cadena"],[caso,bkwd]), forward(), home(), print(), stop()...** todas ellas disponibles a partir de NS 4 y cuya explicación remito como ejercicio al lector interesado en saber más sobre el objeto window.

```
<html>
  <head>
  <title>ejemplo de javascript</title>
  <script type="text/javascript">
     <!--
          function moverventana()
    {
      mi_ventana.moveby(5,5);
      i++;
      if (i<20)
          settimeout('moverventana()',100);
       else
          mi_ventana.close();
    }
     //-->
  </script>
  </head>
  <body>
  <script type="text/javascript”>
     <!--
    var opciones="left=100,top=100,width=250,height=150", i= 0;
    mi_ventana = window.open("","",opciones);
    mi_ventana.document.write("una prueba de abrir ventanas");
    mi_ventana.moveto(400,100);
    moverventana();
     //-->
  </script>
</body>
```

9.2. Objeto document

Para la generación de texto y elementos HTML desde código, se utilizan las propiedades y métodos del objeto document.

Cada documento cargado en una ventana del navegador, será un objeto de tipo document.

El objeto document proporciona a los scripts, el acceso a todos los elementos HTML dentro de una página.

Este objeto forma parte además del objeto window, y puede ser accedido a través de la propiedad `window.document` o directamente document (ya que podemos omitir la referencia a la window actual).

Tabla Colecciones del objeto document

Colecciones	Descripción
anchors[]	Es un array que contiene todos los hiperenlaces del document
applets[]	Es un array que contiene todos los applets del document
forms[]	Es un array que contiene todos los formularios dei document
images[]	Es un array que contiene todas las imágenes del document
links[]	Es un array que contiene todos los enlaces del documento

Tabla Propiedades del objeto document

Propiedades	Descripción
cookie	Devuelve todos los nombres/valores de las cookies en el document
domain	Cadena que contiene el nombre de dominio del servidor que cargó el document
lastmodified	Devuelve la fecha y hora de la última modificación del documento
readystate	Devuelve el estado de carga del document actual
referrer	Cadena que contiene la URL del document desde el cual llegamos al docuemnto actual
title	Devuelve o ajusta el título del documento
URL	Devuelve la URL completa del documento

Tabla Métodos del objeto document

Métodos	Descripción
close()	Cierra el flujo abierto previamente con document.open()
getElementById()	Para acceder al elemento especificado por el id escrito entre paréntesis
getElementsByTagName()	Para acceder a los elementos identificados por el atributo name escrito entre paréntesis
open()	Abre el flujo de escritura para poder utilizer document.write o document.writeln en el documento
write()	Para poder escribir expresiones HTML o código de JavaScript dentro del documento
writeln()	Lo mismo que write() pero añade un salto de línea al final de la instrucción

9.3. Objeto location

Este objeto contiene la URL actual, así como algunos datos de interés respecto a esta URL. Su finalidad principal es, por una parte, modificar el objeto location para cambiar a una nueva URL, y extraer los componentes de dicha URL de forma separada para poder trabajar con ellos de forma individual si es el caso. Recordemos que la sintaxis de una URL era:

```
protocolo://maquina_host[:puerto]/camino_al_recurso
```

Tabla Propiedades del objeto location

Propiedades	Descripción
hash	Cadena que contiene el nombre del enlace, dentro de la URL
host	Cadena que contiene el nombre del servidor y el número del puerto, dentro de la URL
hostname	Cadena que contiene el nombre de dominio del servidor (o la dirección IP), dentro de URL
href	Cadena que contiene la URL completa
pathname	Cadena que contiene el camino al recurso, dentro de la URL
port	Cadena que contiene el número de puerto del servidor, dentro de la URL
protocol	Cadena que contiene el protocolo utilizado (incluyendo los dos puntos), dentro de URL
search	Cadena que contiene la información pasada en una llamada a un script, dentro de la URL

Tabla Métodos del objeto location

Métodos	Descripción
assign()	Carga un nuevo documento
reload()	Vuelve a cargar la URL especificada en la propiedad href del objeto location
replace()	Reemplaza el historial actual mientras carga la URL especificada en cadenaURL.

```
<html>
  <head>
    <title>ejemplo de javascript</title>
  </head>
  <body>
  <script language="javascript">
  <!--
  document.write("location <b>href</b>: " + location.href + "<br/>");
  document.write("location <b>host</b>: " + location.host + "<br/>");
  document.write("location <b>hostname</b>: " + location.hostname +
"<br/>");
  document.write("location <b>pathname</b>: " + location.pathname +
"<br/>");
  document.write("location <b>port</b>: " + location.port + "<br/>");
  document.write("location <b>protocol</b>: " + location.protocol +
"<br/>");
     //-->
  </script>
 </body>
</html>
```

9.4. Objeto navigator

Este objeto navigator, contiene información sobre el navegador que estamos utilizando cuando abrimos una URL o un documento local.

Propiedades

Propiedades	Descripción
appCodeName	Cadena que contiene el nombre en Código del navegador
appName	Cadena que contiene el nombre del cliente
appVersion	Cadena que contiene información sobre la versión del cliente
cookieEnabled	Determina si las cookies están o no habilitabas en el navegador
platform	Cadena con la plataforma sobre la que se está ejecutando el programa cliente
userAgent	Cadena que contiene la cabecera completa del agente enviada en una petición HTTP. Contiene la información de las propiedades appCodeName y appVersion

Métodos

Métodos	Descripción
javaEnabled()	Devuelve true si el cliente permite la utilización de Java, en caso contrario, devuelve false

```
<html>
  <head>
      <title>ejemplo de javascript</title>
  </head>
  <body>
    <script language="javascript">
      <!-- document.write("navigator <b>appcodename</b>:
" + navigator.appcodename + "<br/>");
      document.write("navigator <b>appname</b>: " + navigator.appname
+ "<br/>");
       document.write("navigator <b>appversion</b>:" +
navigator.appversion + "<br/>");
       document.write("navigator <b>language</b>: " +
navigator.language + "<br/>");
       document.write("navigator <b>platform</b>: " +
navigator.platform + "<br/>");
       document.write("navigator <b>useragent</b>: " +
navigator.useragent + "<br/>"); //-->
    </script>
  </body>
</html>
```

9.5. Objeto history

El objeto History almacena las referencias de todos los sitios web visitados. Estas referencias se guardan en una lista y sus propiedades y métodos se utilizan principalmente para que el usuario de una aplicación web pueda desplazarse para adelante y para atrás. Sin embargo, al ser una lista de referencias, no podemos acceder a los nombres de las direcciones URL visitadas, ya que son información privada del usuario.

Propiedades del objeto History

Propiedades	Descripción
current	Corresponde a la cadena que contiene la URL de la entrada actual del historial
length	Corresponde al número de páginas que han sido visitadas
next	Corresponde a la cadena que contiene ia siguiente entrada del historial
previous	Corresponde a la cadena que contiene la anterior entrada del historial

Métodos del objeto History

Métodos	Descripción
back()	Carga la URL del documento anterior del historial
forward()	Carga la URL del document siguiente del historial
go()	Carga la URL del documento especificado por el índice que pasamos como parámetro dentro del historial

Por ejemplo, en el fichero HTML que queramos usar este objeto, podríamos crear dos botones para que el usuario pueda desplazarse adelante o atrás según su historial de navegación.

```
<form>
   <input type="button" value="Atras" onClick="history.back();">
   <input type="button" value="Adelante" onClick="history.forward;">
</form>
```

9.6. Objeto screen

El objeto Screen corresponde a la pantalla utilizada por el usuario. Este objeto cuenta con seis propiedades, aunque no posee ningún método. Todas sus propiedades son solamente de lectura, lo que significa que podemos consultar los valores de las propiedades del objeto, pero no las podemos modificar.

Propiedades del objeto Screen

Propiedades	Descripción
availHeight	Corresponde a la altura disponible de la pantalla para el uso de ventanas
availWidth	Corresponde a la anchura disponible de la pantalla para el uso de ventanas
colorDepth	Corresponde al número de colores que puede representar la pantalla
height	Corresponde a la altura total de la pantalla
pixelDepth	Corresponde a la resolución de la pantalla expresada en bits por pixel
width	Corresponde a la anchura total de la pantalla

La altura y anchura disponibles para las ventanas es menor que la altura y anchura total de la pantalla, debido a que cada sistema operativo ocupa una parte de la pantalla con barras de tareas o menús.

El uso del objeto Screen y sus propiedades, es muy común en los diseñadores de páginas web que necesitan saber la resolución de la pantalla del usuario para poder adaptar sus diseños antes de cargarlos. Otro uso bastante común de este objeto, es consultar todas sus propiedades con el fin de adaptar la posición y tamaño de las ventanas emergentes que abra la aplicación web.

```
<script type="text/javascript">
     document.write("<br/>Altura total: " + screen.height) ;
     document.write("<br/>Altura disponible: " + screen.availHeight);
     document.write("<br/>Anchura total: " + screen.width);
     document.write("<br/>Anchura disponible: " + screen.availWidth);
     document.write("<br/>Profundidad de color: " +
screen.colorDepth);
</script>
```

10. MODELO DE OBJETOS DEL DOCUMENTO (DOM)

10.1. Bases del modelo de objetos del documento

El DOM (Modelo de Objetos del Documento), es esencialmente una interfaz de programación de aplicaciones (API), que proporciona un conjunto estándar de objetos, para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML, que es para lo que se diseñó principalmente. El responsable del DOM es el W3C.

El DOM está separado en 3 partes / niveles:

- DOM Core – modelo estándar para cualquier documento estructurado.
- DOM XML – modelo estándar para documentos XML.
- DOM HTML – modelo estándar para documentos HTML.

EL DOM HTML es un estándar dónde se define cómo acceder, modificar, añadir o borrar elementos HTML.

En el DOM se definen los **objetos** y **propiedades** de todos los elementos del documento, y los métodos para acceder a ellos.

10.2. Árbol de nodos

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

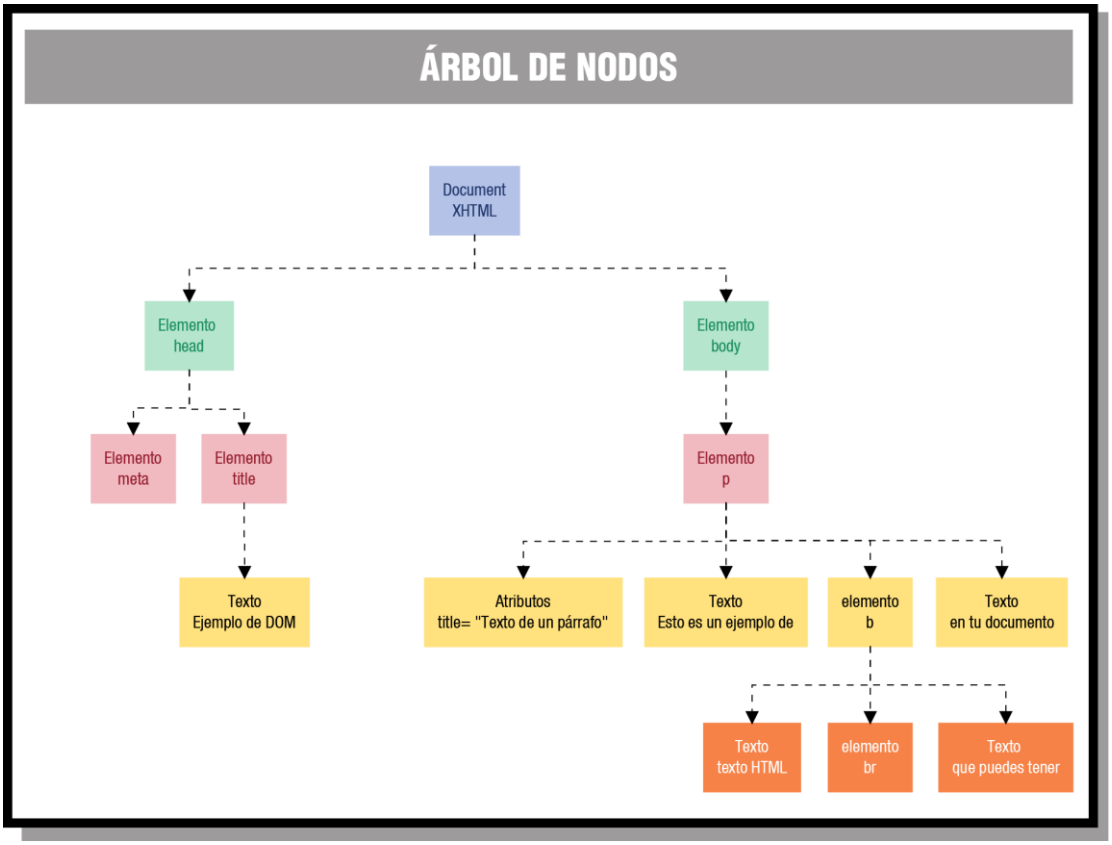
Todas estas tareas habituales son muy sencillas de realizar gracias a DOM. Sin embargo, para poder utilizar las utilidades de DOM, es necesario "transformar" la página original. Una página HTML normal no es más que una sucesión de caracteres, por lo que es un formato muy difícil de manipular. Por ello, los navegadores web transforman automáticamente todas las páginas web en una estructura más eficiente de manipular.

