Resumen Javascript

1. Lo básico

1.1. Introducción

En este tema haremos un resumen de lo visto en el manual de javascript. El resumen puede servirnos de recordatorio o de consulta para nuestros trabajos.

Aunque no sigamos exactamente el mismo orden que en el manual, pondremos todo lo más importante concerniente al código javascript.

En breves dispondremos también del resumen en formato PDF ... estamos trabajando en ello

1.2. Preparación

1.2.1. Escribir el código

- Espacios y tabulaciones: no se tienen en cuenta espacios extra o tabulaciones al esccribir el código, al igual que en HTML y CSS.
- **Mayúsculas y minúsculas**: **sí** se distingue entre mayúsculas y minúsculas.
- **Comentarios**: De una línea; empiezan por dos barras inclinadas `//` y hasta el cambio de línea. De varias líneas; encerrados entre `/* ... */`.
- **Sentencias**: Cada sentencia o instrucción se escribe en una línea aparte. De no ser así debe acabar por un punto y coma (`;`). No poner una misma sentencia en dos líneas distintas.

1.2.2. Insertar el código

En el mismo documento: mediante la etiqueta de "script":

```
<script type="text/javascript"> ... </script>
```

En un documento aparte: Enlazándolo mediante la etiqueta de "script":

```
<script type="text/javascript" src="ruta_del_archivo"></script>
```

En los elementos HTML: mediante los atributos de eventos u otros que enlacen con javascript.

```
<p onclick=" /*codigo javascript*/ "> ... </p>
```

1.2.3. A tener en cuenta

- **El Doctype:** La página HTML tiene que tener declarado el Doctype para que funcione correctamente javascript en todos los navegadores.
- **Validación:** Tener bien escrito el código HTML es necesario para un correcto funcionamiento de javascript. La validación garantiza un código bien escrito.
- **Etiqueta noscript**: Podemos usar la etiqueta `<noscript> ... </noscript>` para mostrar un texto alternativo en caso de que el usuario no tenga activado javascript.
- **Corrección de errores:** Para buscar los errores que podamos hacer al escribir el código, la mayoría de los navegadores tienen una "consola de errores" en la que podemos verlos.

1.3. Variables

Variable: Elemento que se emplea para guardar un dato o valor.

1.3.1. Definir una variable

- Mediante la palabra reservada `var` : `var variable`
- Asignándole un valor : `variable = 8`
- Con las dos formas anteriores a la vez : `var variable = 8`

1.3.2. Normas para nombrar las variables

Una variable puede tener cualquier nombre salvo algunas excepciones:

- No usar las palabras reservadas para el lenguaje javascript.
- Solo podemos usar letras, números, y los signos $ (dólar) y _ (guión bajo)
- El primer carácter no puede ser un número.

1.3.3. Tipos de variables

Dependiendo del dato que almacene, la variable puede ser de alguno de los siguientes tipos:

- **Numérico:** Su valor es un número. Pueden ser números enteros, con decimales o en notación científica. También admite números en sistema octal o hexadecimal.
- **Cadena:** Su valor es una cadena de texto. Para escribirlo debemos ponerlo entre comillas. ej.: `var texto = "esta es una cadena de texto"`.
- **Booleano:** Indica si se cumple una condición. Los únicos valores posibles son `true` (verdadero) y `false` (falso).
- **Objetos:** Elementos más complejos que veremos más adelante, como pueden ser arrays, funciones, elementos del DOM, etc.

1.3.4. Caracteres de escape

Podemos usar comillas simples y dobles para poner comillas dentro de un código que ya está entre comillas, pero no podemos ir más allá. En este caso debemos usar los caracteres de escape. Estos son:

- \n : Salto de línea.
- \" : Comillas dobles.
- \' : Comillas simples.
- \t : Tabulador.
- \r : Retorno del carro.
- \f : Avance de página.
- \b : Retroceder espacio.
- \\ : Contrabarra.

La forma de escribir los caracteres especiales es usando la contrabarra seguida del carácter a mostrar o de una referencia a la acción que se quiere obtener.

2. Operadores

2.1. Definición

Los operadores son elementos que permiten hacer operaciones entre los distintos datos que se manejan con javascript.

Éstas pueden hacerse utilizando el dato directamente o con la variable que lo contiene.

Aunque la mayoría de operadores se utilizan con datos numéricos, también hay algunos que pueden utilizarse con otro tipo de datos.

2.2. Operadores numéricos

Realizan las operaciones aritméticas clásicas entre dos números, a la que se le añade la operación módulo, que es el resto de la división. Los números pueden pasarse directamente o con variables.

Operadores numéricos

Nombre	Signo	Descripción	Ejemplo
Suma	**+**	Suma de dos números. También podemos pasar cadenas de texto, la segunda cadena se escribe seguida de la primera.	`c = a + b`
Resta	**-**	Resta de dos números	`c = a - b`
Multiplicación	*****	Multiplicación de dos números	`c = a * b`
División	**/**	División entre dos números	`c = a / b`
Módulo	**%**	Resto de la división entre dos números	`c = a % b`

La suma puede aplicarse también a las cadenas de texto, en este caso el resultado es una cadena que contiene a la primera cadena seguida por la segunda. Podemos sumar también cadenas con números, en este caso el número se convierte en cadena.

2.3. Operadores de incremento

Utilizados con números enteros (aunque también pueden usarse con números con decimales), modifican en una unidad el número al que se refieren.

Operadores de incremento

Nombre	Signo	Descripción	Ejemplo
Incremento creciente	++	Aumenta en una unidad el número.	++a
Incremento decreciente	--	Disminuye en una unidad el número	--a

2.3.1. Posición del operador

- **Delante:** (++a) Primero se produce el incremento y depués las demás operaciones.
- **Detrás:** (a++) Primero se hace el resto de operaciones y después se incrementa.

2.4. Operadores de asignación

2.4.1. Operador de asignación simple

Es el signo igual (=), y su principal misión es asignar un nuevo valor a las variables, o crearlas dándoles un valor, si no existían antes.

2.4.2. Operadores de asignación compuestos

Combinan los operadores numéricos con el operador se asignación simple. Ej.:

```
a += b // mismo resultado que a = a + b
```

Operadores de asignación compuestos

Nombre	Signo	Descripción	Ejemplo
Suma y asignación	**+=**	Suma dos números y asigna el resultado al primero.	`a += b`
Resta y asignación	**-=**	Resta dos números y asigna el resultado al primero.	`a -= b`
Multiplicación y asignación	***=**	Multiplica dos números y asigna el resultado al primero	`a *= b`
División y asignación	**/**	Divide dos números y asigna el resultado al primero	`a /= b`
Módulo y asignación	**%=**	Calcula el resto de la división entre dos números y asigna su valor al primero	`a %= b`

2.5. Operadores condicionales

Comprueban si un elemento cumple una condición. El resultado es siempre un valor booleano. Si la condición se cumple, el resultado es true (verdadero) y si no se cumple el resultado es false (falso). El operador indica la condición que debe cumplir.

Operadores condicionales

Nombre	Signo	Descripción	Ejemplo
igual	**==**	Comprueba si los dos elementos son iguales	`num1 == num2`
idéntico	**===**	Comprueba si los dos elementos son iguales, pero aquí se comprueba también que los datos sean del mismo tipo.	`num1 === num2`
No igual	**!=**	Comprueba si los dos elementos son distintos, si son iguales devuelve `false`	`num1 != num2`
No idéntico	**!==**	Igual que el anterior, pero también comprueba si los datos son del mismo tipo	`num1 !== num2`
Mayor que	**>**	Comprueba si el primer valor es mayor que el segundo, si es así devuelve `true` y si no devuelve `false`	`num1 > num2`
Menor que	**<**	Comprueba si el primer valor es menor que el segundo	`num1 == num2`
Mayor o igual que	**>=**	Comprueba si el primer valor es mayor o igual que el segundo.	`num1 >= num2`
Menor o igual que	**<=**	Comprueba si el primer valor es mayor o igual que el segundo.	`num1 <= num2`

2.6. Operadores lógicos

Cuando debe cumplirse (o no cumplirse) más de una condición al mismo tiempo, se utilizan los operadores lógicos. Los valores que devuelven los operadores lógicos son siempre booleanos e indican si lo indicado por el operador se cumple o no.

Operadores lógicos

Nombre	Signo	Descripción	Ejemplo
AND	**&&**	Devuelve `true` sólo si las dos condiciones son verdaderas, si no es así devuelve `false`	`n1 != n2 && n1 != n3`
OR	**||**	Devuelve `true` sólo si una de las dos condiciones es verdadera, sólo si las dos son falsas devuelve `false`	`n1 == n2 || n1 == n3`
Negación	**!**	Cambia el valor del elemento al que se le aplica es decir, si `variable` vale `true`, `!variable` valdrá `false`	`!variable`

3. Estructuras

3.1. Arrays

Definición: Lista o colección de datos que se guardan con un sólo nombre, dentro de un elemento.

Podemos guardar en un array todo tipo de datos, (numéricos, cadenas, objetos, etc ..) pudiendo cada dato de un mismo array ser de distinto tipo que los otros.

3.1.1. Formas de crear un array

Indicando sus elementos: éstos se ponen entre corchetes y separados por comas.

```
var direccion = ["norte","sur","este","oeste"];
```

Mediante la instrucción `new Array()`:

Aquí podemos también indicar los elementos.

```
var direccion = new Array("norte","sur","este","oeste");
```

Podemos crear también un array sin elementos, para añadírselos más tarde:

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

Y podemos indicar solamente el número de elementos que va a tener, por ejemplo para un array de 4 elementos:

```
var direccion = new Array(4);
```

3.1.2. Acceso y modificación de los datos

Para acceder a un dato concreto escribiremos el nombre del array y luego entre corchetes la posición en la que está. Ésta se empieza a contar desde el número 0.

```
dato1 = direccion[0] // valor de dato1 = primer elemento del array
```

Para cambiar el valor de un elemento o crear uno nuevo basta con asignarle un nuevo valor:

```
direccion[4] = "noroeste"
```

Al dar un nuevo valor a un elemento pueden pasar varias cosas:

- El elemento ya existía antes y tenía un valor. En este caso el nuevo valor sustituye al antiguo.
- El elemento existía antes pero no tenía valor. El elemento toma el valor que se le ha asignado.
- El elemento no existía anteriormente al ser el array más corto. Se crea el nuevo elemento y se alarga el número de elementos del array hasta el indicado. Si hace falta, se crean nuevos elementos sin valor para alcanzar el número indicado.

3.1.3. Saber el número de elementos de un array.

la propiedad `.length` nos dice el número de elementos que tiene un array:

```
num=dirección.length;
```

3.2. Funciones

Una función es un bloque de código que lo aislamos del resto para poder ejecutarlo una o varias veces desde otro punto del código.

3.2.1. Sintaxis.

La sintaxis de la estructura para crear una función es la siguiente:

```
function miFuncion(a,b) {
   //código del interior de la función, aislado del resto
   return valorRetorno;
   }
```

Veamos las partes de esta estructura:

- **`function`**: Palabra reservada `function` que ponemos siempre al principio.
- **`miFuncion`**: Nombre que le damos a la función para poder llamarla después y distinguirla de otras. Si ignoramos este paso tendremos una función anónima.
- **`(a,b)`:** Argumentos o parámetros. Son datos que pasamos al interior de la función para que opere con ellos. Podemos pasar 0, 1, ó varios argumentos. Cuando hay más de uno van separados por comas. Los paréntesis son siempre obligatorios, aunque no pasemos argumentos.
- **`{ ... }`** : Entre llaves escribiremos el código javascript que se ejecuta al llamar a la función.
- **`return valorRetorno :`** Valor que devuelve la función una vez ejecutado el código. Consta de la palabra reservada `return` seguido de la variable donde hemos guardado el valor que queremos retornar. Lo pondremos siempre al final del código de la función. No es obligatorio si no queremos que devuelva un valor.

