Lenguaje de Programación

JavaScript

INIDICE

1. Introducción.

Vamos a profundizar en el lenguaje de scripting más utilizado en la actualidad, en el desarrollo de aplicaciones web en lado del cliente, JavaScript.

El lenguaje que vas 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.

2. Variables. Ámbitos de utilización.

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.

2.1. Ámbito de las variables.

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.

3. 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>
```

4. 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.

5. Literales.

Los literales se utilizan para representar valores en JavaScript. Estos son valores constantes, no variables. Nos podemos encontrar con los siguientes tipos de literales en JavaScript:

- Arrays de literales
- Literales lógicos [booleans]
- Literales de punto flotante
- Literales enteros
- Objetos literales
- Cadenas literales (string)

5.1. Arrays de literales.

Un array literal es una lista de cero o más expresiones, cada una de las cuales representa un elemento del array, delimitada por corchetes ([]). Cuando crea un array usando un array literal, aquel será inicializado con los valores especificados como sus elementos y su longitud será el número de argumentos especificados.

El siguiente ejemplo crea un arrays de cafés con tres elementos y una longitud de tres:

```
cafés = ["Tueste francés", "Colombiano", "Kona"]
```

Array de tamaño 10

5.2. Literales lógicos booleans.

El tipo Boolean posee dos valores literales: true y false.

No debe confundirse los valores primitivos Boolean true y false con los valores verdadero [true] y falso [false] del objeto Boolean.

X	Y	And
V	V	V
V	F	F
F	V	F
F	F	F

X	Y	OR
V	V	V
V	F	V
F	V	V
F	F	F

X	NOT (X)
V	F
F	V

Tablas de verdad para los literales booleanos

5.3. Literales enteros

Los enteros pueden ser expresados en decimales (base 10), hexadecimales (base 16) y octales (base 8). Un literal entero decimal consiste de una secuencia de dígitos sin un 0 (cero) por delante. Un 0 (cero) por delante en un literal entero indica que este es un octal; Un 0x (o 0X) por delante indica que es un hexadecimal. Un entero hexadecimal puede incluir los dígitos (0-9) y las letras a-f y A-F. Los enteros octales pueden incluir solamente los dígitos 0-7.

Los literales enteros octales están obsoletos y han sido removidos del estándar de la ECMA-262, Edición 3. JavaScript 1.5 mantiene un soporte para ellos sólo por compatibilidad con versiones anteriores.

Algunos ejemplos de literales enteros son:

```
0, 117 y -345 (decimales, base 10)
015, 0001 y -077 (octales, base 8)
0x1123, 0x00111 y -0xF1A7 (hexadecimales, "hex" o base 16)
```

5.4. Literales de coma flotante.

Un literal de coma flotante puede contener las siguientes partes:

- Un entero decimal el cual puede tener signo (precedido por "+" o "-"),
- Un punto decimal ("."),
- Una fracción (otro número decimal),
- Un exponente.

La parte exponencial es una "e" o "E" seguida de un entero, el cual puede tener signo (precedido por "+" o "-"). Un literal de coma flotante debe tener al menos un dígito y, o un punto decimal o una "e" (o "E").

Algunos ejemplos de literales de coma flotante son: 3.1415, -3.1E12, .1e12 y 2E-12.

De forma breve, la sintaxis es: [dígitos][.dígitos][(E|e)[(+|-)]dígitos]

Por ejemplo:

```
3.14 2345.789 .3333333333333333333
```

5.5. Literales de objetos.

Un literal de objeto es una lista de cero o más pares de nombres de propiedades y valores asociados a dicho objeto, delimitada por llaves ({}). Usted no debe utilizar un literal de objeto al inicio de una sentencia. Esto provocaría un error o un comportamiento inesperado, debido a que una {será interpretada como el comienzo de un bloque.

El siguiente es un ejemplo de un literal de objeto. El primer elemento del objeto automovil define una propiedad, miAuto; el segundo elemento, la propiedad obtenerAuto invoca a la función (TipoAuto("Honda")); el tercer elemento, la propiedad especial usa una variable existente (Ventas).

```
var Ventas = "Toyota";

function TipoAuto(nombre)
{
  if(nombre == "Honda")
    return nombre;
  else
    return "Lo siento, no vendemos " + nombre + ".";
}
```

```
automovil = {miAuto: "Saturn", obtenerAuto: TipoAuto("Honda"),
especial: Ventas}

document.write(automovil.miAuto); // Saturn
document.write(automovil.obtenerAuto); // Honda
document.write(automovil.especial); // Toyota
```

Adicionalmente, usted puede usar literales de cadena o numéricos para el nombre de una propiedad o para anidar un objeto dentro de otro. El siguiente ejemplo utiliza estas opciones.