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

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

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1ransitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Ejemplo de DOM</title>
</head>
<body>
    <p title="Texto de un párrafo">Esto es un ejemplo de <b>texto HTML<br />que puedes tener</b> en tu documento.</p>
</body>
</html>
```

Ese código se transformaría en el siguiente árbol de nodos:



Cada rectángulo del gráfico representa un nodo del DOM, y las líneas indican cómo se relacionan los nodos entre sí. La raíz del árbol de nodos es un nodo especial, denominado "**document**". A partir de ese nodo, cada etiqueta XHTML se transformará en nodos de tipo "**elemento**" o "**texto**". Los nodos de tipo "texto", contendrán el texto encerrado para esa etiqueta XHTML. Esta conversión se realiza de forma jerárquica. El nodo inmediatamente superior será el **nodo padre** y todos los nodos que están por debajo serán **nodos hijos**.

10.3. Tipos de nodos

La especificación del DOM define 12 tipos de nodos, aunque generalmente nosotros emplearemos solamente cuatro o cinco tipos de nodos:

- **Document**, es el nodo raíz y del que derivan todos los demás nodos del árbol.
- **Element**, representa cada una de las etiquetas XHTML. Es el único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- **Attr**, con este tipo de nodos representamos los atributos de las etiquetas XHTML, es decir, un nodo por cada atributo=valor.

- **Text**, es el nodo que contiene el texto encerrado por una etiqueta XHTML.
- **Comment**, representa los comentarios incluidos en la página XHTML.

Los otros tipos de nodos pueden ser: **CdataSection**, **DocumentFragment**, **DocumentType**, **EntityReference**, **Entity**, **Notation** y **ProcessingInstruction**.

10.4. Acceso a los nodos

Cuando ya se ha construido automáticamente el árbol de nodos del DOM, ya podemos comenzar a utilizar sus funciones para acceder a cualquier nodo del árbol. El acceder a un nodo del árbol, es lo equivalente a acceder a un trozo de la página de nuestro documento. Así que, una vez que hemos accedido a esa parte del documento, ya podemos modificar valores, crear y añadir nuevos elementos, moverlos de sitio, etc.

Para acceder a un nodo específico (elemento XHTML) lo podemos hacer empleando dos métodos: o bien a través de los nodos padre, o bien usando un método de acceso directo. A través de los nodos padre partiremos del nodo raíz e iremos accediendo a los nodos hijo, y así sucesivamente hasta llegar al elemento que deseemos. Y para el método de acceso directo, que por cierto es el método más utilizado, emplearemos funciones del DOM, que nos permiten ir directamente a un elemento sin tener que atravesar nodo a nodo.

Algo muy importante que tenemos que destacar es, que para que podamos acceder a todos los nodos de un árbol, el árbol tiene que estar completamente construido, es decir, cuando la página XHTML haya sido cargada por completo, en ese momento es cuando podremos acceder a cualquier elemento de dicha página.

Consideremos el siguiente ejemplo y veamos las formas de acceso:

```
<input type="text" id="apellidos" name="apellidos" />
```

- **`getElementsByName()`**

Esta función obtiene una colección, que contiene todos los elementos de la página XHTML cuyo atributo name coincida con el indicado como parámetro.

```
var elementos = document.getElementsByName("apellidos");
```

Si sólo tenemos un elemento con name="apellidos" para acceder a él haremos:

```
var elemento = document.getElementsByName("apellidos")[0];
```

Por ejemplo, si tuviéramos 3 elementos con el atributo name="apellidos" para acceder al segundo elemento haríamos:

```
var segundo = document.getElementsByName("apellidos")[1];
```

- **`getElementsByTagName()`**

Esta función es muy similar a la anterior y también devuelve una colección de elementos cuya etiqueta XHTML coincida con la que se pasa como parámetro. Por ejemplo:

```
var elementos = document.getElementsByTagName("input");
// Este array de elementos contendrá todos los elementos input del
documento.
var cuarto = document.getElementsByTagName("input")[3];
```

- **`getElementById()`**

Esta función es la más utilizada, ya que nos permite acceder directamente al elemento por el ID. Entre paréntesis escribiremos la cadena de texto con el ID. Es muy importante que el ID sea único para cada elemento de una misma página. La función nos devolverá únicamente el nodo buscado.

Por ejemplo:

```
var elemento= document.getElementById("apellidos");
```

Si tenemos por ejemplo una tabla con id="datos" y queremos acceder a todas las celdas de esa tabla, tendríamos que combinar getElementById con getElementsByTagName. Por ejemplo:

```
var miTabla= document.getElementById("datos");
var filas= miTabla.getElementsByTagName("td");
```

10.5. Creación y eliminación de nodos

Podremos crear elementos y luego insertarlos en el DOM, y la actualización quedará reflejada automáticamente por el navegador. También podremos mover nodos ya existentes simplemente insertándolo en cualquier otro lugar del árbol del DOM.

Ten en cuenta que cuando estemos creando nodos de elementos, el elemento debe estar en minúsculas. Aunque en HTML esto daría igual, el XHTML sí que es sensible a mayúsculas y minúsculas y tendrá que ir, por lo tanto, en minúsculas.

Usaremos los métodos **`createElement()`, `createTextNode()` y `appendChild()`,** que nos permitirán crear un elemento, crear un nodo de texto y añadir un nuevo nodo hijo.

Ejemplo de creación de un nuevo párrafo, suponiendo que partimos del siguiente código HTML:

```
<p title="Texto de un párrafo" id="parrafito">Esto es un ejemplo de
<b>texto HTML<br />que puedes tener</b> en tu documento.</p>
```

Para crear el nuevo párrafo haremos:

```
var nuevoParrafo = document.createElement('p');
var nuevoTexto = document.createTextNode('Contenido añadido al
párrafo.');
nuevoParrafo.appendChild(nuevoTexto);
document.getElementById('parrafito').appendChild(nuevoParrafo);
```

Y obtendremos como resultado HTML:

```
<p id="parrafito" title="Texto de un párrafo"> Esto es un ejemplo de
<b>texto HTML<br>que puedes tener</b>en tu documento. <p>Contenido
añadido al párrafo.</p> </p>
```

Podríamos haber utilizado **`insertBefore`** en lugar de **`appendChild`** o, incluso, añadir manualmente el nuevo elemento al final de la colección de nodos **`childNodes`**. Si usamos **`replaceChild`**, incluso podríamos sobrescribir nodos ya existentes. También es posible copiar un nodo usando **`cloneNode(true)`.** Esto devolverá una copia del nodo, pero no lo añade automáticamente a la colección **`childNodes`.**

Para eliminar un nodo existente, lo podremos hacer con **`element.removeChild`**(referencia al nodo hijo).

Ejemplo de creación de elementos e inserción en el documento:

```
//Creamos tres elementos nuevos: p, b, br
var elementoP = document.createElement('p');
var elementoB = document.createElement('b');
var elementoBR = document.createElement('br');

//Le asignamos un nuevo atributo title al elementoP que hemos creado.
elementoP.setAttribute('title','Parrafo creado desde JavaScript');

//Preparamos los nodos de texto
var texto1 = document.createTextNode('Con JavaScript se ');
var texto2 = document.createTextNode('pueden realizar ');
var texto3 = document.createTextNode('un monton');
var texto4 = document.createTextNode(' de cosas sobre el documento.');

//Añadimos al elemento B los nodos de texto2, elemento BR y texto3.
elementoB.appendChild(texto2);
elementoB.appendChild(elementoBR);
elementoB.appendChild(texto3);

//Añadimos al elemento P los nodos de texto1, elemento B y texto 4.
elementoP.appendChild(texto1);
elementoP.appendChild(elementoB);
elementoP.appendChild(texto4);

//insertamos el nuevo párrafo como un nuevo hijo de nuestro párrafo
document.getElementById('parrafito').appendChild(elementoP);
```

11. EVENTOS

Hasta ahora, todas las aplicaciones y scripts que se han creado tienen algo en común: se ejecutan desde la primera instrucción hasta la última de forma secuencial. Gracias a las estructuras de control de flujo (`if`, `for`, `while`) es posible modificar ligeramente este comportamiento y repetir algunos trozos del script y saltarse otros trozos en función de algunas condiciones.

Este tipo de aplicaciones son poco útiles, ya que no interactúan con los usuarios y no pueden responder a los diferentes eventos que se producen durante la ejecución de una aplicación. Afortunadamente, las aplicaciones web creadas con el lenguaje JavaScript pueden utilizar el modelo de programación basada en eventos.

En este tipo de programación, los scripts se dedican a esperar a que el usuario "haga algo" (que pulse una tecla, que mueva el ratón, que cierre la ventana del navegador). A continuación, el script responde a la acción del usuario normalmente procesando esa información y generando un resultado.

Los eventos hacen posible que los usuarios transmitan información a los programas. JavaScript define numerosos eventos que permiten una interacción completa entre el usuario y las páginas/aplicaciones web. La pulsación de una tecla constituye un evento, así como pinchar o mover el ratón, seleccionar un elemento de un formulario, redimensionar la ventana del navegador, etc.

JavaScript permite asignar una función a cada uno de los eventos. De esta forma, cuando se produce cualquier evento, JavaScript ejecuta su función asociada. Este tipo de funciones se denominan "event handlers" en inglés y suelen traducirse por "manejadores de eventos".

11.1. Modelos de eventos

Crear páginas y aplicaciones web siempre ha sido mucho más complejo de lo que debería serlo debido a las incompatibilidades entre navegadores. A pesar de que existen decenas de estándares para las tecnologías empleadas, los navegadores no los soportan completamente o incluso los ignoran.

Las principales incompatibilidades se producen en el lenguaje XHTML, en el soporte de hojas de estilos CSS y, sobre todo, en la implementación de JavaScript. De todas ellas, la incompatibilidad más importante se da precisamente en el modelo de eventos del navegador. Así, existen hasta tres modelos diferentes para manejar los eventos dependiendo del navegador en el que se ejecute la aplicación.

1. **Modelo básico de eventos.** Este modelo simple de eventos se introdujo para la versión 4 del estándar HTML y se considera parte del nivel más básico de DOM. Aunque sus características son limitadas, es el único modelo que es compatible en todos los navegadores y por tanto, el único que permite crear aplicaciones que funcionan de la misma manera en todos los navegadores.
2. **Modelo de eventos estándar.** Las versiones más avanzadas del estándar DOM (DOM nivel 2) definen un modelo de eventos completamente nuevo y mucho más poderoso que el original. Todos los navegadores modernos lo incluyen, salvo Internet Explorer.

3. **Modelo de eventos de Internet Explorer.** Internet Explorer utiliza su propio modelo de eventos, que es similar pero incompatible con el modelo estándar. Se utilizó por primera vez en Internet Explorer 4 y Microsoft decidió seguir utilizándolo en el resto de versiones, a pesar de que la empresa había participado en la creación del estándar de DOM que define el modelo de eventos estándar

11.2. Modelo básico de eventos

11.2.1. Tipos de eventos

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

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

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

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

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

Algunos eventos de la tabla anterior (**onclick, onkeydown, onkeypress, onreset, onsubmit**) permiten evitar la "acción por defecto" de ese evento.

Las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar por ejemplo sobre un botón de tipo **<input type="submit">** se desencadenan los eventos **onmousedown, onclick, onmouseup y onsubmit** de forma consecutiva.

11.2.2. Manejadores de eventos

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

Las funciones o código JavaScript que se definen para cada evento se denominan "manejador de eventos" y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:

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

11.2.2.1. Manejadores como atributos de los elementos XHTML

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

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

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

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

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

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

```
encima
</div>
```

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

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

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

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

11.2.2.2. Manejadores de eventos y variable this

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

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

Elemento original:

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

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

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

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

siguiente manera:

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

El código anterior es mucho más compacto, más fácil de leer y de escribir y sigue funcionando correctamente, aunque se modifique el valor del atributo id del .

11.2.2.3. Manejadores de eventos como funciones externas

La definición de los manejadores de eventos en los atributos XHTML es el método más sencillo, pero menos aconsejable de tratar con los eventos en JavaScript. El principal inconveniente es que se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para los casos más sencillos.

Si se realizan aplicaciones complejas, como por ejemplo la validación de un formulario, es aconsejable agrupar todo el código JavaScript en una función externa y llamar a esta función desde el elemento XHTML.

Siguiendo con el ejemplo anterior que muestra un mensaje al pinchar sobre un botón:

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

Utilizando funciones externas se puede transformar en:

```
function muestraMensaje() {
      alert('Gracias por pinchar');
}

<input type="button" value="Pinchame y verás"
onclick="muestraMensaje()"
/>
```

Esta técnica consiste en extraer todas las instrucciones de JavaScript y agruparlas en una función externa. Una vez definida la función, en el atributo del elemento XHTML se incluye el nombre de la función, para indicar que es la función que se ejecuta cuando se produce el evento.

La llamada a la función se realiza de la forma habitual, indicando su nombre seguido de los paréntesis y de forma opcional, incluyendo todos los argumentos y parámetros que se necesiten.

El principal inconveniente de este método es que en las funciones externas no se puede seguir utilizando la variable this y por tanto, es necesario pasar esta variable como parámetro a la función:

```
function resalta(elemento) {
      switch(elemento.style.borderColor) {
         case 'silver':
```

```
          case 'silver silver silver silver':
          case '#c0c0c0':
              elemento.style.borderColor = 'black';
              break;
          case 'black':
          case 'black black black black':
          case '#000000':
             elemento.style.borderColor = 'silver';
             break;
      }
}
<div style="width:150px; height:60px; border:thin solid silver"
onmouseover="resalta(this)" onmouseout="resalta(this)">
      Sección de contenidos...
</div>
```

En el ejemplo anterior, la función externa es llamada con el parámetro this, que dentro de la función se denomina elemento. La complejidad del ejemplo se produce sobre todo por la forma en la que los distintos navegadores almacenan el valor de la propiedad borderColor.

Mientras que Firefox almacena (en caso de que los cuatro bordes coincidan en color) el valor black, Internet Explorer lo almacena como black black black black y Opera almacena su representación hexadecimal #000000.

11.2.2.4. Manejadores semánticos

Los métodos que se han visto para añadir manejadores de eventos (como atributos XHTML y como funciones externas) tienen un grave inconveniente: "ensucian" el código XHTML de la página.

Como es conocido, una de las buenas prácticas básicas en el diseño de páginas y aplicaciones web es la separación de los contenidos (XHTML) y su aspecto o presentación (CSS). Siempre que sea posible, también se recomienda separar los contenidos (XHTML) y su comportamiento o programación (JavaScript).

Mezclar el código JavaScript con los elementos XHTML solamente contribuye a complicar el código fuente de la página, a dificultar la modificación y mantenimiento de la página y a reducir la semántica del documento final producido.

Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica es una evolución del método de las funciones externas, ya que se basa en utilizar las propiedades DOM de los elementos XHTML para asignar todas las funciones externas que actúan de manejadores de eventos. Así, el siguiente ejemplo:

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

Se puede transformar en:

```
// Función externa
function muestraMensaje() {
      alert('Gracias por pinchar');
```

```
}

// Asignar la función externa al elemento
document.getElementById("pinchable").onclick = muestraMensaje;

// Elemento XHTML
<input id="pinchable" type="button" value="Pinchame y verás" />
```

La técnica de los manejadores semánticos consiste en:

1. Asignar un identificador único al elemento XHTML mediante el atributo id.
2. Crear una función de JavaScript encargada de manejar el evento.
3. Asignar la función externa al evento correspondiente en el elemento deseado.

El último paso es la clave de esta técnica. En primer lugar, se obtiene el elemento al que se desea asociar la función externa:

```
document.getElementById("pinchable");
```

A continuación, se utiliza una propiedad del elemento con el mismo nombre que el evento que se quiere manejar. En este caso, la propiedad es onclick:

```
document.getElementById("pinchable").onclick = ...
```

Por último, se asigna la función externa mediante su nombre sin paréntesis. Lo más importante (y la causa más común de errores) es indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

```
document.getElementById("pinchable").onclick = muestraMensaje;
```

Si se añaden los paréntesis después del nombre de la función, en realidad se está ejecutando la función y guardando el valor devuelto por la función en la propiedad onclick de elemento.

```
// Asignar una función externa a un evento de un elemento
document.getElementById("pinchable").onclick = muestraMensaje;

// Ejecutar una función y guardar su resultado en una propiedad de un
elemento
document.getElementById("pinchable").onclick = muestraMensaje();
```

La gran ventaja de este método es que el código XHTML resultante es muy "limpio", ya que no se mezcla con el código JavaScript. Además, dentro de las funciones externas asignadas sí que se puede utilizar la variable this para referirse al elemento que provoca el evento.

El único inconveniente de este método es que la página se debe cargar completamente antes de que se puedan utilizar las funciones DOM que asignan los manejadores a los elementos XHTML. Una de las formas más sencillas de asegurar que cierto código se va a ejecutar después de que la página se cargue por completo es utilizar el evento onload:

```
window.onload = function() {
      document.getElementById("pinchable").onclick = muestraMensaje;
```

```
}
```

La técnica anterior utiliza el concepto de funciones anónimas, que no se va a estudiar, pero que permite crear un código compacto y muy sencillo. Para asegurarse que un código JavaScript va a ejecutarse después de que la página se haya cargado completamente, sólo es necesario incluir esas instrucciones entre los símbolos { y }:

```
<!DOCTYPE html>
<html>
<head>
<script>
    window.onload=function(){
    var resultado = 5 + 6;
    document.getElementById("solucion").innerHTML = "La suma de 5 + 6 es: " +resultado;
}
</script>
</head>
<body>

<h2>My First Web Page</h2>
<p>My First Paragraph.</p>

<p id="solucion"></p>



</body>
</html>
```

```
window.onload = function() {
      ...
}
```

En el siguiente ejemplo, se añaden eventos a los elementos de tipo input=text de un formulario complejo:

```
function resalta() {
      // Código JavaScript
}

window.onload = function() {
      var formulario = document.getElementById("formulario");
      var camposInput = formulario.getElementsByTagName("input");

      for(var i=0; i<camposInput.length; i++) {
         if(camposInput[i].type == "text") {
            camposInput[i].onclick = resalta;
         }
      }
}
```

11.3. El objeto Event

Generalmente, los manejadores de eventos necesitan información adicional para procesar las tareas que tienen que realizar. Si una función procesa, por ejemplo, el evento click, lo más probable es que necesite conocer la posición en la que estaba el ratón en el momento de realizar el clic; aunque esto quizás tampoco sea muy habitual, a no ser que estemos programando alguna utilidad de tipo gráfico.

Lo que sí es más común es tener información adicional en los eventos del teclado. Por ejemplo, cuando pulsamos una tecla nos interesa saber cuál ha sido la tecla pulsada, o si tenemos a mayores alguna tecla especial pulsada como Alt, Control, etc.

Para gestionar toda esa información disponemos del objeto Event, el cual nos permitirá acceder a esas propiedades adicionales que se generan en los eventos.

Como siempre, los navegadores gestionan de forma diferente los objetos Event.

11.3.1. Propiedades y métodos del objeto event.

Propiedades del objeto Event

PROPIEDADES	DESCRIPCIÓN
altKey, ctrlKey, metaKey, shiftKey	Valor booleano que indica si están presionadas alguna de las teclas **Alt**, **Ctrl**, **Meta** o **Shift** en el momento del evento.
bubbles	Valor booleano que indica si el evento burbujea o no.
button	Valor **integer** que indica que botón del ratón ha sido presionado o soltado, 0=izquierdo, 2=derecho, 1=medio.
cancelable	Valor booleano que indica si el evento se puede cancelar.
charCode	Indica el carácter Unicode de la tecla presionada.
clientX, clientY	Devuelve las coordenadas de la posición del ratón en el momento del evento.
currentTarget	El elemento al que se asignó el evento. Por ejemplo, si tenemos un evento de click en un **divA** que contiene un hijo **divB**. Si hacemos clic en **divB**, **currentTarget** referenciará a **divA** (el elemento dónde se asignó el evento) mientras que **target** devolverá **divB**, el elemento dónde ocurrió el evento.
eventPhase	Un valor integer que indica la fase del evento que está siendo procesada. Fase de captura (1), en destino (2) o fase de burbujeo (3).
layerX, layerY	Devuelve las coordenadas del ratón relativas a un elemento posicionado absoluta o relativamente. Si el evento ocurre fuera de un elemento posicionado se usará la esquina superior izquierda del documento.
pageX, pageY	Devuelve las coordenadas del ratón relativas a la esquina superior izquierda de una página.
relatedTarget	En un evento de "**mouseover**" indica el nodo que ha abandonado el ratón. En un evento de "**mouseout**" indica el nodo hacia el que se ha movido el ratón.
screenX, screenY	Devuelve las coordenadas del ratón relativas a la pantalla dónde se disparó el evento.

target	El elemento dónde se originó el evento, que puede diferir del elemento que tenga asignado el evento. Véase **currentTarget**.
timestamp	Devuelve la hora (en milisegundos desde **epoch**) a la que se creó el evento. Por ejemplo, cuando se presionó una tecla. No todos los eventos devuelven **timestamp**.
type	Una cadena de texto que indica el tipo de evento "**click**", "**mouseout**", "**mouseover**", etc.
which	Indica el Unicode de la tecla presionada. Idéntico a **charCode**, excepto que esta propiedad también funciona en Netscape 4.

Métodos del objeto Event

MÉTODOS	DESCRIPCIÓN
preventDefault()	Cancela cualquier acción asociada por defecto a un evento.
stopPropagation()	Evita que un evento burbujee. Por ejemplo, si tenemos un **divA** que contiene un **divB** hijo. Cuando asignamos un evento de click a **divA**, si hacemos click en **divB**, por defecto se dispararía también el evento en **divA** en la fase de burbujeo. Para evitar ésto se puede llamar a **stopPropagation()** en **divB**. Para ello creamos un evento de click en **divB** y le hacemos **stopPropagation()**.

11.3.2. Eventos del teclado

Uno de los eventos más complicados de gestionar en JavaScript son los eventos de teclado, debido a que suele haber bastantes incompatibilidades entre navegadores, teclados, idiomas, etc.

Para el teclado disponemos de 3 tipos de eventos: **`keydown`**, **`keypress`** y **`keyup`**. Y además disponemos de dos tipos de teclas: las especiales (Shift, Alt, AltGr, Enter, etc.) y las teclas **normales**, que contienen letras, números, y símbolos.

En el **proceso de pulsación de una tecla** se generan tres eventos seguidos: **keydown**, **keypress** y **keyup**. Y para cada uno de ellos disponemos de las propiedades **`keyCode`** y **`charCode`**. Para saber la tecla que se ha pulsado lo más cómodo es acceder al evento **keypress**.

- **`keydown:`** se produce al presionar una tecla y mantenerla presionada.
  - Su comportamiento es el mismo en todos los navegadores.
  - Propiedad **keyCode**: devuelve el código interno de la tecla.
  - Propiedad **charCode**: no está definida.

- **`keypress:`** se produce en el instante de presionar la tecla.
  - Propiedad **keyCode**: devuelve el código interno de las teclas especiales, para las teclas normales no está definido.
  - Propiedad **charCode**: devuelve 0 para las teclas especiales o el código del carácter de la tecla pulsada para las teclas normales.

(En Internet Explorer **keyCode** devuelve el carácter de la tecla pulsada, y **charCode** no está definido).

- **`keyup:`** se produce al soltar una tecla presionada.
  - Su comportamiento es el mismo en todos los navegadores.
    - Propiedad **keyCode**: devuelve el código interno de la tecla.
    - Propiedad charCode: no está definida.

Ejemplo que mueve el foco de un campo de texto a otro, dentro de un formulario, al pulsar la tecla ENTER dentro de cada campo:

```
<form name="formulario" id="formulario">

     <label for="nombre">
          Nombre:
     </label>

<input type="text" id="nombre" name="nombre" />

     <label for="apellidos">
          Apellidos:
     </label>

<input type="text" id="apellidos" name="apellidos" />

     <label for="provincia">
          Provincia:
     </label>

     <input type="text" id="provincia" name="provincia" />
     <input type="button" id="enviar" value="Enviar" />
</form>

<script type="text/javascript">

function cambiar(evt)
{
     if (evt.keyCode==13)  // Código de la tecla Enter
          if (this.nextSibling.nextSibling.type=="text")
               this.nextSibling.nextSibling.focus();
}

var inputs=document.getElementsByTagName("input");

for (i=0; i<inputs.length; i++)
{
     inputs[i].addEventListener("keypress",cambiar,false);
}