3.2.2. Llamada a una función

Una vez creada la función podemos llamarla desde cualquier parte del código, simplemente escribiendo su nombre seguido del paréntesis (con parámetros o no). Por ejemplo en la función anterior:

```
miFunción(2,5);
```

Esta llamada ejecutará el código del interior de la función. Pero si queremos recoger el **valor de retorno** debemos guardar la llamada en una variable:

```
resultado=miFuncion(2,5)
```

La variable `resultado` recoge el valor de retorno de la función.

3.3. Estructuras condicionales

Las estructuras condicionales comprueban si se cumple una determinada condición. Si ésta se cumple se ejecuta un código, si no es así se ejecuta otro código o ninguno.

3.3.1. Estructura if

La sintaxis básica de la estructura `if` es la siguiente

```
if (a==b /*condición*/ ) {
   //código cuando se cumple la condición.
   }
```

- **`if:`** Se pone en primer lugar la palabra reservada `if`.
- **`(/*condición*/):`** Entre paréntesis escribimos la condición que debe cumplirse (normalmente se emplean los operadores condicionales y lógicos para escribir la condición).
- **`{...}:`** Entre llaves se escribe el código que debe ejecutarse en el caso de que la condición se cumpla.

Podemos ampliar este código escribiendo también lo que debe hacer javascript cuando la condición no se cumple:

```
if (a==b /*condición*/ ) {
   //código cuando se cumple la condición.
   }
else { //código cuando la condición no se cumple.
   }
```

- **`else :`** Ponemos la palabra reservada `else`
- **`{ ... } :`** Escribimos después entre llaves el código que debe ejecutarse cuando la condición no se cumple.
-

Comprobar varias condiciones.

Cuando hay varias condiciones que deben cumplirse podemos anidar varias estructuras if - else de la siguiente manera:

```
if (a==b /*condición*/ ) {
   //código cuando se cumple la primera condición.
   }
else if (a==c /*condición*/ ) {
   //código cuando se cumple la segunda condición.
   }
/*   ....
  ....  */
else if (a==n /*condición*/ ) {
   //código cuando se cumple la enésima condición.
   }
else { //código cuando no se cumple ninguna condición de las anteriores.
   }
```

3.3.2. Estructura switch

La estructura condicional **`switch`** comprueba los diferentes valores que puede tomar una variable, de manera que con cada valor puede ejecutare un código diferente. Su sintaxis es:

```
switch (variable) {
case valor_1 :
//código a ejecutar para el valor 1
break;
case valor_2 :
//código a ejecutar para el valor 2
break;
 /*....
 ....*/
case valor_n :
//código a ejecutar para el valor n
break;
default:
//Código a ejecutar en caso de que no tenga ningún valor de los anteriores
break;
}
```

En primer lugar ponemos la palabra reservada `switch` seguida de un paréntesis donde escribiremos del nombre de la variable que queremos comprobar.

Después dentro de las llaves, escribimos una serie de instrucciones que se repiten para cada valor. En cada uno de estos bloques pondremos:

- En la primera línea escribimos la palabra reservada `case` seguida del valor que queremos comprobar. cerramos la línea con el signo de dos puntos.
- Las siguientes líneas constan del código a ejecutar en el caso de que la variable tenga el valor indicado.
- La última línea del bloque es la palabra reservada `break`, que indica que hemos encontrado la coincidencia y se sale de la estructura.

El último de estos bloques no empieza por `case valor_n:`, sino por `default:`. Indicamos aquí el código a ejecutar cuando ninguna de las condiciones anteriores se dan.

3.4. Bucles

Un bucle es una estructura en la cual hay un código que se repite una y otra vez mientras se cumpla una condición.

El bucle, por tanto consta de dos partes, el código que se repite, y la condición. En cada repetición se comprueba si la condición se cumple. Cuando esta deja de cumplirse se sale del bucle.

Por tanto para que el bucle no se repita indefinidamente debemos cambiar en cada repetición la variable que controla el bucle, de forma que haya un momento en que la condición no se cumpla.

3.4.1. Estructura for

Es la estructura más común para crear un bucle, y su sintaxis es la siguiente:

```
for (i=1 /*inicialización*/ ; i<=10 /*condición*/ ; i++ /*actualización*/ ) {
   //código javascript que se repite en cada vuelta del bucle.
   }
```

En primer lugar escribimos la palabra reservada `for`. Después dentro de un paréntesis escribimos tres sentencias separadas por punto y coma, estas son:

- **Inicialización:** Damos un valor inicial a la variable que controla el bucle.
- **Condición:** Indicamos una condición para la variable que controla el bucle, de manera que cuando no se cumpla deje de repetirse.
- **Actualización:** Cambiamos aquí el valor de la variable que controla el bucle, para que llegue un momento en que la condición deje de cumplirse.

Después entre llaves escribimos el código que debe repetirse. Dentro de este código podemos poner también la variable que controla el bucle; ésta tendrá diferente valor en cada vuelta.

3.4.2. Estructura for in para arrays

Esta es una estructura especial para recorrer los elementos de un array. Es similar a la anterior, sin embargo cambia lo que ponemos dentro del paréntesis. Su sintaxis es la siguiente:

```
for ( i in miarray ) {
  document.write(miarray[i]);//En cada vuelta mostramos un elemento del array.
  }
```

Dentro del paréntesis escribimos en primer lugar la variable que controla el bucle `i` seguida de la palabra reservada `in` y el nombre del array

La variable `i` toma un valor inicial de `0`, en cada vuelta aumenta en una unidad y comprueba si existe el elemento del array correspondiente `miarray[i]`. Si no existe detiene el bucle.

En cada repetición podemos obtener el elemento correspondiente mediante su código. `miarray[i].`

3.4.3. Estructura while

La estructura `while` forma un bucle en el cual el código indicado se repite mientras se cumpla una determinada condición. Su sintaxis es:

```
while( /*condicion*/ ) {
   //Código que se repite
   }
```

Escribimos en primer lugar la palabra reservada `while` seguida de la condición entre paréntesis. Después entre llaves ponemos las sentencias que deben repetirse.

Para que esta estructura funcione correctamente dentro de la condición debe estar la variable o el elemento que controla el bucle, el cual debe estar definido anteriormente. En el código que se repite debe cambiar en cada vuelta la variable o elemento de control, de manera que haya un momento en el cual la condición no se cumpla.

3.4.4. Estructura do while

Como una variante de la estructura **`while`** tenemos la etructura **`do while`** con la siguiente sintaxis:

```
do { //Código que se repite
   }
while( /*condicion*/ )
```

Aquí, aparte de usar la palabra reservada do, la condición se indica al final. Esto hace que el código que se indica dentro de la estructura se ejecuta siempre al menos una vez, aunque la condición no se cumpla.

Por lo demás funciona igual que la estructura while.

4. Acceso al DOM

4.1. El DOM o Document Object Model

El DOM Es el Modelo de Objeto de Documento, es decir, es un modelo de estructura que deben seguir todas las páginas Web.

Aunque estas se escriben de manera secuencial, los distintos elementos que componen la página siguen un modelo jerárquico, de manera que dependen unos de otros. Estos elementos se llaman también objetos:

En la página hay un objeto inicial (elemento padre), el objeto *document* (etiqueta HTML). De él dependen dos elementos secundarios (elementos hijos) que son las etiquetas *head* y *body*. De éstas parten todos los demás elementos.

El objeto *document* depende a su vez del objeto *window* que es el navegador. El objeto *window* tiene también otros elementos hijos además del *document*.

4.2. Objetos, métodos y propiedades

Concepto	Definición
Objetos	Cualquier cosa que pueda guardarse en una variable, desde simples datos hasta elementos complejos (arrays, funciones, fechas, etc.)
Métodos	Forma de trabajar: funciones predefinidas. Para aplicar un método debemos llamar a la función que lo contiene
Propiedades	Elementos que definen o cambian características de un objeto. Pueden ser de sólo lectura o de lectura y escritura. La propiedad puede ser a su vez un objeto del que dependan otras propiedades.

Acceso desde javascript: Javascript utiliza el operador punto (`.`) para acceder a los diferentes métodos y propiedades de un elemento.

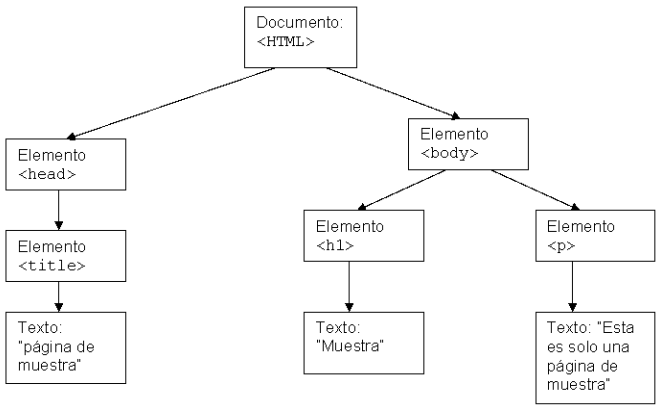
```
n=miarray.lenght
```

```
n=fecha.getDate()
```

Debemos distinguir entre dos tipos de objetos, los objetos de javascript propiamente dicho, y los objetos del DOM. En este apartao nos referiremos exclusivamente a los objetos del DOM

4.3. Estructura del DOM

El DOM es la estructura de la página, la cual está construida de manera **jerárquica**, al depender unos elementos de otros.



Cada elemento se denomina **nodo** y estos dependen unos de otros de manera jerárquica, de modo que cada nodo tiene su **nodo padre** y puede tener **nodo hijos**.

Objeto inicial: es el objeto del que depende todo. Este es el objeto **window**, que es el navegador. Como se sobreentiende no hay que escribirlo, por lo que la página depende del objeto **document**, hijo del anterior. El objeto **document** marca el inicio de la página, y se corresponde con la etiqueta `<html>`

4.4. Tipos de nodos

Aunque existen 12 tipos de nodos, en realidad, para manipular las páginas web sólo necesitamos los 5 siguientes:

Tipos de nodos en HTML

`document`	Cada una de las etiquetas HTML
`element`	Nodo raíz del que derivan los demás
`attr`	Cada uno de los atributos de las etiquetas HTML
`text`	Texto encerrado en las etiquetas HTML
`comment`	Comentarios de la página HTML

Los nodos `attr` y `text` son siempre hijos del nodo `element` (etiqueta) que lo contiene.

Solamente los nodos `document` y `element` pueden tener nodos hijo.

Los nodos `comment` son los comentarios que el programador pone en la página, y son hijos del nodo `element` en el que están escritos.

4.5. Lectura de elementos de la página.

Se accede a los elementos de la página mediante los siguientes métodos. Todos ellos van precedidos siempre de `document.`.