```
car = {manyCars: {a: "Saab", b: "Jeep"}, 7: "Mazda"}

document.write(car.manyCars.b); // Jeep
document.write(car[7]); // Mazda
```

5.6. Literales de cadenas de caracteres (string).

Una cadena literal tiene cero o más caracteres encerrados entre comillas dobles (") o simples ('). Una cadena debe estar delimitada por comillas del mismo tipo, es decir, tanto las comillas simples como dobles al inicio y al final de la cadena. Los siguientes ejemplos son cadenas literales:

```
"blah"
'blah'
"1234"
"una línea \n otra línea"
"El gato de Juan"
```

Se puede invocar cualquiera de los métodos del objeto **String** (cadena) en un valor literal de cadena—JavaScript automáticamente la convierte de una cadena literal a un objeto **String temporal**, mediante la invocación del método, luego descarta el objeto **String** temporal. También se puede usar la propiedad **String.length** con una cadena literal:

```
"El gato de Juan".length;
```

5.7. Uso de caracteres especiales en cadenas de caracteres.

Además de los caracteres ordinarios, se puede también incluir caracteres especiales en las cadenas, tal como se muestra en el siguiente ejemplo.

```
"una línea \n otra línea"
```

Tabla. Caracteres especiales de JavaScript

Carácter	Significado Elementos para los que está definido
\b	Retroceso (Backspace)
\f	[Form feed]
\n	Nueva línea
\r	Retorno de carro [Carriage return]
\t	Tabulador [Tab]
\v	Tabulador vertical
\'	Apóstrofe o comilla simple
\"	Doble comilla
****	Caracter Backslash (\).
\XXX	El carácter de codificación Latin-1 especificado por tres dígitos octales XXX entre 0 y 377.
\xXX	El carácter de codificación Latin-1 especificado por dos dígitos hexadecimales XX entre 00 y FF.
\uXXXX	El carácter Unicode especificado por cuatro dígitos hexadecimales XXXX.

5.8. Caracteres de escape.

Se puede insertar una comilla dentro de una cadena precediéndola por una contrabarra [backslash]. Esto se conoce comoescapar la comilla. Por ejemplo:

```
var texto = "El lee \"La Incineración de Sam McGee\" de R.W. Service."
document.write(texto)
```

El resultado de ello sería:

El lee "La Incineración de Sam McGee" de R.W. Service.

Para incluir el literal correspondiente a la contrabarra [backslash] dentro de una cadena, deberá escapar el caracter de contrabarra [backslash]. Por ejemplo, para asignar la ruta de un archivo c:\temp a una cadena, use lo siguiente:

```
var home = "c:\\temp"
```

6. 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. && || !
Bit a Bit	Operaciones aritméticas o de desplazamiento. & | ^ ~<< >> >>>
Objeto	Evalúan la herencia y capacidades de un objeto. . [] () delete in instanceof new this
Miscelánea	Operadores con comportamiento especial. , ¿: typeof void

6.1. Operadores de comparación.

Tabla. Operadores de comparación

Sintaxis	Nombre	Tipos de operandos	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.

6.2. Operadores aritméticos.

Tabla. Operadores de aritméticos

Sintaxis	Nombre	Tipos de operandos	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
-varlor	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.
```

6.3. 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 operandos	Significado
=	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

6.4. 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
```

6.5. Operadores bit a bit.

Para los programadores de scripts, las operaciones bit a bit suelen ser un tema avanzado. Los operandos numéricos, pueden aparecer en JavaScript en cualquiera de los tres formatos posibles (decimal, octal o hexadecimal). Tan pronto como el operador tenga un operando, su valor se convertirá a representación binaria (32 bits de longitud). Las tres primeras operaciones binarias bit a bit que podemos realizar son AND, OR y XOR y los resultados de comparar bit a bit serán:

Bit a bit AND: 1 si ambos dígitos son 1.

Bit a bit OR: 1 si cualquiera de los dos dígitos es 1.

Bit a bit XOR: 1 si sólo un dígito es 1.

Tabla de operador bit a bit en JavaScript

Operador	Nombre	Operando izquierdo	Operando derecho
&	Desplazamiento AND	Valor integer	Valor integer
|	Desplazamiento OR	Valor integer	Valor integer
^	Desplazamiento XOR	Valor integer	Valor integer
~	Desplazamiento NOT	(ninguno)	Valor integer
<<	Desplazamiento a la izquierda	Valor integer	Cantidad a deplazar
>>	Desplazamiento a la derecha	Valor integer	Cantidad a deplazar
>>>	Desplazam. derecha rellanando con 0	Valor integer	Cantidad a deplazar

Por ejemplo:

```
4 << 2 // resultado = 16
```

6.6. 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.
}
```

6.7. 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);
}
```

7. Comentarios al código.

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
}
```

8. Sentencias.

Los lenguajes de programación suelen definir tres tipos de sentencias o instrucciones: secuenciales, selectivas e iterativas.

Sentencias Secuenciales

Podemos describir tres sentencias secuenciales en JavaScript: asignación, escritura y lectura.

```
variable=expresión
```

Esta sentencia, por si misma, no ejecuta ninguna acción "visible" para el usuario de nuestra página web, pero es fundamental para que nuestros programas puedan ir haciendo poco a su trabajo, realizando cálculos que se van almacenando en las variables.