</script>
```

En la estructura HTML del formulario, los campos del formulario no llevan saltos de línea entre unos y otros, por las siguientes razones:

- **`this.nextSibling`** - hace referencia al siguiente hermano al actual (la siguiente etiqueta label del siguiente campo).

- **`this.nextSibling.nextSibling`** - hermano siguiente, al hermano del elemento actual. (será otro elemento input. Si pusiéramos un salto de línea entra campos input entonces ya ese `this.nextSibling.nextSibling` ya no sería un campo input y sería un nodo de texto con el carácter del salto de línea que hemos puesto como separador de los campos input).

11.3.3. Eventos del ratón

Los eventos del ratón son uno de los eventos más importantes en JavaScript.

Cada vez que un usuario hace click en un elemento, al menos se disparan tres eventos y en el siguiente orden:

1. **`mousedown`**, cuando el usuario presiona el botón del ratón sobre el elemento.
2. **`mouseup`**, cuando el usuario suelta el botón del ratón.
3. **`click`**, cuando el usuario pulsa y suelta el botón sobre el elemento.

En general, los eventos de **`mousedown`** y **`mouseup`** son mucho más útiles que el evento click.

Si por ejemplo presionamos el botón sobre un elemento A, nos desplazamos y soltamos el botón sobre otro elemento B, se detectarán solamente los eventos de **`mousedown`** sobre A y **`mouseup`** sobre B, pero no se detectará el evento de **`click`**. Ésto quizás pueda suponer un problema, dependiendo del tipo de interacción que quieras en tu aplicación. Generalmente a la hora de registrar eventos, se suele hacer para **`mousedown`** y **`mouseup`**, a menos de que quieras el evento de **`click`** y no ningún otro.

El evento de **`dblclick`** no se usa muy a menudo. Incluso si lo usas, tienes que ser muy prudente y no registrar a la vez **`click`** y **`dblclick`** sobre el mismo elemento, para evitar complicaciones.

El evento de **`mousemove`** funciona bastante bien, aunque tienes que tener en cuenta que la gestión de este evento le puede llevar cierto tiempo al sistema para su procesamiento. Por ejemplo, si el ratón se mueve 1 pixel, y tienes programado el evento de **`mousemove`**, para cada movimiento que hagas, ese evento se disparará, independientemente de si el usuario realiza o no realiza ninguna otra opción. En ordenadores antiguos ésto puede ralentizar el sistema, ya que para cada movimiento del ratón estaría realizando las tareas adicionales programadas en la función. Por lo tanto, se recomienda utilizar este evento sólo cuando haga falta, y desactivarlo cuando hayamos terminado.

Otros eventos adicionales del ratón son los de **`mouseover`** y **`mouseout`**, que se producen cuando el ratón entra en la zona del elemento o sale del elemento. Si, por ejemplo, tenemos tres contenedores anidados divA, divB y divC: si programamos un evento de **`mouseover`** sobre el divA y nos vamos moviendo hacia el contenedor interno, veremos que ese evento sigue disparándose cuando estemos sobre divB o entremos en divC. Esta reacción se debe al burbujeo de eventos. Ni en divB o divC tenemos registrado el evento de **`mouseover`**, pero cuando se produce el burbujeo de dicho evento, se encontrará que tenemos registrado ese evento en el contenedor padre divA y por eso se ejecutará.

Muchas veces es necesario saber de dónde procede el ratón y hacia dónde va, y para ello W3C añadió la propiedad **relatedTarget** a los eventos de **mouseover** y **mouseout**. Esta propiedad contiene el elemento desde dónde viene el ratón en el caso de **mouseover**, o el elemento en el que acaba de entrar en el caso de **mouseout**.

Para saber los botones del ratón que hemos pulsado, disponemos de las propiedades **which** y **button**. Y para detectar correctamente el botón pulsado, lo mejor es hacerlo en los eventos de **mousedown** o **mouseup**. **Which** es una propiedad antigua de Netscape, así que simplemente vamos a citar **button** que es la propiedad propuesta por el W3C:

Los valores de la propiedad **button** pueden ser:

- **Botón izquierdo**: 0
- **Botón medio**: 1
- **Botón derecho**: 2

También es muy interesante conocer la posición en la que se encuentra el ratón, y para ello disponemos de un montón de propiedades que nos facilitan esa información:

- **clientX, clientY:** devuelven las coordenadas del ratón relativas a la ventana.
- **offsetX, offsetY:** devuelven las coordenadas del ratón relativas al objeto destino del evento.
- **pageX, pageY:** devuelven las coordenadas del ratón relativas al documento. Estas coordenadas son las más utilizadas.
- **screenX, screenY:** devuelven las coordenadas del ratón relativas a la pantalla.

Ejemplo que muestra las coordenadas del ratón al moverlo en el documento:

```
<input type="text" id="coordenadas" name="coordenadas" size="12"/>

<script type="text/javascript">

function mostrarCoordenadas(elEvento){
     document.getElementById("coordenadas").value=elEvento.clientX+" :
"+elEvento.clientY;
}

document.addEventListener('mousemove',mostrarCoordenadas,false);

</script>
```

12. FORMULARIOS

La programación de aplicaciones que contienen formularios web siempre ha sido una de las tareas fundamentales de JavaScript. De hecho, una de las principales razones por las que se inventó el lenguaje de programación JavaScript fue la necesidad de validar los datos de los formularios directamente en el navegador del usuario. De esta forma, se evitaba recargar la página cuando el usuario cometía errores al rellenar los formularios.

No obstante, la aparición de las aplicaciones AJAX ha relevado al tratamiento de formularios como la principal actividad de JavaScript. Ahora, el principal uso de JavaScript es el de las comunicaciones asíncronas con los servidores y el de la manipulación dinámica de las aplicaciones. De todas formas, el manejo de los formularios sigue siendo un requerimiento imprescindible para cualquier programador de JavaScript.

12.1. Propiedades básicas de formularios y elementos

JavaScript dispone de numerosas propiedades y funciones que facilitan la programación de aplicaciones que manejan formularios. En primer lugar, cuando se carga una página web, el navegador crea automáticamente un array llamado **forms** y que contiene la referencia a todos los formularios de la página.

Para acceder al array **forms**, se utiliza el objeto **document**, por lo que **document.forms** es el array que contiene todos los formularios de la página. Como se trata de un array, el acceso a cada formulario se realiza con la misma sintaxis de los arrays. La siguiente instrucción accede al primer formulario de la página:

```
document.forms[0];
```

Además del array de formularios, el navegador crea automáticamente un array llamado **elements** por cada uno de los formularios de la página. Cada array **elements** contiene la referencia a todos los elementos (cuadros de texto, botones, listas desplegables, etc.) de ese formulario. Utilizando la sintaxis de los arrays, la siguiente instrucción obtiene el primer elemento del primer formulario de la página:

```
document.forms[0].elements[0];
```

La sintaxis de los arrays no siempre es tan concisa. El siguiente ejemplo muestra cómo obtener directamente el último elemento del primer formulario de la página:

```
document.forms[0].elements[document.forms[0].elements.length-1];
```

Aunque esta forma de acceder a los formularios es rápida y sencilla, tiene un inconveniente muy grave. ¿Qué sucede si cambia el diseño de la página y en el código HTML se cambia el orden de los formularios originales o se añaden nuevos formularios? El problema es que "el primer formulario de la página" ahora podría ser otro formulario diferente al que espera la aplicación.

En un entorno tan cambiante como el diseño web, es muy difícil confiar en que el orden de los formularios se mantenga estable en una página web. Por este motivo, siempre debería evitarse el acceso a los formularios de una página mediante el array **document.forms.**
Una forma de evitar los problemas del método anterior consiste en acceder a los formularios de una página a través de su nombre (atributo **name**) o a través de su **atributo id**. El

objeto **document** permite acceder directamente a cualquier formulario mediante su atributo **name**:

```
var formularioPrincipal = document.formulario;
var formularioSecundario = document.otro_formulario;

<form name="formulario" >
  ...
</form>

<form name="otro_formulario" >
  ...
</form>
```

Accediendo de esta forma a los formularios de la página, el script funciona correctamente, aunque se reordenen los formularios o se añadan nuevos formularios a la página. Los elementos de los formularios también se pueden acceder directamente mediante su atributo name:

```
var formularioPrincipal = document.formulario;
var primerElemento = document.formulario.elemento;

<form name="formulario">
  <input type="text" name="elemento" />
</form>
```

Obviamente, también se puede acceder a los formularios y a sus elementos utilizando las funciones DOM de acceso directo a los nodos. El siguiente ejemplo utiliza la habitual función **`document.getElementById()`** para acceder de forma directa a un formulario y a uno de sus elementos:

```
var formularioPrincipal = document.getElementById("formulario");
var primerElemento = document.getElementById("elemento");
<form name="formulario" id="formulario" >
  <input type="text" name="elemento" id="elemento" />
</form>
```

Independientemente del método utilizado para obtener la referencia a un elemento de formulario, cada elemento dispone de las siguientes propiedades útiles para el desarrollo de las aplicaciones:

- **type:** indica el tipo de elemento que se trata. Para los elementos de tipo <input> (text, button, checkbox, etc.) coincide con el valor de su atributo type. Para las listas desplegables normales (elemento <select>) su valor es select-one, lo que permite diferenciarlas de las listas que permiten seleccionar varios elementos a la vez y cuyo tipo es select-multiple. Por último, en los elementos de tipo <textarea>, el valor de type es textarea.
- **form**: es una referencia directa al formulario al que pertenece el elemento. Así, para acceder al formulario de un elemento, se puede utilizar **document.getElementById("id_del_elemento").form**
- **name**: obtiene el valor del atributo name de XHTML. Solamente se puede leer su valor, por lo que no se puede modificar.

- **value**: permite leer y modificar el valor del atributo value de XHTML. Para los campos de texto (<input type="text"> y <textarea>) obtiene el texto que ha escrito el usuario. Para los botones obtiene el texto que se muestra en el botón. Para los elementos checkboxy radiobutton no es muy útil, como se verá más adelante

Por último, los eventos más utilizados en el manejo de los formularios son los siguientes:

- **`onclick:`** evento que se produce cuando se pincha con el ratón sobre un elemento. Normalmente se utiliza con cualquiera de los tipos de botones que permite definir XHTML (<input type="button">, <input type="submit">, <input type="image">).
- **`onchange:`** evento que se produce cuando el usuario cambia el valor de un elemento de texto (<input type="text"> o <textarea>). También se produce cuando el usuario selecciona una opción en una lista desplegable (<select>). Sin embargo, el evento sólo se produce si después de realizar el cambio, el usuario pasa al siguiente campo del formulario, lo que técnicamente se conoce como que "el otro campo de formulario ha perdido el foco".
- **`onfocus:`** evento que se produce cuando el usuario selecciona un elemento del formulario.
- **`onblur:`** evento complementario de onfocus, ya que se produce cuando el usuario ha deseleccionado un elemento por haber seleccionado otro elemento del formulario. Técnicamente, se dice que el elemento anterior "ha perdido el foco".

12.2. Utilidades básicas para formularios

12.2.1. Obtener el valor de los campos de formulario

La mayoría de técnicas JavaScript relacionadas con los formularios requieren leer y/o modificar el valor de los campos del formulario. Por tanto, a continuación, se muestra cómo obtener el valor de los campos de formulario más utilizados.

12.2.1.1. Cuadro de texto y textarea

El valor del texto mostrado por estos elementos se obtiene y se establece directamente mediante la propiedad value.

```
<input type="text" id="texto" />
```

```
var valor = document.getElementById("texto").value;
```

```
<textarea id="parrafo"></textarea>
```

```
var valor = document.getElementById("parrafo").value;
```

12.2.1.2. Radiobutton

Cuando se dispone de un grupo de *radiobuttons*, generalmente no se quiere obtener el valor del atributo **value** de alguno de ellos, sino que lo importante es conocer cuál de todos los *radiobuttons* se ha seleccionado. La propiedad **checked** devuelve true para el *radiobutton* seleccionado y false en cualquier otro caso. Si por ejemplo se dispone del siguiente grupo de *radiobuttons*:

```
<input type="radio" value="si" name="pregunta" id="pregunta_si"/> SI
<input type="radio" value="no" name="pregunta" id="pregunta_no"/> NO
<input type="radio" value="nsnc" name="pregunta" id="pregunta_nsnc"/>
NS/NC
```

El siguiente código permite determinar si cada radiobutton ha sido seleccionado o no:

```
var elementos = document.getElementsByName("pregunta");

for(var i=0; i<elementos.length; i++) {
  console.log(" Elemento: " + elementos[i].value + "\n Seleccionado: "
+ elementos[i].checked);
}
```

12.2.1.3. Checkbox

Los elementos de tipo *checkbox* son muy similares a los radiobutton, salvo que en este caso se debe comprobar cada *checkbox* de forma independiente del resto. El motivo es que los grupos de radiobutton son mutuamente excluyentes y sólo se puede seleccionar uno de ellos cada vez. Por su parte, los *checkbox* se pueden seleccionar de forma independiente respecto de los demás.

Si se dispone de los siguientes checkbox:

```
<input type="checkbox" value="condiciones" name="condiciones"
id="condiciones"/> He leído y acepto las condiciones
<input type="checkbox" value="privacidad" name="privacidad"
id="privacidad"/> He leído la política de privacidad
```

Utilizando la propiedad checked, es posible comprobar si cada checkbox ha sido seleccionado:

```
var elemento = document.getElementById("condiciones");
console.log(" Elemento: " + elemento.value + "\n Seleccionado: " +
elemento.checked);

elemento = document.getElementById("privacidad");
console.log(" Elemento: " + elemento.value + "\n Seleccionado: " +
elemento.checked);
```

12.2.1.4. Select

Las listas desplegables (**<select>**) son los elementos en los que es más difícil obtener su valor. Si se dispone de una lista desplegable como la siguiente:

```
<select id="opciones" name="opciones">
  <option value="1">Primer valor</option>
  <option value="2">Segundo valor</option>
  <option value="3">Tercer valor</option>
  <option value="4">Cuarto valor</option>
</select>
```

En general, lo que se requiere es obtener el valor del atributo **value** de la opción (**<option>**) seleccionada por el usuario. Obtener este valor no es sencillo, ya que se deben realizar una serie de pasos. Además, para obtener el valor seleccionado, deben utilizarse las siguientes propiedades:

- **`options`**, es un array creado automáticamente por el navegador para cada lista desplegable y que contiene la referencia a todas las opciones de esa lista. De esta forma, la primera opción de una lista se puede obtener mediante **`document.getElementById("id_de_la_lista").options[0].`**

- **`selectedIndex`**, cuando el usuario selecciona una opción, el navegador actualiza automáticamente el valor de esta propiedad, que guarda el índice de la opción seleccionada. El índice hace referencia al array **`options`** creado automáticamente por el navegador para cada lista.