`getElementsByTagName("etiqueta")`	Crea un array con todas las etiquetas cuyo nombre sea `"etiqueta"`
`getElementsByName("valor")`	Crea un array con todas las etiquetas cuyo atributo name sea `"valor"`
`getElementById("valor")`	Accede al nodo cuyo atributo `id` sea `"valor"`

4.6. Crear una nueva etiqueta

Pasos para crear una nueva etiqueta

Paso	Código y explicación	
1º	**Código**	`var escribir="nuevo texto."`
	Explicación	Variable con el nuevo texto
2º	**Código**	`var etiqueta = document.createElement("tag")`
	Explicación	Crear el nodo de la etiqueta. `"tag"`=etiqueta HTML.
3º	**Código**	`var texto = document.createTextNode(escribir)`
	Explicación	Crear el nodo de texto. Como argumento pasamos el texto, ya sea en una variable o directamente entre comillas.
4º	**Código**	`var nuevoElemento = etiqueta.appendChild(texto)`
	Explicación	Se inserta el nodo de texto como hijo del nodo de etiqueta.

Estos pasos muestran cómo crear un nodo *element* (de etiqueta) que contiene un nodo *text* (de texto). Sin embargo esta estructura no está integrada en la página, sino que solamente existe en una variable. El siguiente paso es integrar la estructura en la página.

4.7. Insertar o reemplazar un nodo en la página.

Siguiendo con el ejemplo anterior, una vez creado el nodo de etiqueta con su texto, debemos insertarlo en la página o remplazarlo por otro ya existente. Veamos las opciones:

4.7.1. Insertar después de un elemento

Insertar el nodo al final de un elemento padre: método `appendChild()`.

insertar mediante "appendChild()"

Paso	Código y explicación	
5º	**Código**	`document.getElementById("caja1").appendChild(nuevoElemento)`
	Explicación	Especificamos el nodo (etiqueta) de la página en el que queremos introducir el nuevo elemento -`document.getElementById(("..")-` e insertamos el elemento como hijo. El nuevo elemento se colocará detrás de los ya existentes en ese nodo.

4.7.2. Insertar antes de un elemento.

Insertar el nuevo elemento antes de un elemento concreto de la página. Método `insertBefore`.

insertar mediante "insertBefore()"

Paso	Código y explicación	
5º	**Código**	`var referencia = document.getElementById("ref")`
	Explicación	Guardamos en una variable el nodo de referencia para poder insertarlo antes de éste.
6º	**Código**	`var padre = referencia.parentNode`
	Explicación	En una variable guardamos el nodo padre del nodo de referencia, localizado mediante la propiedad `parentNode`.
7º	**Código**	`padre.insertBefore(nuevoElemento,referencia)`
	Explicación	Desde el elemento padre insertamos el nuevo elemento con el método `insertBefore(.. , ..)` donde el primer argumento será el elemento a insertar, y el segundo el elemento de referencia.

4.7.3. Reemplazar un elemento

Para remplazar el nuevo elemento por otro ya existente utilizamos el método `replaceChild`.

Remplazar mediante "replaceChild()"

Paso	Código y explicación	
5º	**Código**	`var viejoElemento = document.getElementById("remplazar")`
	Explicación	En una variable guardamos el nodo que queremos remplazar.
6º	**Código**	`var padre = viejoElemento.parentNode`
	Explicación	En una variable guardamos el nodo padre del nodo que queremos remplazar, localizado mediante la propiedad `parentNode`.
7º	**Código**	`padre.replaceChild(nuevoElemento,viejoElemento)`
	Explicación	Desde el elemento padre remplazmos el nuevo elemento con el método `replaceChild(nuevo,viejo)` donde el primer argumento será el elemento a insertar, y el segundo el elemento a remplazar.

4.8. Eliminar un nodo

Para eliminar un nodo utilizamos el método `removeChild()`

Eliminar mediante "removeChild()"

Paso	Código y explicación	
1º	**Código**	`var suprimir = "document.getElementById("provisional")`
	Explicación	En una variable guardamos el nodo que queremos eliminar.
2º	**Código**	`suprimir.parenNode.removeChild(suprimir)`
	Explicación	Localizamos primero su elemento padre `parentNode` y desde ahí lo eliminamos mediante el método `removeChild` pasándolo como argumento.

4.9. Cambiar un nodo de sitio

Para cambiar un elemento de sitio seguimos los mismos pasos que para eliminarlo (`removeChild()`). El elemento, aunque está eliminado, ha quedado guardado en una variable. Esto permite recuperarlo mediante el método `appendChild()`.

Trasladar mediante "removeChild()" y "appendChild()"

Paso	Código y explicación	
1º	**Código**	`var trasladar = "document.getElementById("elemento1")`
	Explicación	En una variable guardamos el nodo que queremos trasladar.
2º	**Código**	`trasladar.parenNode.removeChild(trasladar)`
	Explicación	Eliminamos de la página el elemento que queremos trasladar, sin embargo éste sigue guardado en la variable.
3º	**Código**	`document.getElementById("caja1").appendChild(trasladar)`
	Explicación	Insertamos el elemento en otro nodo mediante el método `appendChild` tal como haríamos con un elemento nuevo.

También podemos insertar el elemento mediante el método `insertBefore()` siguiendo los pasos vistos anteriormente para insertar con este método.

4.10. Copiar un nodo

Para copiar un elemento en otro lugar de la página, conservando el original, utilizamos el método `cloneNode(true)`.

Copiar mediante "cloneNode(true)"

Paso	Código y explicación	
1º	**Código**	`var elemento1 = "document.getElementById("elem1")`
	Explicación	En una variable guardamos el nodo que queremos copiar.
2º	**Código**	`var copia = elemento1.cloneNode(true)`
	Explicación	Hacemos una copia mediante el método `cloneNode(true)`, la cual queda guardada en una variable.
3º	**Código**	`document.getElementById("caja1").appendChild(copia)`
	Explicación	Copiamos el elemento en otro nodo mediante el método `appendChild` tal como haríamos con un elemento nuevo.

También podemos utilizar el método `insertBefore()` siguiendo los pasos vistos anteriormente para insertar con este método.

4.11. Propiedad "innerHTML"

La propiedad `innerHTML` permite leer y escribir el código HTML que hay dentro de una etiqueta. Esto permite escribir directamente el contenido HTML que habrá dentro de una etiqueta, es decir podemos escribir directamente etiquetas anidadas.

4.11.1.Lectura:

Paso	Código y explicación	
1º	**Código**	`var contenido = document.getElementById("elem1").innerHTML`
	Explicación	Guarda en una variable el contenido HTML del elemento.

Aquí hemos usado el método `getElementById()` para acceder al elemento, pero podemos usar otros métodos o utilizar variables en las que hayamos guardado variables del tipo *element*.

4.11.2. Escritura

Paso	Código y explicación	
1º	**Código**	`document.getElementById("elem1").innerHTML = "<a href = 'http://google.com'>Ir a Google</a>"`
	Explicación	Localizamos el elemento en la página mediante `getElementById()` , aplicamos después la propiedad `innerHTML` a la cual le damos como valor el contenido HTML que queremos que haya dentro de esa etiqueta.

La operación de escritura borra el contenido antiguo que tenía el elemento al que se le aplica, y lo remplaza por el que le hemos dado.

4.11.3. Añadir contenido a un elemento

Podemos escribir más contenido dentro de una etiqueta sin borrar el anterior utilizando el operador +=.

Paso	Código y explicación	
1º	**Código**	`var contenido = document.genElementById("elem1")`
	Explicación	Guardamos el elemento en una variable, para poder manejarlo mejor.
2º	**Código**	`contenido.innerHTML += "<p>añadir otro párrafo</p>"`
	Explicación	Aplicamos la propiedad `innerHTML` con el operador de suma y asignación `+=`, de esta manera el código que se escribe se añade al que ya tenía el elemento.

4.12. Acceso a atributos

4.12.1. Acceso como propiedades

Los atributos son en realidad propiedades del nodo *element* (etiqueta) que los contiene, por lo que podemos acceder a ellos de la misma manera que se accede a un propiedad de un elemento, es decir mediante la forma `.nombre_atributo`. Podemos acceder tanto a su lectura como a su escritura

Lectura del atributo

Paso	Código y explicación	
1º	**Código**	`var valor = document.getElementById("enlace1").href`
	Explicación	Para leer el valor del atributo localizamos el elemento que lo contiene y lo escribimos detrás como una propiedad. La variable `valor` será el valor que tenga el atributo.

Escritura de atributo

Paso	Código y explicación	
1º	**Código**	`document.getElementById("enlace1").href = "http://yahoo.es"`
	Explicación	Localizamos el nodo que contiene el atributo y lo escribimos detrás como una propiedad. Le asignamos un nuevo valor. Éste podemos escribirlo directamente o en una variable.

Al escribir un atributo, si éste ya existía, el valor antiguo se remplaza por el nuevo. Si no existía se crea el nuevo atributo con el valor asignado.

4.12.2. Acceso con métodos

Podemos acceder a los atributos mediante los métodos `getAttribute()` (lectura) y `setAttribute()` (escritura).

Lectura de atributo

Paso	Código y explicación	
1º	**Código**	`valor = document.getElementById("elem1").getAttribute("align")`
	Explicación	Localizamos el nodo que contiene el atributo y le aplicamos el método `getAtribute("..")` . Como argumento pasamos el nombre del atributo. Obtenemos así su valor

Escritura de atributo

Paso	Código y explicación	
1º	**Código**	`document.getElementById("elem1").setAttribute("align","center")`
	Explicación	Localizamos el nodo que contiene el atributo y le aplicamos el método `setAtribute("..","..")` . Como primer argumento pasamos el nombre del atributo, y como segundo su valor.

Al escribir un atributo, si éste ya existía, el valor antiguo se remplaza por el nuevo. Si no existía se crea el nuevo atributo con el valor asignado.

4.13. Acceso al código CSS

Para acceder al código CSS utilizamos las propiedades `.style.propiedad`CSS.

4.13.1. Acceso a CSS: Escritura

Paso	Código y explicación	
1º	**Código**	`document.getElementById("elem1").style.fontSize = "1.5em"`
	Explicación	Localizamos el nodo al que queremos aplicar la propiedad. Ponemos después la propiedad `.style` seguido de un punto y la propiedad CSS; le asignamos después un nuevo valor a la propiedad.

Si la propiedad ya existía antes se le cambia su valor, y si no existía se crea como nueva con el valor indicado.

Si el nombre de la propiedad tiene más de una palabra (separadas por guiones en CSS), lo escribiremos todo junto y empezando por mayúscula la segunda palabra y siguientes.

Escribiremos los valores entre comillas (a no ser que esté guardado en una variable), y al igual que en CSS en las medidas indicaremos el tipo. Solo en el caso de que el valor sea un número (sin indicación de medida) puede escribirse sin comillas.

4.13.2. Acceso a CSS: Lectura

Sólo se puede acceder a la lectura de una propiedad CSS si ésta ha sido escrita con javascript en la forma indicada anteriormente.

Paso	Código y explicación	
1º	**Código**	`var valor =document.getElementById("elem1").style.fontSize`
	Explicación	Localizamos el nodo que contiene la propiedad y aplicamos las mismas propiedades vistas anteriormente; pero aquí no le damos un nuevo valor, sino que guardamos el valor que tiene en una variable.