Si queremos interactuar con el usuario, habremos de utilizar las otras dos sentencias secuenciales: la escritura y la lectura. Por simplificar un poco, la sentencia de escritura podemos que considerar que es write. En realidad, debemos decir que es document.write ya que es un método de un objeto que se llama document. Como hemos dicho, este es uno de los mecanismos que podemos usar para escribir cosas desde JavaScript para que se puedan ver a través del navegador (normalmente, el otro mecanismo que se ha impuesto como estándar es el de utilizar capas y su propiedad innerHTML, como también veremos al estudiar el objeto document).

A continuación se muestran algunos ejemplos en los que se usa la asignación y la escritura en un programa de JavaScript.

```
document.write("<h2>Ejemplo de uso de sentencias de escritura</h2>" );
document.write( "<p>" );
anioActual=2016;
document.write( "Estamos en ", anioActual );
anioNacimiento=1981;
document.write( " por lo que si usted nació en ", anioNacimiento );
document.write( " su edad es de ", anioActual-anioNacimiento,"años" );
document.write( "</p>" );
```

9. Bloques de código.

Un bloque de código es utilizado para agrupar sentencias. El bloque está delimitado por un par de llaves:

```
{
  sentencia_1
  sentencia_2
  .
  sentencia_n
}
```

Ejemplo:

Los bloques de sentencias son comúnmente utilizados con sentencias de flujo de control (en general [por ejemplo] if, for, while).

```
while (x < 10)
{
  x++;
}
```

aquí, { x++; } es un bloque de sentencias.

JavaScript no posee alcance de bloque. Las variables introducidas en un bloque están limitadas a la función contenedora o script y los efectos de configurarlas persisten más allá del bloque en sí. En otras palabras, los bloques de sentencias no presentan un alcance. sin embargo los bloques "aislados" poseen una sintaxis válida, Usted no deseará utilizar bloques aislados en JavaScript, porque ellos no harán lo que usted piensa que harán, si usted piensa que ellos harán cualquier cosa tal como los bloques en C o Java.

Por ejemplo:

```
var x = 1; //x vale 1
{
  var x = 2; //x vale 2
}
alert(x); //salida 2
```

La salida de 2 se debe a que la sentencia var x dentro de un bloque está bajo el mismo alcance que la sentencia var x antes del bloque. En C o Java, el código equivalente tendría una salida de 1.

10. Decisiones.

En los lenguajes de programación, las instrucciones que te permiten controlar las decisiones y bucles de ejecución, se denominan "Estructuras de Control". Una estructura de control, dirige el flujo de ejecución a través de una secuencia de instrucciones, basadas en decisiones simples y en otros factores.

Una parte muy importante de una estructura de control es la "condición". Cada condición es una expresión que se evalúa a true o false.

En JavaScript tenemos varias estructuras de control, para las diferentes situaciones que te puedas encontrar durante la programación. Tres de las estructuras de control más comunes son: construcciones if, construcciones if...else y los bucles.

10.1. Construcción 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");
}
```

10.2. Construcción 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.");
}
```

10.3. Construcción 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;
}
```

11. Bucles.

Los bucles son estructuras repetitivas, que se ejecutarán un número de veces fijado expresamente, o que dependerá de si se cumple una determinada condición.

11.1. Bucle 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.
}
```

11.2. Bucle 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++;
}
```

12. Utilización de objetos.

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.

Una vez repasados en la unidad anterior los fundamentos de la programación, vamos a profundizar un poco más en lo que se refiere a los objetos, que podremos colocar en la mayoría de nuestros documentos.

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.

12.1. 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/>");
```

El objeto String se utiliza para manipular una cadena almacenada de texto.

Los objetos String se crean con new String()

Sintaxis

```
var txt = new String("cadena");
```

o simplemente:

```
var txt = "cadena";
```

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

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 especificado por 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
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.
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 expresión 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
loUpperCase()	Convierte una cadena a mayúsculas
valueOf()	Devuelve el valor primitive de un objeto String

12.2. 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 />");
```

12.3. 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. Sí 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);
```

12.4. 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:

- 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.
- 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
```

12.5. 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)

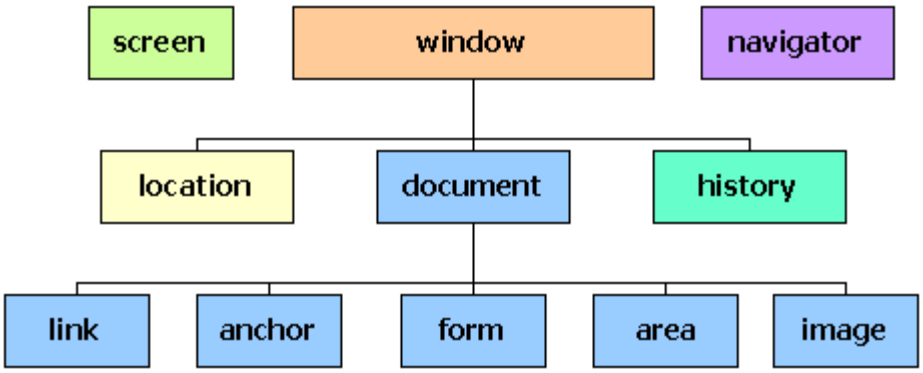
Algunos ejemplos de uso:

```
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);
```

13. 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

13.1. 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
	Elimina e! 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(), etc…**

```
<html>
  <head>
  <title>ejemplo de javascript</title>
  <script>
  <!--
    function moverventana()
    {
      mi_ventana.moveby(5,5);
      i++;
      if (i<20)
        settimeout('moverventana()',100);
      else
        mi_ventana.close();
    }
  //-->
  </script>
  </head>
  <body>
  <script language="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>
```

13.2. 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 documento
applets[]	Es un array que contiene todos los applets del documento
forms[]	Es un array que contiene todos los formularios dei documento
images[]	Es un array que contiene todas las imágenes del documento
links[]	Es un array que contiene todos los enlaces del documento

Tabla Propiedades del objeto document

Propiedades	Descripción
cookie	Devuelve todos ¡os nombres/valores de las cookies en el documento
domain	Cadena que contiene el nombre de dominio del servidor que cargó el documento
lastmodified	Devuelve la fecha y hora de la última modificación del documento
readystate	Devuelve el estado de carga del documento actual
referrer	Cadena que contiene la URL del documento desde el cuál llegamos al documento 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 utilizar 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

13.3. 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 de¡ 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 ¡a 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>
     <!--
      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>
```

13.4. 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>
      <!-- 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>
```

13.5. 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 Histoy

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 Histoy

Métodos	Descripción
back()	Carga la URL del documento anterior del historial
forward()	Carga la URL del documento 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>
```

13.6. 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 píxel
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>
  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>
```

14. Funciones predefinidas del lenguaje.

Cuando se trabaja con los objetos predefinidos en JavaScript, estamos utilizando sus métodos para poder interactuar con ellos. Todos los métodos utilizados, son en realidad funciones (llevan siempre paréntesis con o sin parámetros). En JavaScript, disponemos de algunos elementos que necesiten ser tratados a escala global y que no pertenecen a ningún objeto en particular (o que se pueden aplicar a cualquier objeto).

Tabla. Propiedades globales o predefinidas en JavaScript

Propiedad	Descripción
Infinity	Un valor numérico que representa el infinito positivo/negativo.
NaN	Valor que no es numérico “Not a Number”
undefined()	Indica que a esa variable no le ha sido asignado un valor.

Las funciones predefinidas en el lenguaje JavaScript, que se pueden utilizar a nivel global en cualquier parte del código JavaScript, son las que aparecen en la siguiente tabla:

Tabla Funciones globales o predefinidas en JavaScript

Propiedad	Descripción
decodeURI()	Decodifica los caracteres especiales de una URL excepto:, / ? : @ & = + $ #
decodeURIComponent()	Decodifica todos los caracteres especiales de la URL.
encodeURI()	Codifica los caracteres especiales de una URL excepto:, / ? : @ & = + $ #
escape()	Codifica caracteres especiales excepto: * @ - _ + . /
eval()	Evalúa una cadena y la ejecuta si contiene código u operaciones.
isFinite()	Determina si un valor es un número finito válido.
isNaN()	Determina cuando un valor no es un número.
Number()	Convierte el valor de un objeto a un número.
parseFloat()	Convierte el valor de una cadena a un número real.
parseInt()	Convierte el valor de una cadena a un número entero.
unescape()	Decodifica caracteres especiales en una cadena, excepto: * @ - _ + . /

Estas funciones no están asociadas a ningún objeto en particular. La funcionalidad principal es convertir un objeto en otro.

- **escape().** La función escape () tiene como argumento una cadena de texto y devuelve dicha cadena utilizando la codificación hexadecimal en el conjunto de caracteres latinos ISO3. Si por ejemplo quisiéramos saber la codificación del carácter (?), podríamos utilizar el siguiente ejemplo y verificar que obtendríamos la codificación%3F.

```
var input=prompt("Introduce una cadena");
varinputCodificado = escape (input);
alert("Cadena codificada: " + inputCodificado);
```

- **uncape().** Es la función opuesta a escape (). Es decir, que esta función decodifica los caracteres que estén codificados.
- **eval().** Esta función tiene como argumento una expresión y devuelve el valor de la misma para poder ser ejecutada como código JavaScript. El siguiente ejemplo permite ingresar al usuario una operación numérica y a continuación muestra el resultado de dicha operación:

```
var input = prompt("Introduce una operación numérica");
var resultado = eval(input);
alert ("El resultado de la operación es: " + resultado);
```

- **isFinite().** Esta función comprueba si el valor pasado como argumento corresponde o no a un número infinito. En JavaScript un valor se define como finito si se encuentra en el rango de ±1.7976931348623157 · 10308. Si el argumento no se encuentra en este rango o no es un valor numérico, la función devuelve false, en caso contrario devuelve true. Esta función es útil a la hora de realizar comprobaciones en sentencias condicionales y decidir en base al resultado si ejecutar una serie de instrucciones u otras. De este modo podemos comprobar que los resultados de las operaciones matemáticas no sobrepasen los límites numéricos de JavaScript y evitar la generación de errores de este tipo en nuestra aplicación web.