```
// Obtener la referencia a la lista
var lista = document.getElementById("opciones");

// Obtener el índice de la opción que se ha seleccionado
var indiceSeleccionado = lista.selectedIndex;

// Con el índice y el array "options", obtener la opción seleccionada
var opcionSeleccionada = lista.options[indiceSeleccionado];

// Obtener el valor y el texto de la opción seleccionada
var textoSeleccionado = opcionSeleccionada.text;
var valorSeleccionado = opcionSeleccionada.value;

console.log("Opción seleccionada: " + textoSeleccionado + "\n Valor de
la opción: " + valorSeleccionado);
```

Como se ha visto, para obtener el valor del atributo value correspondiente a la opción seleccionada por el usuario, es necesario realizar varios pasos. No obstante, normalmente se abrevian todos los pasos necesarios en una única instrucción:

```
var lista = document.getElementById("opciones");
```

```
// Obtener el valor de la opción seleccionada
var valorSeleccionado = lista.options[lista.selectedIndex].value;

// Obtener el texto que muestra la opción seleccionada
var valorSeleccionado = lista.options[lista.selectedIndex].text;
```

Lo más importante es no confundir el valor de la propiedad selectedIndex con el valor correspondiente a la propiedad value de la opción seleccionada. En el ejemplo anterior, la primera opción tiene un value igual a 1. Sin embargo, si se selecciona esta opción, el valor de selectedIndex será 0, ya que es la primera opción del array options (y los arrays empiezan a contar los elementos en el número 0).

12.2.2. Establecer el foco de un elemento

En programación, cuando un elemento está seleccionado y se puede escribir directamente en el o se puede modificar alguna de sus propiedades, se dice que tiene el foco del programa.

Si un cuadro de texto de un formulario tiene el foco, el usuario puede escribir directamente en el sin necesidad de pinchar previamente con el ratón en el interior del cuadro. Igualmente, si una lista desplegable tiene el foco, el usuario puede seleccionar una opción directamente subiendo y bajando con las flechas del teclado.

Al pulsar repetidamente la tecla **TABULADOR** sobre una página web, los diferentes elementos (enlaces, imágenes, campos de formulario, etc.) van obteniendo el foco del navegador (el elemento seleccionado cada vez suele mostrar un pequeño borde punteado).

Si en una página web el formulario es el elemento más importante, como por ejemplo en una página de búsqueda o en una página con un formulario para registrarse, se considera una buena práctica de usabilidad el asignar automáticamente el foco al primer elemento del formulario cuando se carga la página.

Para asignar el foco a un elemento de XHTML, se utiliza la función **`focus()`**. El siguiente ejemplo asigna el foco a un elemento de formulario cuyo atributo **id** es igual a **primero**:

```
document.getElementById("primero").focus();
<form id="formulario" action="#">
  <input type="text" id="primero" />
</form>
```

Ampliando el ejemplo anterior, se puede asignar automáticamente el foco del programa al primer elemento del primer formulario de la página, independientemente del id del formulario y de los elementos:

```
if(document.forms.length > 0) {
  if(document.forms[0].elements.length > 0) {
    document.forms[0].elements[0].focus();
  }
}
```

El código anterior comprueba que existe al menos un formulario en la página mediante el tamaño del array **forms**. Si su tamaño es mayor que 0, se utiliza este primer formulario. Empleando la misma técnica, se comprueba que el formulario tenga al menos un elemento

`(if(document.forms[0].elements.length > 0))`. En caso afirmativo, se establece el foco del navegador en el primer elemento del primer formulario **`(document.forms[0].elements[0].focus();)`**.

Para que el ejemplo anterior sea completamente correcto, se debe añadir una comprobación adicional. El campo de formulario que se selecciona no debería ser de tipo **hidden**:

```
if(document.forms.length > 0) {
  for(var i=0; i < document.forms[0].elements.length; i++) {
    var campo = document.forms[0].elements[i];
    if(campo.type != "hidden") {
      campo.focus();
      break;
    }
  }
}
```

12.2.3. Evitar el envío duplicado de un formulario

Uno de los problemas habituales con el uso de formularios web es la posibilidad de que el usuario pulse dos veces seguidas sobre el botón "Enviar". Si la conexión del usuario es demasiado lenta o la respuesta del servidor se hace esperar, el formulario original sigue mostrándose en el navegador y por ese motivo, el usuario tiene la tentación de volver a pinchar sobre el botón de "Enviar".

En la mayoría de los casos, el problema no es grave e incluso es posible controlarlo en el servidor, pero puede complicarse en formularios de aplicaciones importantes como las que implican transacciones económicas.

Por este motivo, una buena práctica en el diseño de aplicaciones web suele ser la de deshabilitar el botón de envío después de la primera pulsación. El siguiente ejemplo muestra el código necesario:

```
<form id="formulario" action="#">
  ...
  <input type="button" value="Enviar" onclick="this.disabled=true;
this.value='Enviando...'; this.form.submit()" />
</form>
```

Cuando se pulsa sobre el botón de envío del formulario, se produce el evento **`onclick`** sobre el botón y, por tanto, se ejecutan las instrucciones JavaScript contenidas en el atributo **`onclick`**:

1. En primer lugar, se deshabilita el botón mediante la instrucción **`this.disabled = true;`**. Esta es la única instrucción necesaria si sólo se quiere deshabilitar un botón.
2. A continuación, se cambia el mensaje que muestra el botón. Del original "Enviar" se pasa al más adecuado "Enviando..."
3. Por último, se envía el formulario mediante la función **`submit()`** en la siguiente instrucción: **`this.form.submit()`**

El botón del ejemplo anterior está definido mediante un botón de tipo `<input type="button" />`, ya que el código JavaScript mostrado no funciona correctamente con un botón de

tipo `<input type="submit" />`. Si se utiliza un botón de tipo submit, el botón se deshabilita antes de enviar el formulario y por tanto el formulario acaba sin enviarse.

12.2.4. Limitar el tamaño de caracteres de un textarea

La carencia más importante de los campos de formulario de tipo textarea es la imposibilidad de limitar el máximo número de caracteres que se pueden introducir, de forma similar al atributo **`maxlength`** de los cuadros de texto normales.

JavaScript permite añadir esta característica de forma muy sencilla. En primer lugar, hay que recordar que con algunos eventos (como **`onkeypress, onclick y onsubmit`**) se puede evitar su comportamiento normal si se devuelve el valor false.

Evitar el comportamiento normal equivale a modificar completamente el comportamiento habitual del evento. Si por ejemplo se devuelve el valor false en el evento onkeypress, la tecla pulsada por el usuario no se tiene en cuenta. Si se devuelve false en el evento onclick de un elemento como un enlace, el navegador no carga la página indicada por el enlace.

Si un evento devuelve el valor true, su comportamiento es el habitual:

```
<textarea onkeypress="return true;"></textarea>
```

En el textarea del ejemplo anterior, el usuario puede escribir cualquier carácter, ya que el evento onkeypress devuelve true y, por tanto, su comportamiento es el normal y la tecla pulsada se transforma en un carácter dentro del textarea.

Sin embargo, en el siguiente ejemplo:

```
<textarea onkeypress="return false;"></textarea>
```

Como el valor devuelto por el evento onkeypress es igual a false, el navegador no ejecuta el comportamiento por defecto del evento, es decir, la tecla presionada no se transforma en ningún carácter dentro del textarea. No importa las veces que se pulsen las teclas y no importa la tecla pulsada, ese textarea no permitirá escribir ningún carácter.

Aprovechando esta característica, es sencillo limitar el número de caracteres que se pueden escribir en un elemento de tipo textarea: se comprueba si se ha llegado al máximo número de caracteres permitido y en caso afirmativo se evita el comportamiento habitual del evento y, por tanto, los caracteres adicionales no se añaden al textarea:

```
function limita(maximoCaracteres) {
  var elemento = document.getElementById("texto");
```

```
  if(elemento.value.length >= maximoCaracteres ) {
    return false;
  }
  else {
    return true;
  }
}
<textarea id="texto" onkeypress="return limita(100);"></textarea>
```

En el ejemplo anterior, con cada tecla pulsada se compara el número total de caracteres del textarea con el máximo número de caracteres permitido. Si el número de caracteres es igual o mayor que el límite, se devuelve el valor false y por tanto, se evita el comportamiento por defecto de onkeypress y la tecla no se añade.

12.2.5. Restringir los caracteres permitidos en un cuadro de texto

En ocasiones, puede ser útil bloquear algunos caracteres determinados en un cuadro de texto. Si por ejemplo un cuadro de texto espera que se introduzca un número, puede ser interesante no permitir al usuario introducir ningún carácter que no sea numérico.

Igualmente, en algunos casos puede ser útil impedir que el usuario introduzca números en un cuadro de texto. Utilizando el evento **`onkeypress`** y unas cuantas sentencias JavaScript, el problema se resuelve fácilmente:

```
function permite(elEvento, permitidos) {

// Variables que definen los caracteres permitidos
  var numeros = "0123456789";
  var caracteres = "
abcdefghijklmnñopqrstuvwxyzABCDEFGHIJKLMNÑOPQRSTUVWXYZ";
  var numeros_caracteres = numeros + caracteres;
  var teclas_especiales = [8, 37, 39, 46];
  // 8 = BackSpace, 46 = Supr, 37 = flecha izquierda, 39 = flecha
derecha

  // Seleccionar los caracteres a partir del parámetro de la función
  switch(permitidos) {
    case 'num':
      permitidos = numeros;
      break;
    case 'car':
      permitidos = caracteres;
      break;
    case 'num_car':
      permitidos = numeros_caracteres;
      break;
  }


  // Obtener la tecla pulsada
  var evento = elEvento || window.event;
```

```
  var codigoCaracter = evento.charCode || evento.keyCode;
  var caracter = String.fromCharCode(codigoCaracter);

  // Comprobar si la tecla pulsada es alguna de las teclas especiales
  // (teclas de borrado y flechas horizontales)
  var tecla_especial = false;
  for(var i in teclas_especiales) {
    if(codigoCaracter == teclas_especiales[i]) {
      tecla_especial = true;
      break;
    }
  }

  // Comprobar si la tecla pulsada se encuentra en los caracteres
permitidos
  // o si es una tecla especial
  return permitidos.indexOf(caracter) != -1 || tecla_especial;
}
```

```
// Sólo números
<input type="text" id="texto" onkeypress="return permite(event,
'num')" />

// Sólo letras
<input type="text" id="texto" onkeypress="return permite(event,
'car')" />

// Sólo letras o números
<input type="text" id="texto" onkeypress="return permite(event,
'num_car')" />
```

El funcionamiento del script anterior se basa en permitir o impedir el comportamiento habitual del evento **`onkeypress`**. Cuando se pulsa una tecla, se comprueba si el carácter de esa tecla se encuentra dentro de los caracteres permitidos para ese elemento **<input>**.

Si el carácter se encuentra dentro de los caracteres permitidos, se devuelve true y por tanto el comportamiento de onkeypress es el habitual y la tecla se escribe. Si el carácter no se encuentra dentro de los caracteres permitidos, se devuelve false y por tanto se impide el comportamiento normal de onkeypress y la tecla no llega a escribirse en el input.

Además, el script anterior siempre permite la pulsación de algunas teclas *especiales*. En concreto, las teclas *BackSpace y Supr* para borrar caracteres y las teclas *Flecha Izquierda y Flecha Derecha* para moverse en el cuadro de texto siempre se pueden pulsar independientemente del tipo de caracteres permitidos.

12.3. Validación

La principal utilidad de JavaScript en el manejo de los formularios es la validación de los datos introducidos por los usuarios. Antes de enviar un formulario al servidor, se recomienda validar mediante JavaScript los datos insertados por el usuario. De esta forma, si el usuario ha cometido algún error al rellenar el formulario, se le puede notificar de forma instantánea, sin necesidad de esperar la respuesta del servidor.

Notificar los errores de forma inmediata mediante JavaScript mejora la satisfacción del usuario con la aplicación (lo que técnicamente se conoce como "mejorar la experiencia de usuario") y ayuda a reducir la carga de procesamiento en el servidor.

Normalmente, la validación de un formulario consiste en llamar a una función de validación cuando el usuario pulsa sobre el botón de envío del formulario. En esta función, se comprueban si los valores que ha introducido el usuario cumplen las restricciones impuestas por la aplicación.

Aunque existen tantas posibles comprobaciones como elementos de formulario diferentes, algunas comprobaciones son muy habituales: que se rellene un campo obligatorio, que se seleccione el valor de una lista desplegable, que la dirección de email indicada sea correcta, que la fecha introducida sea lógica, que se haya introducido un número donde así se requiere, etc.

A continuación, se muestra el código JavaScript básico necesario para incorporar la validación a un formulario:

```
<form action="" method="" id="" name="" onsubmit="return
validacion()">
  ...
</form>
```

Y el esquema de la función `validacion()` es el siguiente:

```
function validacion() {
  if (condicion que debe cumplir el primer campo del formulario) {
    // Si no se cumple la condicion...
    console.log('[ERROR] El campo debe tener un valor de...');
    return false;
  }
  else if (condicion que debe cumplir el segundo campo del formulario)
{
    // Si no se cumple la condicion...
    console.log('[ERROR] El campo debe tener un valor de...');
    return false;
  }
  ...
  else if (condicion que debe cumplir el último campo del formulario)
{
    // Si no se cumple la condicion...
    console.log('[ERROR] El campo debe tener un valor de...');
    return false;
  }
   // Si el script ha llegado a este punto, todas las condiciones
  // se han cumplido, por lo que se devuelve el valor true
  return true;
}
```

El funcionamiento de esta técnica de validación se basa en el comportamiento del evento `onsubmit` de JavaScript. Al igual que otros eventos como `onclick y onkeypress`, el evento `onsubmit` varía su comportamiento en función del valor que se devuelve.

Así, si el evento `onsubmit` devuelve el valor true, el formulario se envía como lo haría normalmente. Sin embargo, si el evento `onsubmit` devuelve el valor false, el formulario no se envía. La clave de esta técnica consiste en comprobar todos y cada uno de los elementos del formulario. En cuando se encuentra un elemento incorrecto, se devuelve el valor false. Si no se encuentra ningún error, se devuelve el valor true.

Por lo tanto, en primer lugar, se define el evento `onsubmit` del formulario como:

```
onsubmit="return validacion()"
```

Como el código JavaScript devuelve el valor resultante de la función `validacion()`, el formulario solamente se enviará al servidor si esa función devuelve true. En el caso de que la función `validacion()` devuelva false, el formulario permanecerá sin enviarse.

Dentro de la función `validacion()` se comprueban todas las condiciones impuestas por la aplicación. Cuando no se cumple una condición, se devuelve false y por tanto el formulario no se envía. Si se llega al final de la función, todas las condiciones se han cumplido correctamente, por lo que se devuelve true y el formulario se envía.

La notificación de los errores cometidos depende del diseño de cada aplicación. En el código del ejemplo anterior simplemente se muestran mensajes mediante la función `console.log()` indicando el error producido. Las aplicaciones web mejor diseñadas muestran cada mensaje de error al lado del elemento de formulario correspondiente y también suelen mostrar un mensaje principal indicando que el formulario contiene errores.

Una vez definido el esquema de la función `validacion()`, se debe añadir a esta función el código correspondiente a todas las comprobaciones que se realizan sobre los elementos del formulario. A continuación, se muestran algunas de las validaciones más habituales de los campos de formulario.

12.3.1. Validar un campo de texto obligatorio

Se trata de forzar al usuario a introducir un valor en un cuadro de texto o textarea en los que sea obligatorio. La condición en JavaScript se puede indicar como:

```
valor = document.getElementById("campo").value;
if( valor == null || valor.length == 0 || /^\s+$/.test(valor) ) {
  return false;
}
```

Para que se de por completado un campo de texto obligatorio, se comprueba que el valor introducido sea válido, que el número de caracteres introducido sea mayor que cero y que no se hayan introducido sólo espacios en blanco.

La palabra reservada **null** es un valor especial que se utiliza para indicar *"ningún valor"*. Si el valor de una variable es null, la variable no contiene ningún valor de tipo objeto, array, numérico, cadena de texto o booleano.

La segunda parte de la condición obliga a que el texto introducido tenga una longitud superior a cero caracteres, esto es, que no sea un texto vacío.

Por último, la tercera parte de la condición **(/^\s+$/.test(valor))** obliga a que el valor introducido por el usuario no sólo esté formado por espacios en blanco. Esta comprobación se basa en el uso de "expresiones regulares", un recurso habitual en cualquier lenguaje de programación pero que por su gran complejidad no se van a estudiar. Por lo tanto, sólo es necesario copiar literalmente esta condición, poniendo especial cuidado en no modificar ningún carácter de la expresión.

12.3.2. Validar un campo de texto con valores numéricos

Se trata de obligar al usuario a introducir un valor numérico en un cuadro de texto. La condición JavaScript consiste en:

```
valor = document.getElementById("campo").value;
if( isNaN(valor) ) {
  return false;
}
```

Si el contenido de la variable valor no es un número válido, no se cumple la condición. La ventaja de utilizar la función interna **`isNaN()`** es que simplifica las comprobaciones, ya que JavaScript se encarga de tener en cuenta los decimales, signos, etc.

A continuación se muestran algunos resultados de la función **`isNaN():`**

```
isNaN(3);           // false
isNaN("3");         // false
isNaN(3.3545);      // false
isNaN(32323.345);   // false
isNaN(+23.2);       // false
isNaN("-23.2");     // false
isNaN("23a");       // true
isNaN("23.43.54"); // true
```

12.3.3. Validar que se ha seleccionado una opción de una lista

Se trata de obligar al usuario a seleccionar un elemento de una lista desplegable. El siguiente código JavaScript permite conseguirlo:

```
indice = document.getElementById("opciones").selectedIndex;
if( indice == null || indice == 0 ) {
  return false;
}
<select id="opciones" name="opciones">
  <option value="">- Selecciona un valor -</option>
  <option value="1">Primer valor</option>
  <option value="2">Segundo valor</option>
  <option value="3">Tercer valor</option>
</select>
```

A partir de la propiedad **`selectedIndex`**, se comprueba si el índice de la opción seleccionada es válido y además es distinto de cero. La primera opción de la lista (- Selecciona un valor –) no es válida, por lo que no se permite el valor 0 para esta propiedad selectedIndex.

12.3.4. Validar una dirección de email

Se trata de obligar al usuario a introducir una dirección de email con un formato válido. Por tanto, lo que se comprueba es que la dirección parezca válida, ya que no se comprueba si se trata de una cuenta de correo electrónico real y operativa. La condición JavaScript consiste en:

```
valor = document.getElementById("campo").value;
if( !(/\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)/.test(valor)) ) {
  return false;
}
```

La comprobación se realiza nuevamente mediante las expresiones regulares, ya que las direcciones de correo electrónico válidas pueden ser muy diferentes. Por otra parte, como el estándar que define el formato de las direcciones de correo electrónico es muy complejo, la expresión regular anterior es una simplificación. Aunque esta regla valida la mayoría de direcciones de correo electrónico utilizadas por los usuarios, no soporta todos los diferentes formatos válidos de email.

12.3.5. Validar una fecha

Las fechas suelen ser los campos de formulario más complicados de validar por la multitud de formas diferentes en las que se pueden introducir. El siguiente código asume que de alguna forma se ha obtenido el año, el mes y el día introducidos por el usuario:

```
var ano = document.getElementById("ano").value;
var mes = document.getElementById("mes").value;
var dia = document.getElementById("dia").value;

valor = new Date(ano, mes, dia);

if( !isNaN(valor) ) {
  return false;
}
```

La función **`Date(ano, mes, dia)`** es una función interna de JavaScript que permite construir fechas a partir del año, el mes y el día de la fecha. Es muy importante tener en cuenta que el número de mes se indica de 0 a 11, siendo 0 el mes de Enero y 11 el mes de Diciembre. Los días del mes siguen una numeración diferente, ya que el mínimo permitido es 1 y el máximo 31.

La validación consiste en intentar construir una fecha con los datos proporcionados por el usuario. Si los datos del usuario no son correctos, la fecha no se puede construir correctamente y por tanto la validación del formulario no será correcta.

12.3.6. Validar un número de DNI

Se trata de comprobar que el número proporcionado por el usuario se corresponde con un número válido de Documento Nacional de Identidad o DNI. Aunque para cada país o región los requisitos del documento de identidad de las personas pueden variar, a continuación, se muestra un ejemplo genérico fácilmente adaptable. La validación no sólo debe comprobar que el número esté formado por ocho cifras y una letra, sino que también es necesario comprobar que la letra indicada es correcta para el número introducido:

```
valor = document.getElementById("campo").value;
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X',
'B', 'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];

if( !(/^\d{8}[A-Z]$/.test(valor)) ) {
  return false;
}

if(valor.charAt(8) != letras[(valor.substring(0, 8))%23]) {
  return false;
}
```

La primera comprobación asegura que el formato del número introducido es el correcto, es decir, que está formado por 8 números seguidos y una letra. Si la letra está al principio de los números, la comprobación sería **/^[A-Z]\d{8}$/**. Si en vez de ocho números y una letra, se requieren diez números y dos letras, la comprobación sería **/^\d{10}[A-Z]{2}$/** y así sucesivamente.

La segunda comprobación aplica el algoritmo de cálculo de la letra del DNI y la compara con la letra proporcionada por el usuario. El algoritmo de cada documento de identificación es diferente, por lo que esta parte de la validación se debe adaptar convenientemente.

12.3.7. Validar un número de teléfono

Los números de teléfono pueden ser indicados de formas muy diferentes: con prefijo nacional, con prefijo internacional, agrupado por pares, separando los números con guiones, etc.

El siguiente script considera que un número de teléfono está formado por nueve dígitos consecutivos y sin espacios ni guiones entre las cifras:

```
valor = document.getElementById("campo").value;
if( !(/^\d{9}$/.test(valor)) ) {
  return false;
}
```

Una vez más, la condición de JavaScript se basa en el uso de expresiones regulares, que comprueban si el valor indicado es una sucesión de nueve números consecutivos. A continuación, se muestran otras expresiones regulares que se pueden utilizar para otros formatos de número de teléfono:

Número	Expresión regular	Formato
900900900	/^\d{9}$/	9 cifras seguidas
900-900-900	/^\d{3}-\d{3}-\d{3}$/	9 cifras agrupadas de 3 en 3 y separadas por guiones
900 900900	/^\d{3}\s\d{6}$/	9 cifras, las 3 primeras separadas por un espacio
900 90 09 00	/^\d{3}\s\d{2}\s\d{2}\s\d{2}$/	9 cifras, las 3 primeras separadas por un espacio, las siguientes agrupadas de 2 en 2
(900) 900900	/^\(\d{3}\)\s\d{6}$/	9 cifras, las 3 primeras encerradas por paréntesis y un espacio de separación respecto del resto
+34 900900900	/^\+\d{2,3}\s\d{9}$/	Prefijo internacional (+ seguido de 2 o 3 cifras), espacio en blanco y 9 cifras consecutivas

12.3.8. Validar que un checkbox ha sido seleccionado

Si un elemento de tipo *checkbox* se debe seleccionar de forma obligatoria, JavaScript permite comprobarlo de forma muy sencilla:

```
elemento = document.getElementById("campo");
if( !elemento.checked ) {
  return false;
}
```

Si se trata de comprobar que todos los checkbox del formulario han sido seleccionados, es más fácil utilizar un bucle:

```
formulario = document.getElementById("formulario");
for(var i=0; i<formulario.elements.length; i++) {
  var elemento = formulario.elements[i];
  if(elemento.type == "checkbox") {
    if(!elemento.checked) {
      return false;
    }
  }
}
```

12.3.9. Validar que un radiobutton ha sido seleccionado

Aunque se trata de un caso similar al de los *checkbox*, la validación de los *radiobutton* presenta una diferencia importante: en general, la comprobación que se realiza es que el usuario haya seleccionado algún *radiobutton* de los que forman un determinado grupo. Mediante JavaScript, es sencillo determinar si se ha seleccionado algún *radiobutton* de un grupo:

```
opciones = document.getElementsByName("opciones");

var seleccionado = false;
for(var i=0; i<opciones.length; i++) {
  if(opciones[i].checked) {
    seleccionado = true;
    break;
  }
}

if(!seleccionado) {
  return false;
}
```

El anterior ejemplo recorre todos los *radiobutton* que forman un grupo y comprueba elemento por elemento si ha sido seleccionado. Cuando se encuentra el primer *radiobutton* seleccionado, se sale del bucle y se indica que al menos uno ha sido seleccionado.

13. JQUERY

13.1. Introducción

jQuery es una biblioteca de JavaScript ligera. El propósito de jQuery es facilitar el uso de JavaScript en su sitio web.

jQuery toma muchas tareas comunes que requieren muchas líneas de código JavaScript para cumplirlas, y las envuelve en métodos a los que puedes llamar con una sola línea de código.

jQuery también simplifica muchas de las cosas complicadas de JavaScript, como las llamadas AJAX y la manipulación de DOM.

La biblioteca jQuery contiene las siguientes características:

- Manipulación HTML / DOM
- Manipulación de CSS
- Métodos de eventos HTML
- Efectos y animaciones.
- AJAX
- Utilidades

13.2. Agregar jQuery a páginas web

Hay varias formas de comenzar a usar jQuery en su sitio web:

1. **Descargando la biblioteca jQuery**
   Hay dos versiones de jQuery disponibles para descargar:
   - Versión de producción: esto es para su sitio web en vivo porque ha sido minimizado y comprimido
   - Versión de desarrollo: para pruebas y desarrollo (código no comprimido y legible)

   Ambas versiones se pueden descargar desde jQuery.com .

   La biblioteca jQuery es un único archivo JavaScript, y se hace referencia a él con la etiqueta `<script>` (observa que la etiqueta `<script>` debe estar dentro de la sección `<head>`):

```
<head>
<script                          src="jquery-3.3.1.min.js"></script>
</head>
```

   Consejo: coloca el archivo descargado en el mismo directorio que las páginas donde desea usarlo.

2. **Incluyendo jQuery desde un CDN (Content Delivery Network), como Google**
   Tanto Google como Microsoft alojan jQuery. Para usar jQuery de Google o Microsoft, usa uno de los siguientes:

Google CDN:

```
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
</head>
```

Microsoft CDN:

```
<head>
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.min.js"></script>
</head>
```

13.3. Sintaxis

La sintaxis de jQuery está hecha a medida para **seleccionar** elementos HTML y realizar alguna **acción** en los elementos.

La sintaxis básica es: **$ (*selector*). *acción* ()**

- Un signo $ para definir / acceder a jQuery
- Un (selector) para "consultar (o encontrar)" elementos HTML
- Una *acción* jQuery () que se realizará en el elemento (s)

Ejemplos:
`$(this).hide()` - Oculta el elemento actual.
`$("p").hide()` - Oculta todos los elementos <p>.
`$(".test").hide()` - Oculta todos los elementos con class = "test".
`$("#test").hide()` - Oculta el elemento con id = "test".

Todos los métodos de jQuery están dentro de un evento preparado para documentos:

```
$(document).ready(function(){
  // jQuery methods go here...
});
```

Esto es para evitar que se ejecute cualquier código de jQuery antes de que el documento termine de cargarse (esté listo).

13.4. Selectores

Los selectores jQuery permiten seleccionar y manipular elementos HTML.

Los selectores jQuery se utilizan para "buscar" (o seleccionar) elementos HTML según su nombre, id, clases, tipos, atributos, valores de atributos y mucho más. Se basa en los selectores de CSS existentes y, además, tiene algunos selectores personalizados.

Todos los selectores en jQuery comienzan con el signo de dólar y los paréntesis: **$ ()**.

1. Selector de elementos

El selector de elementos jQuery selecciona elementos en función del nombre del elemento.

Puedes seleccionar todos los elementos <p> en una página como esta:

```
$("p")
```

Ejemplo: Cuando un usuario hace clic en un botón, todos los elementos <p> se ocultarán:

```
$(document).ready(function(){
  $("button").click(function(){
    $("p").hide();
  });
});
```

2. Selector #id

El selector **#id** de jQuery utiliza el atributo id de una etiqueta HTML para encontrar el elemento específico**.**

Un id debe ser único dentro de una página, por lo que debe usar el selector #id cuando desee encontrar un elemento único y único.

Para encontrar un elemento con un id específico, escriba un carácter de hash, seguido de la identificación del elemento HTML:

```
$("#test")
```

Ejemplo: Cuando un usuario hace clic en un botón, el elemento con `id = "prueba"` se ocultará:

```
$(document).ready(function(){
  $("button").click(function(){
    $("#test").hide();
  });
});
```

3. Selector de clase.

El selector **`class`** de jQuery encuentra elementos con una clase específica.

Para buscar elementos con una clase específica, escriba un carácter de punto, seguido del nombre de la clase:

```
$(".test")
```

Ejemplo: Cuando un usuario hace clic en un botón, los elementos con `class = "test"` se ocultarán:

```
$(document).ready(function(){
  $("button").click(function(){
    $(".test").hide();
  });
});
```

4. Más ejemplos de selectores jQuery

Syntax	Description
$("*")	Selects all elements
$(this)	Selects the current HTML element
$("p.intro")	Selects all <p> elements with class="intro"
$("p:first")	Selects the first <p> element
$("ul li:first")	Selects the first element of the first
$("ul li:first-child")	Selects the first element of every
$("[href]")	Selects all elements with an href attribute
$("a[target='_blank']")	Selects all <a> elements with a target attribute value equal to "_blank"
$("a[target!='_blank']")	Selects all <a> elements with a target attribute value NOT equal to "_blank"
$(":button")	Selects all <button> elements and <input> elements of type="button"
$("tr:even")	Selects all even <tr> elements
$("tr:odd")	Selects all odd <tr> elements

13.5. Eventos

Las diferentes acciones de los usuarios a las que puede responder una página web se denominan **eventos**.

Un evento representa el momento preciso en que algo sucede.

Ejemplos:

- moviendo un mouse sobre un elemento
- seleccionando un botón de radio
- haciendo clic en un elemento

Algunos eventos comunes de DOM:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

1. Sintaxis de jQuery para métodos de eventos

En jQuery, la mayoría de los eventos DOM tienen un método jQuery equivalente.

Para asignar un evento de clic a todos los párrafos de una página, puede hacer esto:

```
$("p").click();
```

El siguiente paso es definir qué debe suceder cuando se dispara el evento. Debes pasar una función al evento:

```
$("p").click(function(){
  // action goes here!!
});
```

2. Métodos de eventos jQuery de uso común

- **`$ (documento).ready()`**

El método **`$(document).ready()`** permite ejecutar una función cuando el documento está completamente cargado.

- **`click()`**

El método **`click()`** adjunta una función de controlador de eventos a un elemento HTML. La función se ejecuta cuando el usuario hace clic en el elemento HTML.

Ejemplo: Cuando un evento de clic se activa en un elemento **<p>**; ocultar el elemento actual **<p>**:

```
$("p").click(function(){
                                        $(this).hide();
});
```

- **dblclick()**

El método **dblclick()** adjunta una función de controlador de eventos a un elemento HTML. La función se ejecuta cuando el usuario hace doble clic en el elemento HTML:

```
$("p").dblclick(function(){
  $(this).hide();
});
```

- **mouseenter()**

El método **mouseenter()** adjunta una función de controlador de eventos a un elemento HTML. La función se ejecuta cuando el puntero del mouse entra en el elemento HTML:

```
$("#p1").mouseenter(function(){
  alert("You entered p1!");
});
```

- **mouseleave()**

El método **mouseleave()** adjunta una función de controlador de eventos a un elemento HTML. La función se ejecuta cuando el puntero del mouse deja el elemento HTML:

```
$("#p1").mouseleave(function(){
  alert("Bye! You now leave p1!");
});
```

- **mousedown()**

El método **mousedown()** adjunta una función de controlador de eventos a un elemento HTML. La función se ejecuta cuando se presiona el botón izquierdo, central o derecho del mouse, mientras que el mouse se encuentra sobre el elemento HTML:

```
$("#p1").mousedown(function(){
  alert("Mouse down over p1!");
});
```

- **mouseup()**

El método **mouseup()** adjunta una función de controlador de eventos a un elemento HTML. La función se ejecuta cuando se suelta el botón izquierdo, central o derecho del mouse, mientras que el mouse se encuentra sobre el elemento HTML:

```
$("#p1").mouseup(function(){
  alert("Mouse up over p1!");
});
```

- **hover()**

El método **hover()** tiene dos funciones y es una combinación de los métodos `mouseenter()` **y** `mouseleave()`.
La primera función se ejecuta cuando el mouse ingresa al elemento HTML, y la segunda función se ejecuta cuando el mouse deja el elemento HTML:

```
$("#p1").hover(function(){
  alert("You entered p1!");
},
function(){
  alert("Bye! You now leave p1!");
});
```

- **focus()**

El método **focus()** adjunta una función de controlador de eventos a un campo de formulario HTML. La función se ejecuta cuando el campo de formulario se enfoca:

```
$("input").focus(function(){
  $(this).css("background-color", "#cccccc");
});
```

- **blur()**

El método **blur()** adjunta una función de controlador de eventos a un campo de formulario HTML. La función se ejecuta cuando el campo de formulario pierde el foco:

```
$("input").blur(function(){
  $(this).css("background-color", "#ffffff");
});
```

3. El método on ()

El método **on()** adjunta uno o más controladores de eventos para los elementos seleccionados.

Ejemplo: Adjuntar un evento de clic a un elemento **<p>**:

```
$("p").on("click", function(){
  $(this).hide();
});
```

13.6. Manipulación del DOM

jQuery viene con un montón de métodos relacionados con DOM que facilitan el acceso y la manipulación de elementos y atributos.

DOM = Modelo de Objeto de Documento

El DOM define un estándar para acceder a documentos HTML y XML:

"El Modelo de Objeto de Documento (DOM) de W3C es una plataforma e interfaz de lenguaje neutral que permite a los programas y scripts acceder y actualizar dinámicamente el contenido, la estructura y estilo de un documento ".

13.6.1. Obtener contenido

Para la manipulación de DOM existen tres métodos simples, pero útiles, de jQuery:

- **`text()`** - Establece o devuelve el contenido de texto de los elementos seleccionados.
- **`html()`** - Establece o devuelve el contenido de los elementos seleccionados (incluido el marcado HTML)
- **`val()`** - Establece o devuelve el valor de los campos de formulario.

Ejemplo: Cómo obtener contenido con los métodos **`text() y html()`:**

```
$("#btn1").click(function(){
  alert("Text: " + $("#test").text());
});
$("#btn2").click(function(){
  alert("HTML: " + $("#test").html());
});
```

Ejemplo: Cómo obtener el valor de un campo de entrada con el método **`val()`:**

```
$("#btn1").click(function(){
  alert("Value: " + $("#test").val());
});
```

- **Obtener atributos - `attr()`**

El método **`attr()`** de jQuery se utiliza para obtener valores de atributos.

Ejemplo: Cómo obtener el valor del atributo href en un enlace:

```
$("button").click(function(){
                                          alert($("#w3s").attr("href"));
});
```

13.6.2. Establecer contenido

Para **configurar el contenido**, usaremos los mismos tres métodos anteriores:

- **`text()`** - Establece o devuelve el contenido de texto de los elementos seleccionados.
- **`html()`** - Establece o devuelve el contenido de los elementos seleccionados (incluido el marcado HTML)
- **`val()`** - Establece o devuelve el valor de los campos de formulario.

Ejemplo: muestra cómo establecer el contenido con los métodos **`text(), html() y val()`:**

```
$("#btn1").click(function(){
  $("#test1").text("Hello world!");
});
$("#btn2").click(function(){
  $("#test2").html("<b>Hello world!</b>");
});
$("#btn3").click(function(){
  $("#test3").val("Dolly Duck");
});
```

➢ **Función de devolución de llamada para `text(), html() y val()`**

Los tres métodos anteriores: `text(), html() y val(),` también vienen con una función de devolución de llamada. La función de devolución de llamada tiene dos parámetros: el índice del elemento actual en la lista de elementos seleccionados y el valor original (antiguo). A continuación, devuelve la cadena que desea utilizar como el nuevo valor de la función.

Ejemplo: demuestra **`text()`** `y` **`html()`** con una función de devolución de llamada:

```
$("#btn1").click(function(){
  $("#test1").text(function(i, origText){
    return "Old text: " + origText + " New text: Hello world!
    (index: " + i + ")";
  });
});

$("#btn2").click(function(){
  $("#test2").html(function(i, origText){
    return "Old html: " + origText + " New html: Hello <b>world!</b>
    (index: " + i + ")";
  });
});
```

➢ **Establecer atributos - `attr()`**

El método **`attr()`** también se usa para establecer / cambiar valores de atributos.

Ejemplo: muestra cómo cambiar (configurar) el valor del atributo href en un enlace:

```
$("button").click(function(){
  $("#w3s").attr("href", "https://www.w3schools.com/jquery/");
});
```

El método **attr()** también permite establecer múltiples atributos al mismo tiempo.

Ejemplo: muestra cómo configurar los atributos `href y title` al mismo tiempo:

```
$("button").click(function(){
  $("#w3s").attr({
    "href" : "https://www.w3schools.com/jquery/",
    "title" : "W3Schools jQuery Tutorial"
  });
});
```

- **Función de devolución de llamada para attr()**

El método **attr()**, también viene con una función de devolución de llamada. La función de devolución de llamada tiene dos parámetros: el índice del elemento actual en la lista de elementos seleccionados y el valor del atributo original (antiguo). A continuación, devuelve la cadena que desea utilizar como el nuevo valor de atributo de la función.

Ejemplo: demuestra **attr()** con una función de devolución de llamada:

```
$("button").click(function(){
  $("#w3s").attr("href", function(i, origValue){
    return origValue + "/jquery/";
  });
});
```

13.6.3. Agregar contenido HTML

Los métodos de jQuery que se utilizan para agregar contenido nuevo son:

- **append()** - Inserta contenido al final de los elementos seleccionados.
- **prepend()** - Inserta contenido al principio de los elementos seleccionados.
- **after()** - Inserta el contenido después de los elementos seleccionados.
- **before()** - Inserta contenido antes de los elementos seleccionados.

- **Método append()**

El método **append()** inserta contenido AL FINAL de los elementos HTML seleccionados.