4.14. Propiedades tipo array

El DOM tiene una serie de propiedades de tipo array que nos devuelven un array con todos los elementos que tienen una serie de características. Estas son:

Propiedades tipo array

Propiedad	Explicación
document.links	Devuelve un array con todos los **enlaces** que hay en el documento
document.forms	Devuelve un array con todos los **formularios** del documento
document.images	Devuelve un array con todas las **imágenes** del documento
document.anchors	Devuelve un array con todos los enlaces tipo **referencia** `<a name="..">...</a>` del documento

En los siguientes apartados veremos este tipo de propiedades:

4.15. Acceso a enlaces

Mediante `document.links` accedemos a un array que contiene todos los enlaces de la página.

```
enlaces = document.links
```

El orden de los enlaces en el array es el mismo de aparición en el código de la página. Accedemos a cada enlace mediante su número en el array:

```
enlace1=enlaces[0]
```

Después podemos acceder a la ruta o al texto tanto en modo lectura como escritura:

- **Ruta lectura** : `var ruta = document.links[0].href`
- **Texto lectura** : `var texto = document.links[0].innerHTML`

- **Ruta escritura** : `document.links[0].href = "http://es.yahoo.com"`
- **Texto escritura** : `document.links[0].innerHTML = "Ir a Yahoo"`

Esto podemos aplicarlo también para las demás propiedades de tipo array, en las cuales podemos buscar los elementos por su posición.

Además en las propiedades de tipo array si la etiqueta HTML lleva el atributo `name` podemos usar su valor para buscarlo en el array:

Lenguaje	Código
HTML	`<a href="http://google.es" name="buscador">ir a Google</a>`
Javascript(1)	`ruta = document.links["buscador"].href`
Javascript(2)	`ruta = document.links.buscador.href`

El valor del atributo `name` es la referencia para buscar el enlace en el array.

Sin embargo, con los enlaces (`document.link`) la búsqueda por el atributo `name` no funciona en Internet Explorer, por lo que si queremos que se vea también en este navegador debemos hacerlo por el número de posición en el array.

4.16. Acceso a imágenes

Podemos acceder a todas las imágenes de la página mediante el array `document.images`

Una vez hemos accedido al array, podemos acceder a cada una de ellas mediante el número de posición en la página o mediante su atributo `name`, de la misma manera que hemos visto para los enlaces. El acceso mediante `name` sí que funciona en todos los navegadores, incluido Internet Explorer.

Ejemplo: Supongamos que la siguiente imagen con el siguiente código HTML es la primera de la página:

```
<img src="micoche.gif" alt="Este es mi coche" name="imagenCoche">
```

Podemos acceder a la imagen de cualquiera de estas tres formas:

- `micoche = document.images[0];`
- `micoche = document.images["imagenCoche"];`
- `micoche = document.images.imagenCoche;`

Una vez que accedemos a la imagen podemos acceder a cualquiera de sus atributos, tanto para lectura como para escritura.

Código	Explicación
`var ruta = micoche.src;`	Lee la ruta de la imagen
`micoche.src = "nuevoCoche.gif";`	Cambia la ruta de la imagen y por tanto la imagen que visualizamos.
`var comentario = micoche.alt;`	Lee el atributo alt.
`micoche.alt = "Mi coche nuevo";`	cambia el valor del atributo alt.

4.17. Acceso a enlaces de referencia

El acceso a los enlaces de referencia se hace mediante el array `document.anchors`.

Para acceder a los elementos de este array podemos hacer igual que con los arrays anteriores.

Al igual que sucede con el acceso a enlaces (`document.links`) el acceso mediante el atributo `name` no funciona con Internet Explorer. Veamos un ejemplo en el que el enlace de referencia tiene el siguiente código HTML y es el primero de la página:

```
<a name="ref1">Ir a sección primera</a>
```

Acceso para todos los navegadores:

- `seccion1 = document.anchors[0];`

Acceso a todos los navegadoes excepto Internet Explorer:

- `seccion1 = document.anchors["ref1"];`
- `seccion1 = document.anchors.ref1;`

Una vez hemos accedido al elemento podemos acceder a sus atributos o al contenido del texto mediante `innerHTML`.

4.18. Acceso a formularios.

El acceso a los formularios de la página se hace mediante el array `document.forms`

Esto crea un array con los distintos formularios de la página. Para acceder a cada uno de ellos se procede como en los casos anteriores. En la etiqueta `form` ponemos el atributo name:

```
<form action="#" method="post" name="datos" > /* ... */ </form>
```

Suponiendo que sea éste el primer formulario de la página podemos acceder de cualquiera de estas formas:

- `form1 = document.forms[0];`
- `form1 = document.forms["datos"];`
- `form1 = document.forms.datos;`
- `form1 = document.datos;`

4.18.1. Acceso a los campos del formulario

Dentro de cada formulario se crea un array `elements` con los elementos o campos que éste contiene. Podemos acceder a los campos del formulario mediante su posición en el array o mediante el atributo `name`:

Por ejemplo en el formulario anterior ponemos el primer campo:

```
Nombre: <input type="text" name="nombre" />
```

Podemos acceder a él de cualquiera de estas maneras:

- `elem1 = form1.elements[0];`
- `elem1 = form1.elements["nombre"];`
- `elem1 = form1.nombre`

4.18.2. Propiedades comunes a los campos de formulario

Una vez obtenido el elemento hay varias propiedades comunes que pueden sernos útiles:

Propiedades comunes a elementos de formulario

Propiedad	Ejemplo de código	Explicación
type	`tipo = elem1.type`	Indica el tipo de elemento. Normalmente coincide con el valor del atributo `type`. En los "textarea" su valor es `textarea`. En los "select" su valor puede ser `select-one` o `select-multiple` (sólo lectura).
form	`formulario1 = elem1.form`	Se accede al elemento padre o formulario (sólo lectura).
name	`valorName = elem1.name`	Se accede al valor del atributo `name` (sólo lectura).
value	`texto = elem1.value`	Permite leer y escribir el atributo `value`. En campos de texto ("text", "password", "textarea"), accedemos al texto escrito por el usuario. En campos de tipo botón ("submit", "reset", "button"), accedemos al texto escrito en el botón.

4.18.3. Acceso a campos de texto

En los campos de tipo texto se accede al texto escrito por el usuario mediante la propiedad `.value`, tal como se ha visto antes.

4.18.4. Acceso a botones radio

En un grupo de botones radio llevan todos ellos el mismo atributo `name`.

```
<form action="#" name="elegir">
<input type="radio" value="red" name="color" /> Rojo <br/>
<input type="radio" value="green" name="color" /> Verde <br/>
<input type="radio" value="blue" name="color" /> Azul <br/>
<input type="radio" value="black" name="color" /> Negro <br/>
</form>
```

El acceso mediante el atributo `name` crea un array en el que están todos los botones radio:

```
botones = document.elegir.color
```

Después con la propiedad `checked` aplicada a cada botón nos dirá si éste está seleccionado o no (devuelve `true` o false).

Mediante un bucle recorremos los elementos del array para comprobar cual está seleccionado. El código completo para el ejemplo anterior es el siguiente:

```
botones = document.elegir.color;
for (i=0; i<botones.length; i++) {
    valor = botones[i].checked;
    if (valor == true) {
        elegido = botones[i];
        }
    }
pulsado=elegido.value
```

En el código anterior obtenemos el botón pulsado en la variable `elegido`. Después mediante la propiedad `value` podemos distinguirlo de los demás.

4.18.5. Acceso a botones checkbox

Accedemos al botón checkbox mediante el atributo `name`, de la misma manera que a los demás elementos.

Después la propiedad `checked` nos dirá si el botón está seleccionado (`true`) o no (`false`). Ejemplo:

```
<form action="#" name="condiciones">
<p><input type="checkbox" name="aceptar" /> Acepto las condiciones.</p>
</form>
```

Comprobamos el botón mediante el siguiente código javascript:

```
function compruebaBoton() {
    boton=document.condiciones.aceptar;
    if (boton.checked==true) {
       acepto="si";
       }
    else {
       acepto="no";
       }
    return acepto;
    }
```

Tal como hemos hecho aquí lo normal es poner el código dentro de una función para llamarla cuando tengamos que comprobar el botón.

4.18.6. Acceso a listas desplegables simples

Lo que queremos saber al acceder a una lista desplegable es qué opción ha elegido el usuario. Para ello accedemos primero a etiqueta `select`, la cual llevará el atributo `name`. Tenemos por ejemplo el siguiente formulario.

```
<form action="#" name="elegir">
<select name="color">
  <option value="red">Rojo</option>
  <option value="green">Verde</option>
  <option value="blue">Azul</option>
  <option value="black">negro</option>
</select>
</form>
```

Accedemos primero a la etiqueta select:

```
lista = document.elegir.color
```

Mediante la propiedad `options` obtenemos el array con las opciones:

```
opciones = lista.options;
```

Mediante la propiedad `selectedIndex` obtenemos el número que ocupa en el array el primer elemento seleccionado.

```
num = lista.selectedIndex;
```

Ahora podemos acceder en el array al elemento seleccionado:

```
seleccionado = opciones[num];
```

Una vez tenemos el elemento podemos acceder al atributo value:

```
valor = seleccionado.value
```

También podemos acceder al texto de la etiqueta `option` mediante la propiedad `text`.

```
texto = seleccionado.text
```

Podemos resumir todos estos pasos mediante un código en el que los enlazamos:

```
valor = lista.options[lista.selectedIndex].value
```

4.18.7. Acceso a listas de selección múltiple

Para acceder a las listas de selección múltiple seguimos los mismos pasos que para las de selección simple hasta obtener el array de opciones.

Una vez obtenido el array de opciones, la propiedad `selectedIndex` no nos sirve, ya que solamente nos daría la primera opción seleccionada.

Sin embargo tenemos la **propiedad `.selected`**, que aplicada a los elementos del array nos devuelve `true` si éste está seleccionado, o `false` si no lo está. Para encontrar los elementos seleccionados recorreremos los elementos del array en un bucle, aplicando a cada uno de ellos la propiedad `.selected`. Con esto podemos encontrar los elementos que están seleccionados.

4.19. El objeto window

El objeto window o navegador tiene también sus métodos y propiedades. Podemos verlos todos en el *Manual de Dom*. Sin embargo aquí veremos aquellos con los que podemos realizar acciones en javascript.

4.19.1. Ventanas de alerta

Son aquellas en las que sale un mensaje en el que se alerta sobre algo de la página. Su apertura se realiza mediante los siguientes métodos del objeto window:

Método	**Ejemplo**	Explicación
alert()	`alert("hola mundo")`	Ventana de alerta que muestra el argumento.
prompt()	`nombre = prompt("como te llamas")`	Se muestra el argumento y un cuadro de dialogo para que el usuario escriba la respuesta. Ésta es guardada en una variable.
confirm()	`elige = confirm("acepta o cancela")`	Se muestra el argumento y dos botones de "aceptar" y "cancelar". al pulsar en los botones obtenemos en la variable los valores `true` o `false`.

4.19.2. Ventanas emergentes

Son aquellas que abren otra página en una ventana aparte. Éstas dependen del método `open()` de window.

El método tiene tres argumentos que son:

```
open("URL","nombre","propiedades");
```

El primer argumento es la ruta o URL hacia la página que queremos abrir.

El segundo argumento es un nombre arbitrario que le daremos a la ventana, por ejemplo `"ventana1"`.