```
if(isFinite(argumento)) {
      //instrucciones si el argumento es un número finito
}
else {
      //instrucciones si el argumento no es un número finito
}
```

- **isNaN().** isNaN () es el acrónimo de isNot a Number (no es un número). Esta función evalúa si el objeto pasado como argumento es de tipo numérico. El siguiente ejemplo evalúa si el dato ingresado por el usuario es de tipo numérico y en base a eso, utilizando una sentencia condicional, ejecutamos una instrucción u otra.

```
var input=prompt("Introduce un valor numérico: ");
if (isNaN (input) )
{
  alert ("El dato ingresado no es numérico.");
}else
{
  alert("El dato ingresado es numérico.");
}
```

- **String().** La función String() convierte el objeto pasado como argumento en una cadena de texto. Por ejemplo, podemos crear un objeto de tipo Date, y convertir dicho objeto en una cadena de texto.

```
var fecha = new Date()
var fechaString = String(fecha)
alert ("La fecha actual es: " + fechaString);
```

- **Number().** La función Number () convierte el objeto pasado como argumento en un número. Si la conversión falla, la función devuelve NaN (Not a Number).Si el parámetro es un objeto de tipo Date, la función Number o devolverá el número de milisegundos transcurridos desde la medianoche del 1 de enero de 1970 hasta la fecha actual.
- **parseInt().** Esta función intenta convertir una cadena de caracteres pasada como argumento en un número entero con una base especificada. Si no especificamos la base se utiliza automáticamente la base decimal (10). La base puede ser por ejemplo binaria (2), octal(8), decimal (10) o hexadecimal (16). Si la función encuentra en la cadena a convertir, algún carácter que no sea numérico, devuelve el valor encontrado hasta ese punto. Si el primer valor no es numérico, la función devuelve NaN.

```
var input = prompt("Introduce un valor: ");
var inputParsed = parseInt(input);
alert("parseInt("+input+") : "+inputParsed);
```

- **parseFloat().** Esta función es muy similar a la anterior. La diferencia es que en lugar de convertir el argumento a un número entero, intenta convertirlo a un número de punto flotante. Si la función encuentra en la cadena a convertir, algún carácter que no corresponda al formato de un número decimal, devuelve el valor encontrado hasta ese punto. Si el primer valor no es numérico, la función devuelve NaN.

```
var input = prompt("Introduce un valor: ");
var inputParsed = parseFloat(input);
alert("parseFloat("+input+"): " + inputParsed);
```

15. Llamadas a funciones. Definición de 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.

15.1. 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) + ".");
```

15.2. 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 usarvar)
  // Esta variable local tendrá el mismo nombre que otra variable
    //global pero con distintocontenido.
  // Si no usáramos var estaríamos modificando la variable
globalchica.
  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 + ".");
```

15.3. Funciones anidadas.

Los navegadores más modernos nos proporcionan la opción de anidar unas funciones dentro de otras. Es decir podemos programar una función dentro de otra función.

Cuando no tenemos funciones anidadas, cada función que definamos será accesible por todo el código, es decir serán funciones globales. Con las funciones anidadas, podemos encapsular la accesibilidad de una función dentro de otra y hacer que esa función sea privada o local a la función principal. Tampoco se recomienda el reutilizar nombres de funciones con esta técnica, para evitar problemas o confusiones posteriores.

La estructura de las funciones anidadas será algo así:

```
function principalA()
{
  // instrucciones
  function internaA1()
  {
    // instrucciones
  }
  // instrucciones
}
function principalB()
{
  // instrucciones
  function internaB1()
  {
  // instrucciones
  }
  function internaB2()
  {
   // instrucciones
  }
  // instrucciones
}
```

Una buena opción para aplicar las funciones anidadas, es cuando tenemos una secuencia de instrucciones que necesitan ser llamadas desde múltiples sitios dentro de una función, y esas instrucciones sólo tienen significado dentro del contexto de esa función principal. En otras palabras, en lugar de romper la secuencia de una función muy larga en varias funciones globales, haremos lo mismo pero utilizando funciones locales.
Ejemplo de una función anidada:

```
function hipotenusa(a, b)
{
  function cuadrado(x)
  {
    return x*x;
  }
  return Math.sqrt(cuadrado(a) + cuadrado(b));
}
document.write("<br/>La hipotenusa de 1 y 2 es: "+hipotenusa(1,2));
// Imprimirá: La hipotenusa de 1 y 2 es: 2.23606797749979
```

16. "Arrays".

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

JavaScript emplea infinidad de arrays internamente para gestionar los objetos HTML en el documento, propiedades del navegador, etc. Por ejemplo, si tu documento contiene 10 enlaces, el navegador mantiene una tabla con todos esos enlaces. Tú podrás acceder a esos enlaces por su número de enlace (comenzando en el 0 como primer enlace). Si empleamos la sentencia de array para acceder a esos enlaces, el nombre del array estará seguido del número índice (número del enlace) entre corchetes, como por ejemplo: **document.links[0]** , representará el primer enlace en el documento.

Existen colecciones dentro del objeto document. Pues bien, cada una de esas colecciones (**anchors**[], **forms**[], **links**[], e **images**[]) será un array que contendrá las referencias de todas las anclas, formularios, enlaces e imágenes del documento.

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.

Lo haríamos del siguiente modo:

```
var misCoches=new Array();
misCoches[0]="Seat";
misCoches[1]="BMW";
misCoches[2]="Audi";
misCoches[3]="Toyota";
```

16.1. Creación de un array.

Para crear un objeto array, usaremos el constructor new Array. Por ejemplo:

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

Un objeto array dispone de una propiedad que nos indica su longitud. Esta propiedades **length** (longitud, que será 0 para un array vacío). Si queremos precisar el tamaño del array durante la inicialización (por ejemplo para que se cargue con valores null), podríamos hacerlo pasándole un parámetro al constructor. Por ejemplo, aquí puedes ver cómo crear un nuevo array que almacene la información de 40 personas:

```
var miarray= new Array()
```

En JavaScript podremos asignar un valor a cualquier posición del array en cualquier momento, esté o no definida previamente. La propiedad length se ajustará automáticamente al nuevo tamaño del array.

```
personas[53] = "Irene Sáinz Veiga";
longitud = personas.length;// asignará a la variable longitud 54
```

24.1.1. Introducir datos en un array.

Introducir datos en un array, es tan simple como crear una serie de sentencias de asignación, una porcada 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";
```

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 unPlaneta 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"];
```

Los corchetes sustituyen a la llamada al constructor **new Array().** Hay que tener cuidado con esta forma de creación que quizás no funcione en navegadores antiguos.

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/>");
```

24.1.2. 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++;
}
```

Empleando la sentencia foreach in (disponible en versiones de JavaScript 1.6 o superiores):

```
foreach (variable en objeto)
  sentencia
```

Esta sentencia repite la variable indicada, sobre todos los valores de las propiedades del objeto. Para cada propiedad distinta, se ejecutará la sentencia especificada.

Ejemplo 1:

```
foreach (var planeta in sistemaSolar)
  document.write(planeta+"<br/>");
// Imprimirá todos los nombres de planeta con un salto de línea al
//final de cada nombre.
```

Ejemplo 2:

```
[document.write(planeta +"<br/>") foreach (planeta in sistemaSolar)];
```

24.1.3. 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";
```

24.1.4. 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
```

16.2. Arrays paralelos.

El usar arrays para almacenar información, facilita una gran versatilidad en las aplicaciones, a la horade realizar búsquedas de elementos dentro del array. Pero en algunos casos, podría ser muy útil hacer las búsquedas en varios arrays a la vez, de tal forma que podamos almacenar diferentes tipos de información, en arrays que estén sincronizados.
Por ejemplo consideremos los siguientes arrays:

```
var profesores = ["Cristina","Catalina","Vieites","Benjamin"];
var asignaturas=["Seguridad","Bases de
Datos","SistemasInformáticos","Redes"];
var alumnos=[24,17,28,26];
```

Usando estos tres arrays de forma sincronizada, y haciendo referencia a un mismo índice (por ejemplo índice=3), podríamos saber que Benjamín es profesor de Redes y tiene 26 alumnos en clase. Veamos un ejemplo de código que recorrería todos los profesores que tengamos en el array imprimiendo la información sobre la asignatura que imparten y cuántos alumnos tienen en clase:

```
var profesores = ["Cristina","Catalina","Vieites","Benjamin"];
var asignaturas=["Seguridad","Bases de Datos","Sistemas
Informáticos","Redes"];
var alumnos=[24,17,28,26];
function imprimeDatos(indice)
{
  document.write("<br/>"+profesores[indice]+" del módulo de
  "+asignaturas[indice]+", tiene "+alumnos[indice]+" alumnos en
clase.");
}
for (i=0;i<profesores.length;i++){
  imprimeDatos(i);
}
```

16.3. Arrays multidimensionales.

Una alternativa a los arrays paralelos es la simulación de un array multidimensional. Si bien es cierto que en JavaScript los arrays son unidimensionales, podemos crear arrays que en sus posiciones con tengan otros arrays u otros objetos. Podemos crear de esta forma arrays bidimensionales, tridimensionales, etc. Por ejemplo podemos realizar el ejemplo anterior creando un array bidimensional de la siguiente forma:

```
var datos = new Array();
datos[0] = new Array("Cristina","Seguridad",24);
datos[1] = new Array("Catalina","Bases de Datos",17);
datos[2] = new Array("Vieites","Sistemas Informáticos",28);
datos[3] = new Array("Benjamin","Redes",26);
```

ó bien usando una definición más breve y literal del array:

```
var datos = [
  ["Cristina","Seguridad",24],
  ["Catalina","Bases de Datos",17],
  ["Vieites","Sistemas Informáticos",28],
  ["Benjamin","Redes",26]
];
```

nombrearray[indice1][indice2][indice3] (nos permitiría acceder a una posición determinada en un array tridimensional).
Por ejemplo:

```
document.write("<br/>Quien imparte Bases de Datos? "+datos[1][0]); //
Catalina
document.write("<br/>Asignatura de Vieites: "+datos[2][1]);
// Sistemas Informaticos
document.write("<br/>Alumnos de Benjamin: "+datos[3][2]); // 26
```

Si queremos imprimir toda la información del array multidimensional, tal y como hicimos en el apartado anterior podríamos hacerlo con un bucle for:

```
for (i=0;i<datos.length;i++)
{
  document.write("<br/>"+datos[i][0]+" del módulo de "+datos[i][1]+",
tiene      "+datos[i][2]+"alumnos en clase.");
}
```

Obtendríamos como resultado:
Cristina del módulo de Seguridad, tiene 24 alumnos en clase.
Catalina del módulo de Bases de Datos, tiene 17 alumnos en clase.
Vieites del módulo de Sistemas Informáticos, tiene 28 alumnos en clase.
Benjamin del módulo de Redes, tiene 26 alumnos en clase.

También podríamos imprimir todos los datos de los arrays dentro de una tabla:

```
document.write("<table border=1>");
for (i=0;i<datos.length;i++)
{
  document.write("<tr>");
  for (j=0;j<datos[i].length;j++)
  {
    document.write("<td>"+datos[i][j]+"</td>");
  }
  document.write("</tr>");
}
document.write("</table>");
```

17. Creación de objetos.

Toda la información que proviene del navegador o del documento está organizada en un modelo de objetos, con propiedades y métodos. Pues bien, JavaScript también permite crear tus propios objetos en memoria, objetos con propiedades y métodos que podemos definir como queramos. Estos objetos no serán elementos de la página de interfaz de usuario, pero sí que serán objetos que podrán contener datos (propiedades) y funciones (métodos), cuyos resultados si que se podrán mostrar en el navegador. El definir nuestros propios objetos, nos permitirá enlazar a cualquier número de propiedades o métodos que hayamos creado para ese objeto. Es decir, controlaremos la estructura del objeto, sus datos y su comportamiento.
Los objetos se crean empleando una función especial denominada constructor, determinada por el nombre del objeto. Ejemplo de una función constructor:

```
function Coche()
{
  // propiedades y métodos
}
```

Aunque esta función no contiene código, es sin embargo la base para crear objetos de tipo Coche. Puedes pensar en un constructor como un anteproyecto o plantilla, que será utilizada para crear objetos. Por convención, los nombres de los constructores se ponen generalmente con las iniciales de cada palabra en mayúscula, y cuando creamos un objeto con ese constructor (instancia de ese objeto), lo haremos empleando minúsculas al principio. Por ejemplo:

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

La palabra reservada **new** se emplea para crear objetos en JavaScript. Al crear la variable unCoche, técnicamente podríamos decir que hemos creado una instancia de la clase Coche, o que hemos instanciado el objeto Coche, o que hemos creado un objeto Coche, etc. Es decir hay varias formas de expresarlo, pero todas quieren decir lo mismo.

18. Definición de métodos y propiedades.

JavaScript proporciona una gran cantidad de objetos predefinidos. Los objetos están organizados de forma jerárquica en el que el objeto window se encuentra en el nivel más alto, seguido de document, frame, etc.. También disponemos de los arrays, que funcionan como objetos JavaScript. Cada uno de estos objetos tiene una serie de propiedad y métodos, útiles para desarrollar aplicaciones web.

18.1. Definición de métodos.

Los métodos son funciones que se enlazarán a los objetos, para que se pueda acceder a las propiedades de los mismos. Nos definimos un ejemplo de método, que se podría utilizar en la clase Coche:

```
function rellenarDeposito (litros)
{
  // Modificamos el valor de la propiedad cantidad de combustible
  this.cantidad = litros;
}
```

El método *rellenarDeposito*, que estamos programando a nivel **global**, hace referencia a la propiedad *this.cantidad* para indicar cuantos litros de combustible le vamos a echar al coche. Lo único que faltaría aquí es realizar la conexión entre el método **rellenarDeposito** y el objeto de *tipoCoche* (recuerda que los objetos podrán tener propiedades y métodos y hasta este momento sólo hemos definido propiedades dentro del constructor). Sin esta conexión la palabra reservada this no tiene sentido en esa función, ya que no sabría cuál es el objeto actual. Veamos cómo realizar la conexión de ese método, con el objeto dentro del constructor:

```
function Coche(marca, combustible)
{
  // Propiedades
  this.marca = marca;
  this.combustible = combustible;
  this.cantidad = 0;
  // Cantidad de combustible inicial por defecto en el depósito.
  // Métodos
  this.rellenarDeposito = rellenarDeposito;
}
```

Es muy importante destacar que el método **rellenarDeposito()** se referencia sin paréntesis dentro del constructor, this.rellenarDeposito = rellenarDeposito.
Ejemplo de uso del método anterior:

```
cocheDeMartin.rellenarDeposito(35);
document.write("<br/>El coche de Martin tiene
"+cocheDeMartin.cantidad+ " litros de " +
cocheDeMartin.combustible+ " en el depósito.");
// Imprimirá
// El coche de Martin tiene 35 litros de diesel en el depósito.
```

La forma en la que hemos definido el método rellenarDeposito a nivel global, no es la mejor práctica en la programación orientada a objetos. Una mejor aproximación sería definir el contenido de la función rellenarDeposito dentro del constructor, ya que de esta forma los métodos al estar programados a nivel local aportan mayor privacidad y seguridad al objeto en general, por ejemplo:

```
function Coche(marca, combustible)
{
  // Propiedades
  this.marca = marca;
  this.combustible = combustible;
  this.cantidad = 0;
  // Métodos
  this.rellenarDeposito = function (litros)
  {
    this.cantidad=litros;
  };
}
```

18.2. Definición de propiedades.

Una vez que ya sabemos cómo crear un constructor para un objeto, vamos a ver cómo podemos crear una propiedad específica para ese objeto. Las propiedades para nuestro objeto se crearán dentro del constructor empleando para ello la palabra reservada this. Véase el siguiente ejemplo:

```
function Coche()
{
  // Propiedades
  this.marca = "Audi A6";
  this.combustible = "diesel";
  this.cantidad = 0; // Cantidad de combustible en el depósito.
}
```

La palabra reservada **this**, se utiliza para hacer referencia al objeto actual, que en este caso será el objeto que está siendo creado por el constructor. Por lo tanto, usarás this, para crear nuevas propiedades para el objeto. El único problema con el ejemplo anterior es que todos los coches que hagamos del tipo Coche será siempre Audi A6, diésel, y sin litros de combustible en el depósito. Por ejemplo;

```
var cocheDeMartin = new Coche();
var cocheDeSilvia = new Coche();
```

A partir de ahora, si no modificamos las propiedades del coche de Martin y de Silvia, en el momento de instanciarlos tendrán ambos un Audi A6 a diesel y sin combustible en el depósito.

Lo ideal sería por lo tanto que en el momento de instanciar un objeto de tipo Coche, que le digamos al menos la marca de coche y el tipo de combustible que utiliza. Para ello tenemos que modificar el constructor, que quedará de la siguiente forma:

```
function Coche(marca, combustible)
{
  // Propiedades
  this.marca = marca;
  this.combustible = combustible;
  this.cantidad = 0;
  // Cantidad de combustible inicial por defecto en el depósito.
}
```

Ahora sí que podríamos crear dos tipos diferentes de coche:

```
var cocheDeMartin = new Coche("Volkswagen Golf","gasolina");
var cocheDeSilvia = new Coche("Mercedes SLK","diesel");
```

Y también podemos acceder a las propiedades de esos objetos, consultar sus valores o modificarlos.
Por ejemplo:

```
document.write("<br/>El coche de Martin es un: +cocheDeMartin.marca+"
a"+cocheDeMartin.combustible);
document.write("<br/>El coche de Silvia es un: "+cocheDeSilvia.marca+"
a "+cocheDeSilvia.combustible);
// Imprimirá:
// El coche de Martin es un: Volkswagen Golf a gasolina
// El coche de Silvia es un: Mercedes SLK a diesel
// Ahora modificamos la marca y el combustible del coche de Martin:
cocheDeMartin.marca = "BMW X5";
cocheDeMartin.combustible = "diesel";
document.write("<br/>El coche de Martin es un: " + cocheDeMartin.marca
+ " a " +
cocheDeMartin.combustible);
// Imprimirá: El coche de Martin es un: BMW X5 a diésel
```

18.3. Definición de objetos literales.

Un literal es un valor fijo que se especifica en JavaScript. Un objeto literal será un conjunto, de cero o más parejas del tipo **nombre:valor**.
Por ejemplo:

```
avion = { marca:"Boeing" , modelo:"747" , pasajeros:"450" };
```

Es equivalente a:

```
var avion = new Object();
avion.marca = "Boeing";
avion.modelo = "747";
avion.pasajeros = "450";
```

Para referirnos desde JavaScript a una propiedad del objeto avión podríamos hacerlo con:

```
document.write(avion.marca); // o trambién se podría hacer con:
document.write(avion["modelo"]);
```

Podríamos tener un conjunto de objetos literales simplemente creando un array que contenga encada posición una definición de objeto literal:

```
var datos=[
{"id":"2","nombrecentro":"IES Padre Poveda", "localidad":"Guadix",
"provincia":"Granada"},
{"id":"10","nombrecentro":"IES Aguadulce","localidad": "Roquetas de
Mar","provincia":"Almería"},{"id":"9","nombrecentro":"IES Aricel",
"localidad":"Albolote", "provincia":"Granada"},
```

De la siguiente forma se podría recorrer el array de datos para mostrar su contenido:

```
for (var i=0; i<datos.length; i++)
{
  document.write("Centro ID: "+datos[i].id+" - ");
  document.write("Nombre: "+datos[i].nombrecentro+" - ");
  document.write("Localidad: "+datos[i].localidad+" - ");
  document.write("Provincia: "+datos[i].provincia+"<br/>");
}
```

Y obtendríamos como resultado:
Centro ID: 2 - Nombre: IES Padre Poveda - Localidad: Lugo - Provincia: Lugo
Centro ID: 10 - Nombre: IES Aguadulce - Localidad: Roquetas de Mar - Provincia: Almería
Centro ID: 9 - Nombre: IES Aricel - Localidad: Albolote - Provincia: Granada