```
$("p").append("Some appended text.");
```

- **Método `prepend()`**

El método **`prepend()`** inserta contenido AL COMIENZO de los elementos HTML seleccionados.

```
$("p").prepend("Some prepended text.");
```

- **Añadir varios elementos nuevos con `append()` y `prepend()`**

Tanto el método **`append()`** como **`prepend()`** pueden tomar un número infinito de nuevos elementos como parámetros. Los nuevos elementos se pueden generar con texto / HTML, con jQuery o con código JavaScript y elementos DOM.

En el siguiente ejemplo, creamos varios elementos nuevos. Los elementos se crean con texto / HTML, jQuery y JavaScript / DOM. Luego agregamos los nuevos elementos al texto con el método **`append()`** (esto también habría funcionado con **`prepend()`**):

```
function appendText() {
  var txt1 = "<p>Text.</p>";               // Create element with
HTML
  var txt2 = $("<p></p>").text("Text.");   // Create with jQuery
  var txt3 = document.createElement("p");  // Create with DOM
  txt3.innerHTML = "Text.";
  $("body").append(txt1, txt2, txt3);      // Append the new elements
}
```

- **Métodos `after()` y `before()`**

El método **`after()`** inserta contenido DESPUÉS de los elementos HTML seleccionados.

El método **`before()`** inserta contenido ANTES de los elementos HTML seleccionados.

```
$("img").after("Some text after");

$("img").before("Some text before");
```

- **Añadir varios elementos nuevos con `after()` y `before()`**

Tanto el método **`after()`** como **`before()`** pueden tomar un número infinito de nuevos elementos como parámetros. Los nuevos elementos se pueden generar con texto / HTML (como hemos hecho en el ejemplo anterior), con jQuery o con código JavaScript y elementos DOM.

En el siguiente ejemplo, creamos varios elementos nuevos. Los elementos se crean con texto / HTML, jQuery y JavaScript / DOM. Luego insertamos los nuevos elementos en el texto con el método **`after()`** (esto también habría funcionado con **`before()`**):

```
function afterText() {
  var txt1 = "<b>I </b>";                  // Create element with HTML
  var txt2 = $("<i></i>").text("love "); // Create with jQuery
  var txt3 = document.createElement("b");// Create with DOM
  txt3.innerHTML = "jQuery!";
  $("img").after(txt1, txt2, txt3); // Insert new elements after <img>
}
```

13.6.4. Eliminar contenido HTML

Para eliminar elementos y contenido, existen principalmente dos métodos de jQuery:

- **`remove()`** - Elimina el elemento seleccionado (y sus elementos secundarios)
- **`empty()`** - Elimina los elementos secundarios del elemento seleccionado.

➢ **Método `remove()`**

El método **`remove()`** elimina los elementos seleccionados y sus elementos secundarios.

```
$("#div1").remove();
```

➢ **Método `empty()`**

El método **`empty()`** elimina los elementos secundarios de los elementos seleccionados.

```
$("#div1").empty();
```

➢ **Filtrar los elementos a eliminar**

El método **`remove()`** también acepta un parámetro, que le permite filtrar los elementos que se eliminarán. El parámetro puede ser cualquiera de las sintaxis del selector jQuery.

El siguiente ejemplo elimina todos los elementos **<p>** con **class="test":**

```
$("p").remove(".test");
```

Este ejemplo elimina todos los elementos **<p>** con **class="test"**o **class="demo":**

```
$("p").remove(".test, .demo");
```

13.6.5. Manipulación de CSS

jQuery tiene varios métodos para la manipulación de CSS. Veremos los siguientes métodos:

- **`addClass()`** - Agrega una o más clases a los elementos seleccionados.
- **`removeClass()`** - Elimina una o más clases de los elementos seleccionados
- **`toggleClass()`** - Alterna entre agregar / eliminar clases de los elementos seleccionados
- **`css()`** - Establece o devuelve el atributo de estilo.

La siguiente hoja de estilo se utilizará para todos los ejemplos:

```
.important {
  font-weight: bold;
  font-size: xx-large;
}

.blue {
  color: blue;
}
```

➢ **Método `addClass()`**

El siguiente ejemplo muestra cómo **agregar atributos de clase a diferentes elementos**. Por supuesto que puedes seleccionar múltiples elementos, al agregar clases:

```
$("button").click(function(){
  $("h1, h2, p").addClass("blue");
  $("div").addClass("important");
});
```

También puede especificar múltiples clases dentro del método **`addClass()`:**

```
$("button").click(function(){
  $("#div1").addClass("important blue");
});
```

➢ **Método `removeClass()`**

El siguiente ejemplo muestra cómo **eliminar un atributo de clase específico de diferentes elementos**:

```
$("button").click(function(){
  $("h1, h2, p").removeClass("blue");
});
```

➢ Método `toggleClass()`

Este método alterna entre agregar / eliminar clases de los elementos seleccionados:

```
$("button").click(function(){
  $("h1, h2, p").toggleClass("blue");
});
```

➢ Método `css()`

El método **`css()`** establece o devuelve una o más propiedades de estilo para los elementos seleccionados.

- ***Devolver una propiedad CSS***

Para devolver el valor de una propiedad CSS especificada, se usa la siguiente sintaxis:

```
css("propertyname");
```

El siguiente ejemplo devolverá el valor de color de fondo del elemento coincidente FIRST:

```
$("p").css("background-color");
```

- ***Establecer una propiedad CSS***

Para establecer una propiedad CSS especificada, se usa la siguiente sintaxis:

```
css("propertyname","value");
```

El siguiente ejemplo establecerá el valor de color de fondo para TODOS los elementos coincidentes:

```
$("p").css("background-color", "yellow");
```

- ***Establecer múltiples propiedades CSS***

Para establecer varias propiedades CSS, se usa la siguiente sintaxis:

```
css({"propertyname":"value","propertyname":"value",...});
```

El siguiente ejemplo establecerá un color de fondo y un tamaño de fuente para TODOS los elementos coincidentes:

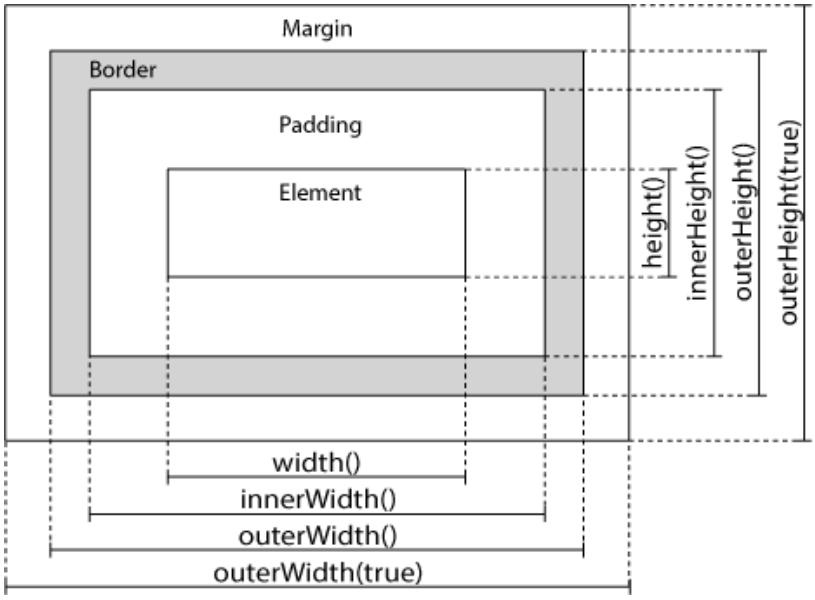
```
$("p").css({"background-color": "yellow", "font-size": "200%"});
```

13.6.6. Métodos para trabajar con dimensiones

jQuery tiene varios métodos importantes para trabajar con dimensiones:

- `width()`
- `height()`
- `innerWidth()`
- `innerHeight()`
- `outerWidth()`
- `outerHeight()`

➢ **Dimensiones de jQuery**



➢ **Métodos `width()` y `height()`**

El método **`width()`** establece o devuelve el ancho de un elemento (excluye el relleno, el borde y el margen).

El método **`height()`** establece o devuelve la altura de un elemento (excluye el relleno, el borde y el margen).

El siguiente ejemplo devuelve el ancho y la altura de un elemento ** ** especificado:

```
$("button").click(function(){
  var txt = "";
  txt += "Width: " + $("#div1").width() + "</br>";
  txt += "Height: " + $("#div1").height();
  $("#div1").html(txt);
});
```

- **Métodos `innerWidth()` y `innerHeight()`**

El método **`innerWidth()`** devuelve el ancho de un elemento (incluye el relleno).

El método **`innerHeight()`** devuelve la altura de un elemento (incluye el relleno).

El siguiente ejemplo devuelve el ancho / alto interno de un elemento ** ** especificado:

```
$("button").click(function(){
  var txt = "";
  txt += "Inner width: " + $("#div1").innerWidth() + "</br>";
  txt += "Inner height: " + $("#div1").innerHeight();
  $("#div1").html(txt);
});
```

- **Métodos `outerWidth()` y `outerHeight()`**

El método **`outerWidth()`** devuelve el ancho de un elemento (incluye el relleno y el borde).

El método **`outerHeight()`** devuelve la altura de un elemento (incluye el relleno y el borde).

El siguiente ejemplo devuelve el ancho / alto externo de un elemento ** ** especificado:

```
$("button").click(function(){
  var txt = "";
  txt += "Outer width: " + $("#div1").outerWidth() + "</br>";
  txt += "Outer height: " + $("#div1").outerHeight();
  $("#div1").html(txt);
});
```

El método **`outerWidth(true)`** devuelve el ancho de un elemento (incluye relleno, borde y margen).

El método **`outerHeight(true)`** devuelve la altura de un elemento (incluye relleno, borde y margen).

```
$("button").click(function(){
  var txt = "";
  txt += "Outer width (+margin): " + $("#div1").outerWidth(true)
+ "</br>";
  txt += "Outer height (+margin): " + $("#div1").outerHeight(true);
  $("#div1").html(txt);
});
```

- **Otras opciones de `width()` y `height()`**

El siguiente ejemplo devuelve el ancho y el alto del documento (el documento HTML) y la ventana (la ventana gráfica del navegador):

```
$("button").click(function(){
  var txt = "";
  txt += "Document width/height: " + $(document).width();
  txt += "x" + $(document).height() + "\n";
  txt += "Window width/height: " + $(window).width();
  txt += "x" + $(window).height();
  alert(txt);
});
```

El siguiente ejemplo establece el ancho y la altura de un elemento ** ** especificado :

```
$("button").click(function(){
  $("#div1").width(500).height(500);
});
```

13.7. Métodos de filtrado

Los métodos de filtrado más básicos son **`first()`, `last()`** y **`eq()`**, que permiten seleccionar un elemento específico según su posición en un grupo de elementos.

Otros métodos de filtrado, como **`filter()`** y **`not()`** permiten seleccionar elementos que coinciden, o no coinciden, con un cierto criterio.

- **Método `first()`**

El método **`first()`** devuelve el primer elemento de los elementos especificados.

El siguiente ejemplo selecciona el primer elemento ** **:

```
$(document).ready(function(){
                                                $("div").first();
});
```

- **Método `last()`**

El método **`last()`** devuelve el último elemento de los elementos especificados.

El siguiente ejemplo selecciona el último elemento ** **:

```
$(document).ready(function(){
                                                $("div").last();
});
```

- **Método `eq()`**

El método **`eq()`** devuelve un elemento con un número de índice específico de los elementos seleccionados.

Los números de índice comienzan en 0, por lo que el primer elemento tendrá el número de índice 0 y no 1. El siguiente ejemplo selecciona el segundo elemento <p> (número de índice 1):

```
$(document).ready(function(){
  $("p").eq(1);
});
```

- **Método `filter()`**

El método **`filter()`** permite especificar un criterio. Los elementos que no coinciden con los criterios se eliminan de la selección y se devolverán los que coincidan.

El siguiente ejemplo devuelve todos los elementos **<p>** con el nombre de clase "intro":

```
$(document).ready(function(){
  $("p").filter(".intro");
});
```

- **Método `not()`**

El método **`not()`** devuelve todos los elementos que no coinciden con los criterios.

El método `not()` método es el opuesto a **`filter().`**

El siguiente ejemplo devuelve todos los elementos **<p>** que no tienen el nombre de clase "intro":

```
$(document).ready(function(){
                                                $("p").not(".intro");
});
```