El tercer argumento es opcional, y es una lista de propiedades que puede tener la ventana. Éstas se escribirán seguidas y separadas por comas. Por ejemplo.

```
open("http://google.com","Google","width=400,height=250,toolbar=yes")
```

Veamos cuales son las propiedades que podemos poner en el tercer argumento:

propiedad	Explicación
toolbar=yes/no	Mostrar o no la barra de herramientas.
statusbar=yes/no	Mostrar o no la barra de estado.
titlebar=yes/no	Mostrar o no la barra de título.
menubar=yes/no	Mostrar o no la barra de menús.
scrollbars=yes/no	Mostrar o no las barras de desplazamiento
resizable=yes/no	Establece si se puede redimensionar el tamañó de la ventana. Sólo funciona en Internet Explorer.
width=num	Ancho de la ventana en pixels (num = número de pixels).
height=num	Alto de la ventana en pixels (num = número de pixels).
top=num	Distancia en pixels (num) desde el borde superior de la pantalla al borde superior de la ventana.
left=num	Distancia en pixels (num) desde el borde izquierdo de la pantalla al borde izquierdo de la ventana.

4.19.3. Historial de páginas

Podemos ir hacia atrás o hacia adelante en las páginas que hemos visitado anteriormente mediante el método `history.go()` de window.

Como argumento pasamos el número de páginas que queremos avanzar o retroceder. En este último caso el número será negativo. Por ejemplo:

```
history.go(-1);
```

Este código nos llevará a la página visitada antes de la actual.

4.19.4. Temporizadores

Dependiendo del objeto window hay una serie de métodos que nos permiten retrasar el tiempo en que se ejecuta una función, o hacer que ésta se repita a intervalos regulares de tiempo. Son los temporizadores.

```
setTimeout("funcion","tiempo")
```

Este método hace que una función se ejecute después de pasado un tiempo determinado. Para ello en el primer argumento se escribe el nombre de la función (sin los paréntesis), y como segundo el número de milisegundos de retardo.

```
setInterval("funcion","tiempo")
```

Este método hace que una función se ejecute repetidamente a intervalos regulares de tiempo. Para ello en el primer argumento se escribe el nombre de la función (sin los paréntesis), y como segundo el número de milisegundos entre cada intervalo.

Para parar los temporizadores puestos en marcha con los métodos anteriores haremos lo siguiente:

En primer lugar debemos guardar el método en una variable de referencia:

```
temp1=setInterval("repetir","3000");
```

El código anterior repite la función `repetir()` (la cual debemos haber creado previamente), cada 3 segundos. Para pararla usaremos el método `clearInterval()` de la siguiente manera:

```
clearInterval(temp1);
```

De la misma manera haremos con el método `setTimeout()`. Para detener la espera y que la función indicada no se ejecute, guardamos el método en una variable:

```
temp2=setTimeout("mostrar","10000")
```

En este ejemplo la función `mostrar` se ejecutará diez segundos después de haber leído el código. Si queremos pararla antes, para que no se ejecute, javascipt debe leer el siguiente código:

```
clearTimeout(temp2);
```

4.19.5. Convertir texto en código

El método `eval()`, convierte el texto que le pasemos en el argumento en código javascript y lo ejecuta. por ejemplo :

```
codigo="alert='hola mundo';";
eval(codigo);
```

El resultado será la ventana de alerta con el texto "hola mundo", ya que hemos convertido una cadena de texto en código javascript.

4.19.6. Redireccionar páginas

La propiedad `location.href` nos devuelve la URL o dirección de la página en la que estamos.

Si cambiamos la URL de la página, ésta se redireccionará a la página indicada

```
location.href="http://google.com";
```

Este código hace que la página se redireccione hacia Google.

4.19.7. Otros métodos y propiedades de window

Vemos aquí otros métodos y propiedades del objeto window que pueden ser interesantes para trabajar con javascript:

Método o propiedad	Explicación
document.bgColor;	Acceso o modificación del color de fondo de la página.
document.fgColor;	Acceso o modificación del color del texto de la página.
document.title	Acceso o modificación del título de la página (el que aparece en la pestaña del navegador).
document.lastModified	Acceso a la fecha de la última modificación de la página.
navigator.userAgent	Acceso a la información sobre el navegador y el sistema operativo que está usando el usuario.
screen.width	Acceso al ancho de resolución de pantalla que está usando el usuario.
screen.height	Acceso al alto de resolución de pantalla que está usando el usuario.

5. Eventos

5.1. Definición

Evento es todo lo que sucede en la página y que pueda ser detectado por javascript. Incluimos aquí los procesos que ocurren cuando se abre o cierra la página, lo que está haciendo el usuario, etc.

Con javascript se pueden detectar cuando se produce un cambio o evento en la página y a partir de ahí hacer que se aplique un determinado código cuando el evento se produce. De esta forma se puede interactuar con el usuario, el cual al provocar el evento desencadena una serie de acciones en la página.

5.2. Lista de eventos

El nombre del evento se construye con el prefijo "on" seguido del nombre en inglés de la acción que desencadena el evento.

Esta es una lista de eventos que funcionan en todos los navegadores:

Eventos

Evento	Descripción	Elementos a los que se aplica
onblur	Deseleccionar el elemento	`<button>, <input>, <label>, <select>, <textarea>, <body>`
onchange	Deseleccionar un elemento que se ha modificado	`<input>, <select>, <textarea>`
onclick	Pulsar y soltar el ratón	`Todos los elementos`
ondblclick	Pulsar y soltar el ratón dos veces seguidas	`Todos los elementos`
onfocus	Seleccionar un elemento (con el ratón o con el tabulador)	`<button>, <input>, <label>, <select>, <textarea>, <body>`
onkeydown	Pulsar una tecla (sin soltar)	`elementos de formulario y <body>`
onkeypress	Pulsar una tecla	`elementos de formulario y <body>`

Evento	Descripción	Elementos a los que se aplica
onkeyup	Soltar una tecla pulsada	elementos de formulario y <body>
onload	La página se ha cargado completamente	<body>
onmousedown	Pulsar (sin soltar) un botón del ratón	todos los elementos
onmousemove	Mover el ratón	todos los elementos
onmouseout	El ratón "sale" del elemento (pasa por encima de otro elemento)	todos los elementos
onmouseover	El ratón "entra" en el elemento (pasa por encima del elemento)	todos los elementos
onmouseup	Soltar el botón que estaba pulsado en el ratón	todos los elementos
onreset	Inicializar el formulario (borrar todos sus datos)	<form>
onresize	Modificar el tamaño de la ventana del navegador	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar un formulario	<form>
onunload	abandonar o cerrar la página	<body>

5.3. Clasificación de eventos

Dependiendo de la manera de interactuar en la página podemos clasificarlos en:

- **Eventos del ratón:** Aquellos en los que el usuario utiliza el ratón para provocarlos.
- **Eventos de teclado:** Aquellos en los que el usuario utiliza el teclado para provocarlos.
- **Eventos de página:** Aquellos en los que cambia el estado de la página (se carga, se descarga, se redimensiona, etc.)
- **Eventos de formulario:** Aquellos en los que se cambia el estado de algún elemento de un formulario.

5.4. Interactuar con enlaces

Podemos usar un enlace para interactuar con javascript. Al pulsar el enlace nos envía a un código javascript el cual se ejecutará.

Para ello la ruta del enlace será como en el siguiente ejemplo:

```
<a href="javascript: mifuncion();">iniciar mi funcion</a>;
```

Escribimos en el atributo `href` la palabra `javascript` seguida por dos puntos, y después la del código javascript; normalmente la llamada a la función que queremos ejecutar.

5.5. Insertar eventos

Tenemos tres maneras de insertar los eventos en la página:

5.5.1. En la etiqueta HTML directamente

En la etiqueta de HTML que debe producir el evento insertamos el evento como un atributo de la misma. El valor del atributo es el código javascript que queremos que se produzca al producirse el evento.

```
onclick=”alert('hola mundo')”;
```

5.5.2. En la etiqueta HTML con una función

Igual que el anterior, pero el código javascript consiste en llamar a una función que contiene el código que se va a ejecutar:

```
onclick="mifuncion();"
```

5.5.3. En el propio código javascript

Primero creamos la función donde estará el código a ejecutar por el evento:

```
function mifuncion() { /*...*/}
```

Después buscamos en la página el elemento al que se le aplica el evento, para incorporar el evento como una propiedad:

```
document.getElementById("elem1").onclick = mifuncion
```

Como valor de la propiedad pasamos el nombre de la función (sin paréntesis).

5.6. Función window.onload =function()

Muchas veces necesitamos que la página ya se haya cargado completamente antes de ejecutar un código javascript. Tal es el caso de la búsqueda de elementos del DOM en la página. Utilizamos para ello el evento `onload` al que le aplicamos una función anónima:

```
window.onload = function () {
/*código que se ejecutará después de cargarse la página completamente*/
}
```

La lectura de la página se realiza de manera secuencial, por lo que si llamamos a un elemento del DOM antes de que éste se haya cargado, javascript no lo encontrará, ignorando el código; sobre todo si ponemos el código al principio de la página.

Para evitar esto debemos poner las llamadas a elementos del DOM y el código que necesite que la página esté cargada dentro de esta función anónima.

5.7. El elemento this

Este elemento o palabra clave hace referencia al mismo elemento HTML en el que está el evento; de manera que no lo tengamos que buscar en el DOM. Según la forma de insertar el evento el código será distinto:

Si insertamos el evento **directamente en el HTML** pondremos "this" para referirnos al propio elemento:

```
<p onclick="this.style.color='green';">Pulsa para cambiar el color</p>
```

Si insertamos el código **en el HTML mediante una función** el elemento "this" debemos pasarlo como argumento de la función:

```
<p onclick="mifuncion(this);">Cambia el color del texto</p>
```

Recogemos en la función el argumento para usarlo como el acceso al propio elemento del DOM:

```
function mifuncion(elem) {
    elem.style.color="green";
    }
```

Si insertamos el código **directamente en javascript**, podemos utilizar el elemento "this" dentro de la función, por ejemplo creamos primero la función.

```
function mifuncion() {
    this.style.color="green";
    }
```

Después provocamos el evento con el código:

```
document.getElementById("texto1").onclick = mifuncion;
```

5.8. El objeto event

Una vez producido un evento a veces necesitamos tener más información sobre el mismo. Para ello utilizamos el objeto "event".

La mayoría de los navegadores obtienen este objeto al pasarlo como un argumento implícito a la función que es llamada por el evento. Sin embargo Internet Explorer la considera como una propiedad del objeto "window", por lo que el código es distinto.

El código compatible con todos los navegadores es el siguiente:

```
function manejaEventos(elEvento) {
   var evento = elEvento || window.event;
   /* código para extraer la información del evento  */
   }
```

Obtenemos así el objeto "event" en la variable `evento`, el cual tiene una serie de propiedades que nos darán la información sobre el evento.

5.9. Información del evento

5.9.1. Tipo de evento

La propiedad `.type` aplicada al objeto "event" nos dará la información del tipo de evento que se produce:

```
function manejaEventos(elEvento) {
   var evento = elEvento || window.event;
   tipo = evento.type;
   }
```

La variable `tipo` nos da el nombre del evento que se ha producido, pero sin el prefijo `on`. Por ejemplo si el evento es `onclick`, el valor de `type` será `"click"`.

5.9.2. Información sobre el ratón

La información que más nos puede interesar sobre el ratón es la posición del mismo, es decir las coordenadas del puntero del ratón en la página.

Coordenadas del ratón

Una vez obtenido el objeto "event" (en `evento`), hay varias propiedades para obtener las coordenadas del ratón. Todas ellas se miden en píxeles. Dependiendo del origen de coordenadas tenemos:

Respecto a la pantalla completa: Origen de coordenadas en la esquina superior izquierda de la pantalla:

- `var ooordX = evento.screenX;` : Distancia horizontal al borde izquierdo de la pantalla.
- `var ooordY = evento.screenY;` : Distancia vertical al borde superior de la pantalla.

Respecto a la ventana del navegador: Origen de coordenadas en la esquina superior izquierda de la parte visible de la página (ventana del navegador donde se muestra la página)

- `var ooordX = evento.clientX;` : Distancia horizontal al borde izquierdo de ventana del navegador.
- `var ooordY = evento.clientY;` : Distancia vertical al borde superior de la ventana del navegador.

Respecto al origen de la página: Origen de coordenadas en la esquina superior izquierda de la propia página web (ésta puede estar desplazada y no verse completa en el navegador, por lo que puede no coincidir con el anterior).

- `var ooordX = evento.pageX;` : Distancia horizontal al borde izquierdo de la página.
- `var ooordY = evento.pageY;` : Distancia vertical al borde superior de la página.

Esta opción no es compatible con Internet Explorer. Para obtener las oordenadas en Internet Explorer, obtendremos las coordenadas del navegador y le sumaremos el desplazamiento de la página. El código es el siguiente:

```
coordX = evento.clientX + document.documentElement.scrollLeft;
coordY = evento.clientY + document.documentElement.scrollTop;
```

Los demás navegadores también admiten esta forma de obtener las coordenadas de la página.

Botones del ratón

Otro tipo de información que podemos obtener es cuál es el botón que se ha pulsado en el ratón. Para ello utilizaremos la propiedad `.button` del objeto "event".

```
boton = evento.button;
```

La propiedad sólo funciona con los eventos `onmousedown` y `onmouseup`. Los demás eventos ignoran esta propiedad.

El resultado es un número que indica el botón pulsado. Sin embargo da distinto resultado en Internet Explorer que en el resto de navegadores:

- En Internet Explorer : 1 = botón izquierdo; 2 = botón derecho; 3 = botón izquierdo y derecho a la vez; 4 = botón central.
- En resto de navegadores :0 = botón izquierdo; 1 = botón central; 2 = botón izquierdo.

5.9.3. Información sobre el teclado

El objeto "event" puede darnos información sobre los eventos que se producen desde el teclado. Para ello utiliza las propiedades `.keyCode` y `.charCode`.

Estas propiedades están disponibles para los eventos de teclado, los cuales se producen cuando el usuario pulsa una tecla. Al pulsar una tecla se producen los siguientes eventos y en este orden:

- **onkeydown:** Corresponde al hecho de pulsar una tecla y no soltarla.
- **onkeypress:** Corresponde a la propia pulsación de la tecla.
- **onkeyup:** Corresponde al hecho de soltar una tecla que estaba pulsada.

Las propiedades `.keyCode` y `.charCode` son admitidas por los eventos anteriores, pero su resultado es distinto dependiendo del evento al que se le aplique. Además en Internet Explorer hay algunos resultados que pueden ser diferentes.

Los datos que podemos obtener con estas propiedades son:

- **El código interno:** Es un código que identifica a la tecla en sí, cada tecla tiene un número independientemente de si con ella se pueden escribir uno, dos o tres caracteres.
- **Código del carácter:** Cada carácter asociado a una tecla tiene un código, una tecla puede tener uno dos o tres, códigos asociados, dependiendo de si se pulsa normalmente, en mayúsculas o con la tecla AlrGr.

Para saber cuál es el carácter pulsado se utiliza la función String.fromCharCode(), que convierte el código de carácter a una cadena de texto que contiene el caracter pulsado.

```
caracter = String.fromCharCode(cod_Caracter)
```

La información que muestran las propiedades anteriores puede resumirse de la siguiente manera:

Propiedades de "event" para eventos de teclado

Eventos	Navegador	Propiedad	Valor obtenido
onkeydown onkeyup	Todos	`keyCode`	Código interno
		`charCode`	No definido
onkeypress	Internet Explorer	`keyCode`	Código de carácter
		`charCode`	No definido
	Todos excepto Internet Explorer	`keyCode`	Teclas normales: no definido Teclas especiales: código interno
		`charCode`	Teclas normales: código carácter Teclas especiales: 0

Son teclas especiales aquellas que no sirven para escribir un carácter sino para cambiar algún aspecto de la escritura (mayúsculas, Alt, retroceso, tabulador, etc.).

6. Objetos javascript

6.1. Introducción

Todo elemento de javascript que puede guardarse en una variable es considerado un objeto en javascript.

Hay dos tipos de objetos que podemos utilizar con javascript: los elementos el DOM y los propios objetos de javascript.

Ya hemos visto los objetos del DOM en un tema anterior. Veremos ahora cuáles son los objetos de javascript.

Javascript organiza sus objetos por clases, de manera que cada clase de objetos tiene sus métodos y sus propiedades. Todo objeto en javascript pertenece a una clase, y por lo tanto se le pueden aplicar los métodos y propiedades de su clase.

En javascript tenemos las siguientes clases de objetos:

Clases de objetos en javascript

Clase	Explicación
String	Cadenas de texto: Cualquier contenido escrito entre comillas es considerado una cadena de texto.
Number	Números: tanto enteros como con decimales, en notación científica, o en base 2, 8, o 16;
Date	Permite manejar fechas, guardarlas en variable y trabajar con ellas.
Array	Los arrays son también una clase de objetos y tienen sus métodos y sus propiedades.
Boolean	Esta clase incluye los valores booleanos `true` y `false`
Function	Las funciones también son una clase de objetos
RegExp	Expresiones regulares: Una clase de objetos para comprobar si un cadena de texto se ajusta a un determinado patrón.
Object	Todos los objetos anteriores dependen jerárquicamente del objeto `Object`, punto de origen de los demás objetos en javascript.

A estos añadiremos el objeto Math, que no es una clase de objetos, pero depende también del objeto Object; y que sirve para hacer algunas operaciones matemáticas complejas (raíces, número PI, potenciación, logaritmos, etc).

Al crear una nueva variable javascript la asigna siempre a una clase de objetos, según el contenido de la variable ésta pertenecerá a una clase u otra de objetos. Si cambiamos el valor de una variable, se cambia también la clase de objetos a la que pertenece si el nuevo valor se ajusta más a otra clase.

También hay una forma general de crear cualquier objeto mediante la palabra reservada `new` seguido del nombre de la clase de objeto al cual le añadimos un paréntesis. dentro del paréntesis pondremos como argumento el valor del objeto :

```
var num = new Number(5);
```

6.2. La clase Number()

Al asignar a una variable un número como valor, ésta pasa a ser automáticamente un objeto de la clase Number().

Pertenecen a esta clase todas las variables que contienen números y además pueden contener algunos valores especiales relacionados con los números (`Infinity, -Infinity, NaN`);

6.2.1. Métodos de la clase Number()

Método Number()

El método intentará convertir el elemento que se le pase como argumento en un número. Si no lo consigue devuelve el valor `NaN`. Valor especial que significa "No es un número" ("Not a Number"). Pueden darse los siguientes casos dependiendo del argumento que pasemos:

Argumento:	Ejemplo	Resultado
Un número	`num = Number(23);`	El resultado es el propio número
Cadena que representa a un número	`num = Number("23");`	Convierte el texto en el número que representa
Cadena con números y letras	`num = Number("23px");`	El resultado es `NaN` (Not a Number).
Cadena con sólo texto	`num = Number("un texto");`	El resultado es `NaN` (Not a Number).
Valor booleano `true`	`num = Number(true);`	El resultado es el número 1
Valor booleano `false`	`num = Number(false);`	El resultado es el número 0

Método isNaN()

Este método comprueba si el elemento que pasamos como argumento es un número. Devuelve siempre un valor booleano: `true` cuando NO es un número, y `false` cuando SÍ es un número.

```
valor = isNaN("texto"); // valor == true;
valor = isNaN(23); // valor == false;
```

Método parseInt()

Convierte un número que está escrito en una base que no es base 10 a número en base 10. El primer argumento recoge el número (en base 2, 8, 16 o otra base); el segundo indica la base en la que está el número.

```
num = parseInt("7b",16)
```

Podemos pasar también un número escrito en sistema octal o hexadecimal para convertirlo a base 10. En este caso no se necesita el segundo argumento:

```
num = parseInt(0x7b)
```

Método toFixed()

Este método redondea el valor del número al que se le aplica en el número de decimales que se especifica en el argumento: si el número es negativo el redondeo se hace a nivel de decenas (-1), centenas(-2), etc. Si no se especifica el argumento se redondea al número entero más próximo.

```
numero = 1234.56789
num1 = numero.toFixed(2) // num1 = 1234.57
num2 = numero.toFixed(4) // num2 = 1234.5679
num3 = numero.toFixed() // num3 = 1235
num4 = numero.toFixed(-2) // num3 = 1200
```

6.2.2. Propiedades del objeto Number

Las propiedades del objeto `Number` son sólo de lectura y devuelven una serie de valores que pueden ser numéricos o valores especiales. Son las siguientes:

Propiedades de Number

Propiedad	Ejemplo	Valor
NaN	`num = Number.NaN`	el valor devuelto es `NaN`
MAX_VALUE	`num = Number.MAX_VALUE`	Devuelve el valor máximo que puede soportar el ordenador. En windows es 1.7976931348623157e+308
MIN_VALUE	`num = Number.MIN_VALUE`	Devuelve el valor mínimo que puede soportar el ordenador. En windows es 5e-324
POSITIVE_INFINITY	`num = Number.POSITIVE_INFINITY`	el valor devuelto es `Infinity`
NEGATIVE_INFINITY	`num = Number.NEGATIVE_INFINITY`	el valor devuelto es `-Infinity`

6.3. El objeto Math

El objeto Math nos permite realizar una serie de operaciones matemáticas más complicadas que las simples operaciones normales hechas con operadores. No es una clase de objetos, y utiliza los objetos de la clase `Number` en sus propiedades y métodos.

6.3.1. Propiedades de Math

Las propiedades del objeto Math son una serie de números o valores bastante utilizados en matemáticas. Estas propiedades son de sólo lectura. Son las siguientes:

- **`Math.E`**: Número e o constante de Euler, base de los logaritmos neperianos.
- **`Math.LN2`**: Logaritmo neperiano de 2.
- **`Math.LN10`**: Logaritmo neperiano de 10.

- **`Math.LOG2E`**: Logaritmo en base 2 de e.
- **`Math.LOG10E`**: Logaritmo en base 10 de e.
- **`Math.PI`**: Número pi, que relaciona la circunferencia con su diámetro.
- **`Math.SQRT1_2`**: Raíz cuadrada de un medio.
- **`Math.SQRT2`**: Raíz cuadrada de 2.

Para emplear estas propiedades basta con asignarlas a una variable, o utilizarlas directamente en las operaciones

6.3.2. Métodos de Math

Los métodos del objeto `Math` permiten realizar una serie de operciones con los números que no podemos hacer con los operadores. Podemos clasificarlos según el tipo de operación que hagan.

Métodos trigonométricos

Las funciones trigonométricas son aquellas que muestran la relación entre un ángulo y los lados de un triángulo rectángulo. Tres son las funciones principales, seno, coseno y tangente.

El valor de los ángulos debemos expresarlos en radianes, por lo que si los tenemos en grados debemos pasarlos a radianes. La siguiente expresión nos pasa los grados a radianes.

```
radianes = (Math.PI/180)*grados
```

Métodos trigonométricos

Nombre	Ejemplo	Explicación
sin	`seno = Math.sin(ang);`	Devuelve el valor del **seno** del ángulo pasado en el argumento.
cos	`coseno = Math.cos(ang);`	Devuelve el valor del **coseno** del ángulo pasado en el argumento.
tan	`tangente = Math.tan(ang);`	Devuelve el valor de la **tangente** del ángulo pasado en el argumento.
asin	`arcsen = Math.asin(num);`	**Arcoseno**: Operación contraria al seno. Pasamos un número como argumento y devuelve el ángulo cuyo seno corresponde a ese número.
acos	`arccos = Math.acos(num);`	**Arcocoseno**: Operación contraria al coseno. Pasamos un número como argumento y devuelve el ángulo cuyo coseno corresponde a ese número.
atan	`arctan = Math.atan(num);`	**Arcotangente**: Operación contraria a la tangente. Pasamos un número como argumento y devuelve el ángulo cuya tangente corresponde a ese número.
atan2	`ang = Math.atan2(coordX,coordY);`	**Arcotangente(2)**: Calcula el ángulo que forma un punto del plano (recta entre el punto y el origen de coordenadas), con la recta X (horizontal) de su origen de coordenadas. Como argumento se pasan las coordenadas del punto.

Métodos de cambio de número

Lo que hacen estos métodos es transformar un número en otro que sea más asequible a la utilidad que le queremos dar. En concreto tenemos las siguientes transformaciones: pasarlo a valor absoluto, redondearlo, y crear un número aleatorio. Los métodos son los siguientes:

Métodos de cambio de número

Nombre	Ejemplo	Explicación
abs	`valor = Math.abs(num);`	Devuelve el **valor absoluto** del número pasado en el argumento (siempre en positivo).
ceil	`valor = Math.ceil(num);`	**Redondeo por exceso**: redondea el número pasado en el argumento al entero inmediatamente superior.
floor	`valor = Math.floor(num);`	**Redondeo por defecto**: redondea el número pasado en el argumento al entero inmediatamente inferior.
round	`valor = Math.round(num);`	**Redondeo al más próximo**: redondea el número pasado en el argumento al entero más próximo.
random	`valor = Math.random();`	Crea un **número aleatorio** decimal entre 0 y 1; A este método no se le pasa ningún argumento.

Si queremos que el número aleatorio creado por `random` esté entre otros valores deberemos transformarlo. Por ejemplo si lo que queremos un número aleatorio entero entre 0 y 100 haremos:

```
azar = Math.random()*100; azar = Math.round(azar);
```

Métodos de cálculo

Calculan ciertas operaciones complejas que no pueden realizarse con los operadores normales.

Métodos de cálculo

Nombre	Ejemplo	Explicación
exp	`valor = Math.exp(num);`	Numero **e elevado a** `num`. Multiplica el número e o constante de Euler por sí mismo las veces indicadas en el argumento: e^{num} .
log	`valor = Math.log(num);`	**Logaritmo neperiano** de `num`. Devuelve el logaritmo neperiano del número pasado como argumento.
max	`valor = Math.max(num1,num2);`	**Mayor de dos números**. Se le pasan dos números como argumentos, y el resultado es el mayor de ellos
min	`valor = Math.min(num1,num2);`	**Menor de dos números**. Se le pasan dos números como argumentos, y el resultado es el menor de ellos
pow	`valor = Math.pow(base,exp);`	**Potenciación**: se le pasan como argumento dos números, `base`, `exp`; eleva el primero de ellos al segundo, es decir el primero se multiplica por sí mismo tantas veces como indica el segundo.
sqrt	`valor = Math.sqrt(num);`	**Raíz cuadrada**: Obtiene la raíz cuadrada del número `num` pasado como argumento.

6.4. La clase Date

La clase de objetos `Date` permite guardar y manejar fechas en javascript.

Para crear objetos de la clase `Date` utilizaremos la forma habitual de crear nuevos objetos.

```
fecha = new Date()
```

Si no pasamos ningún argumento en el paréntesis obtendremos la fecha actual (fecha del ordenador del usuario).

Podemos pasar una fecha distinta de la actual indicándolo con los siguientes argumentos:

```
fechaAnterior = new Date(anno,mes,dia,hora,minuto,segundo)
```

Los argumentos que pasemos deben ser todos números e ir en este orden, y además debemos tener en cuenta lo siguiente:

- El año (`anno`) debe escribirse con las cuatro cifras (ej.: 2012, y no 12);
- Los meses empiezan a contar desde 0 de manera que enero = 0, febrero = 1 ... diciembre = 11.
- No debemos poner ceros delante de ningún número (por ejemplo 7, y no 07) tanto en meses, como en horas, minutos o segundos.
- Las horas van desde 0 a 23, (no existe la hora 24). Los minutos y los segundos van desde 0 a 59.

Podemos prescindir de los tres últimos argumentos, y poner únicamente el año, el mes y el día:

```
fechaAnterior = new Date(anno,mes,dia)
```

En este caso la hora, minuto y segundo serán igual a 0.

Al acceder directamente a un objeto de la clase `Date` obtenemos la fecha en un formato que no es muy habitual. Será parecido a lo siguiente:

```
Mon Mar 01 2010 12:23:15 GMT+0100
```

Esto se interpreta de la siguiente manera:

- **`Mon`**: día de la semana en inglés (lunes).
- **`Mar`**: Mes del año en inglés (Marzo)
- **`01:`** día del mes (de 1 a 31)
- **`2010:`** año
- **`12:23:15:`** Hora, minuto y segundo respectivamente.
- **`GMT+0100:`** Diferencia entre la hora local y la hora internacional, o la del meridiano de Greenwich."

Para poder ver las fechas en un formato más normal, usaremos los métodos de lectura de la clase `Date`. También hay métodos de escritura que nos periten modificar las fechas.

Veamos primero los métodos de lectura:

6.4.1. Métodos de lectura.

En la siguiente tabla partimos del siguiente objeto:

```
fecha = new Date()
```

La variable `fecha` contiene la fecha actual. A partir de ahí con los métodos de lectura obtenemos diferentes datos de la fecha.

Métodos de lectura de Date

Nombre	Ejemplo	Explicación
getDate()	`dia = fecha.getDate();`	Devuelve el **día del mes**.
getDay()	`diaSem = fecha.getDay();`	Devuelve el **día de la semana** en formato numérico. La semana empieza en domingo y le corresponde el 0, y se acaba el sábado con el 6.
getMonth()	`mes = fecha.getMonth();`	Devuelve el **mes** en formato numérico. Empieza a contar en el mes de enero con el 0, y acaba en diciembre con el 11.
getFullYear()	`anno = fecha.getFullYear();`	Devuelve el **año completo** escrito con sus cuatro cifras.
getYear()	`annoCorto = fecha.getYear()`	Devuelve el **año en formato corto**: al año completo se le resta 1900; en fechas anteriores al año 2000 se muestran los dos últimos dígitos; a partir del 2000 se muestran los dos últimos dígitos con un 1 delante. En Internet Explorer devuelve el año completo igual que el anterior.
getHours()	`hora = getHours();`	Devuelve la **hora** en formato de 0 a 23.
getMinutes()	`minuto = getMinutes();`	Devuelve el **minuto** en formato de 0 a 59.
getSeconds()	`segundo = getSeconds();`	Devuelve el **segundo** en formato de 0 a 59.
getMilliseconds()	`miliseg = getMilliseconds();`	Devuelve el **milisegundo** en formato de 0 a 999.
getTime()	`tiempo = getTime();`	Devuelve la cantidad de segundos transcurridos desde el 1 de enero de 1970. Es lo que se conoce como **tiempo Unix**. Se emplea habitualmente para cálculos con fechas
getTimezoneOffset()	`local = getTimezoneOffset();`	Devuelve la **diferencia horaria** expresada en minutos,entre la zona en la que estamos y la hora oficial internacional o del meridiano de Greenwich

Los métodos de lectura empiezan todos por la palabra `get`, y en el paréntesis no pondremos ningún argumento.

Obtenemos mediante estos métodos las diferentes partes de una fecha. Tal vez los más problemáticos sean los meses y los días de la semana. Podemos usar para mostrarlos mejor un array donde hayamos guardado los nombres de los meses, y otro array con los días de la semana. Para mostrarlos hacemos referencia a los elementos de estos arrays con los números obtenidos en los métodos.

Otro concepto nuevo es el del "tiempo Unix". Esta es la forma que tienen los ordenadores de manejar el tiempo. Toda fecha se puede convertir en un único número de "tiempo Unix". El tiempo Unix se empieza a contar en milisegundos a partir del 1 de enero de 1970 a 0 horas, 0 min. 0 seg. Toda fecha anterior será un número negativo. El tiempo Unix nos permite hacer operaciones matemáticas con fechas.

6.4.2. Métodos de escritura.

Los métodos de escritura permiten modificar parte de una fecha o la fecha completa, reescribiéndola entera o sólo en parte.

Los métodos de lectura empiezan todos por `get`. Si cambiamos `get` por `set`, tenemos los métodos de escritura. Estos son los siguientes.

Métodos de escritura de Date

Nombre	Ejemplo	Explicación
setDate()	`fecha.setDate(dia);`	Cambia el **día** del mes al recibido en el argumento.
setMonth()	`fecha.setMonth(mes);`	Cambia el **mes** al indicado en el argumento. Este debe estar escrito en formato numérico (0= Enero ... 11=Diciembre).
setFullYear()	`fecha.setFullYear(anno);`	Cambia el **año** al indicado en el argumento. Este debe escribirse en **formato completo** (con sus 4 cifras).
setYear()	`fecha.getYear(annoCorto)`	Cambia el **año** al indicado en el argumento: Este debe escribirse en **formato corto**, tal como se indicó para `getYear()` .
setHours()	`fecha.setHours(hora);`	Cambia la **hora** a la indicada en el argumento. Ésta debe escribirse en formato de 0 a 23.
setMinutes()	`fecha.setMinutes(minuto);`	Cambia el **minuto** al indicado en el argumento. Éste debe escribirse en formato de 0 a 59.
setSeconds()	`fecha.setSeconds(segundo);`	Cambia el **segundo** al indicado en el argumento. Éste debe escribirse en formato de 0 a 59.
setMilliseconds()	`fecha.setMilliseconds(miliseg);`	Cambia el **milisegundo** al indicdo en el argumento. Éste debe escribirse en formato de 0 a 999.
setTime()	`fecha.setTime(tiempo);`	Cambia el **tiempo Unix** de la fecha por el indicado en el argumento. De esta manera podemos cambiar la fecha entera por la que representa el número indicado.

Todos los métodos de lectura tienen su correspondiente método de escritura, excepto los métodos `getDay()` (dia de la semana) y `getTimezoneOffset` (zona horaria).

6.4.3. Operaciones con fechas

Para hacer operaciones con fechas, podemos utilizar las fechas en tiempo Unix. Hay que tener en cuenta que el tiempo Unix está en milisegundos, por lo que un día es igual a

```
dia = 24*60*60*1000
```

También podemos sumar o restar días a una fecha añadiéndolas al método correspondiente. El siguiente ejemplo indica cómo buscar qué fecha será dentro de 120 días:

```
fecha = new Date(); //fecha actual.
diaActual = fecha.getDate(); //dia actual del mes.
diaBuscado = diaActual + 120; //le sumamos 120 días.
fecha.setDate(diaBuscado); //pasamos el resultado a formato de fecha.
```

6.5. La clase String

La clase String incluye las cadenas de texto. Cualquier variable que tenga como valor una cadena de texto es un objeto de la clase `String`.

Crear un objeto de la clase String es tan simple como asignarle un valor de cadena de texto a una variable; aunque también podemos utilizar el método general par crear objetos:

```
var texto = new String("mi texto");
```

6.5.1. Propiedades

La única propiedad de `String` es la propiedad `.length` que devuelve el número de caracteres que contiene la cadena de texto.

```
caracteres = texto.length
```

6.5.2. Métodos habituales

La clase `String` es la que más métodos tiene. Éstos podemos dividirlos en métodos habituales y en métodos de estilo. Estos últimos cambian el estilo del texto, y son poco usados, ya que es preferible cambiar el estilo cambiando las propiedades CSS.

Métodos habituales de String

Nombre	Ejemplo y explicación
toString()	`texto = variable.toString();` Convierte cualquier elemento en una cadena de texto, por lo que en realidad no se aplica a cadenas de texto, sino a cualquier variable.
concat()	`nuevoTexto = texto1.concat(texto2);` Junta dos cadenas. A la cadena `texto1` se le suma la cadena `texto2` pasada como argumento. El resultado es una nueva cadena; igual que cuando las sumamos.
toUpperCase()	`textoMay = texto.toUpperCase();` Crea la misma cadena de texto, pero en la que todos los caracteres se han escrito en letras mayúsculas
toLowerCase()	`textoMin = texto.toLowerCase();` Crea la misma cadena de texto, pero en la que todos los caracteres se han escrito en letras minúsculas.
indexOf()	`posicion = texto.indexOf(caracter);` Pasamos como argumento un carácter. El método lo busca en la cadena, y devuelve un número que indica la posición del carácter encontrado. Si hay más de uno da la posición del primero. Se empieza a contar desde 0, (primer carácter = 0). Si no se encuentra devuelve -1. Podemos pasar como argumento una cadena con más de un carácter . Se busca la cadena y si está está dentro del texto nos dirá en qué posición empieza
lastIndexOf()	`posicion = texto.lastIndexOf(caracter);` Igual que el anterior pero empieza a buscar desde el final de la cadena. El resultado es la posición empezando a contar por el principio a partir del 0.
substring()	`porcionTexto = texto.substring(num1,num2);` Crea una subcadena que va desde el carácter que está en el `num1` al caracter inmediatamente anterior al que está en el `num2`. El segundo número es opcional, y si no se indica la cadena devuelta va hasta el final. La cuenta de caracteres empieza en el 0.
substr()	`porcionTexto = texto.substr(num1,num2);` Crea una subcadena que empieza en el carácter que está en el `num1` y de una longitud indicada por num2. El segundo número es opcional, y si no se pone, la cadena va hasta el final. La cuenta de caracteres empieza en el 0.
split()	`nuevoArray = texto.split(separador);` Convierte la cadena de texto en un array. Como argumento se escribe el caracter que separa los elementos del array, por ejemplo un espacio en blanco (" ") los separa en palabras. Si no se pasa ningún argumento el caracter por defecto es la coma.
replace()	`cambiado = texto.replace(buscarTexto,nuevoTexto);` Remplaza una porción de texto por otra. para ello el primer argumento `buscarTexto` es el trozo de texto que hay que reemplazar. El segundo argumento `textoNuevo` es el texto por el cual va a ser reemplazado el primero.

Nombre	Ejemplo y explicación
charAt()	`letra = texto.charAt(num);` Devuelve el carácter que se encuentra en la posición indicada por el número que se le pasa como argumento. Las posiciones se empiezan a contar desde el 0.
charCodeAt()	`codigo = texto.charCodeAt(num);` Devuelve el código Unicode del carácter que se encuentra en la posición indicada por el número que se le pasa como argumento. Las posiciones se empiezan a contar desde el 0.
link()	`enlace = texto.link("http://google.com")` Convierte el texto en un enlace. En el argumento se le pasa la URL del enlace.

6.5.3. Métodos de estilo de String

Estos métodos modifican el estilo del texto. Suelen ser poco empleados ya que normalmente se prefiere cambiar el estilo cambiando las propiedades CSS.

Métodos de estilo de String

Nombre	Ejemplo y explicación
big()	`masGrande = texto.big();` Aumenta en un punto el tamaño del texto
small()	`masPequeno = texto.small();` Disminuye en un punto el tamaño del texto
bold()	`negrita = texto.bold();` Pone el texto en negrita.
italics()	`cursiva = texto.italics();` Pone el texto en cursiva.
fixed()	`monotipo = texto.fixed();` Pone el texto en estilo monotype o espaciado fijo
strike()	`tachado = texto.strike();` Tacha el texto con una línea horizontal en medio.
blink()	`parpadeo = texto.blink();` Pone el texto parpadeante.
sub()	`subindice = texto.sub();` El texto se muestra como un subíndice.
sup()	`superindice = texto.sub();` El texto se muestra como un superíndice.
fontcolor()	`colorTexto = texto.fontcolor("color") :` Cambia el color del texto al indicado en el argumento. El color puede indicarse en su nombre en inglés para ciertos colores, o en sistema hexadecimal.
fontsize()	`tamano = texto.fontsize(num);` Cambia el tamaño del texto al indicado en el argumento

6.6. La clase Array

Los arrays o listas de elementos son considerados objetos en javascript y forman una clase.

Ya se ha visto anteriormente la forma de crear y definir un array, por lo que no insistiremos en ello.

La clase array tiene también métodos y propiedades propios, que veremos a continuación.

6.6.1. Propiedades

La única propiedad de `Array` es la propiedad `.length` que devuelve el número de elementos que contiene el array.

```
elementos = miArray.length
```

6.6.2. Métodos

Estos son los métodos que pueden aplicarse a la clase Array, y por tanto a cualquier array.

Métodos de Array

Nombre	Ejemplo y explicación
join()	`texto = miArray.join("separador");` Convierte un array en una cadena de texto. Como argumento pondremos el carácter o texto que separa los elementos del array. éste es opcional y de no ponerse será la coma.
reverse()	`inverso = miArray.reverse();` Invierte el orden de los elementos del array.
slice()	`porcion = miArray.slice(num1,num2);` Devuelve un nuevo array que es una porción del array original. El `num1` indica el principio, El `num2` indica el final y este elemento no está incluido. Los elementos se empiezan a contar desde 0. Si se omite el segundo argumento el nuevo array llega hasta el final.
pop()	`ultimo = miArray.pop();` Elimina el último elemento del array original, y lo devuelve en la variable que se le asigna.
push()	`miArray.push("elemento");` Añade un elemento al array original, el cual lo pasamos como argumento. El elemento se coloca después del último, modificando el array en un elemento más.
shift()	`primero = miArray.shift();` Elimina el primer elemento del array original, y lo devuelve en la variable que se le asigna.
unshift()	`miArray.unshift("elemento");` Añade un elemento al array original, el cual se pasa como argumento. El elemento se coloca antes del primero, modificando el array en un elemento más.
concat()	`array3 = array1.concat(array2);` Crea un nuevo array consistente en la suma del `array1`, con el `array2`, (pasado como argumento). El `array2` coloca sus elementos después de los del array1<*/code>.
sort()	`ordenado = miArray.sort();` Crea un array ordenado alfabéticamente. El problema está en que en español no tiene en cuenta la eñe ni las vocales acentuadas, las cuales las pone al final.

6.7. La clase Boolean

La clase de objetos Boolean incluye las variables booleanas o lógicas que sólo pueden tomar los valores true y false.

Para construir un objeto de clase Boolean basta con asignar un valor booleano a una variable, pero también podemos crearlo por el método general:

```
booleano = new Boolean();
```

Según el valor que le pasemos en el argumento nos devloverá `true` o `false`. Si pasamos directamente los valores `true` o `false`, nos devolverá esos valores. Si no pasamos ningún valor o pasamos como valor el número 0 o una cadena de texto vacía, devolverá false. En los demás casos devolverá true.

La clase Boolean no tiene propiedades y métodos propios, sino los heredados del objeto Object.

6.8. La clase Function

Las funciones son una clase de objetos en Javascript, la clase Function. Aunque la forma más habitual de declararlas es: `function miFuncion() { ...}`; también lo podemos hacer mediante la forma general de construir objetos:

```
miFunción = new Function() { ... }
```

La clase Function no tiene propiedades y métodos propios, sino los heredados del objeto Object.

6.9. La clase RegExp

Las expresiones regulares son una clase de objetos en javascript. Más adelante dedicamos a ellas un tema en este resumen.

6.10. El objeto Object

El objeto Object es el que está en un nivel superior en la jerarquía, y del que derivan todos los demás objetos de javascript.

Permite por lo tanto crear nuevas clases de objetos, nuevos métodos y propiedades para los ya existentes. Veremos más detenidamente este objeto y cómo crear nuevos objetos en un tema posterior.

Sus métodos y propiedades son heredados por el resto de los objetos javascript.

Los métodos son: toString() (ya visto con las cadenas de texto) que convierte cualquier objeto en una cadena; y valueOf() que dependiendo del objeto devuelve un valor u otro, aunque casi siempre es el propio objeto.

Propiedades: Sus propiedades son constructor y prototype, las cuales las veremos en un tema posterior.

7. Nuevos objetos

7.1. La función constructor

Podemos crear nuevas clases de objetos mediante una función constructor. Ésta en principio no se diferencia del resto de funciones. Como argumentos podemos pasar algunos elementos que nos servirán para definir sus propiedades.

Los objetos de las nuevas clases que creamos están en sí mismos vacíos de contenido. Son las propiedades y los métodos que les pongamos los que les dan contenido. No se trabaja con el objeto en sí, sino con sus métodos y propiedades.

Creamos una función, y dentro de ella se definen las propiedades y métodos de la misma:

```
function operar (a,b) { /* ...*/ }
```

Seguiremos un ejemplo en el que creamos un objeto que pueda hacer distintas operaciones entre dos números.

7.2. Definir propiedades

Para definir nuevas propiedades dentro de la función constructor utilizamos la palabra reervada `this`. seguida del operador punto y el nombre del método, y le asignamos el valor del argumento que le pasamos:

```
function Operar (a,b) {
   this.n1 = a;
   this.n2 = b;
   }
```

Hemos creado aquí una clase de objetos llamada Operar, la cual tiene dos propiedades, que son `n1` y `n2`

7.3. Instanciar el objeto

Instanciar un objeto es crear un nuevo objeto de una clase. Para la clase creada anteriormente escribiremos:

```
operar1 = new Operar(3,5)
```

Hemos creado el objeto `operar1`. Como argumentos pasamos los valores que tendrán las propiedades del objeto.

Ahora para ver el valor de las propiedades podemos hacer:

```
alert(operar1.n1+" , "+operar1.n2);
```

Esto nos sacará en una ventana de alerta las dos propiedades del objeto.

Las propiedades son de lectura y de escritura, por lo que podemos cambiar el valor de alguna de las propiedades del objeto instanciado:

```
operar1.n1=7;
```

El valor de la propiedad `n1` para el objeto indicado habrá cambiado.

7.4. Definir métodos

Los métodos son en realidad funciones en las que podemos utilizar los valores que tienen las propiedades, refiriéndonos a ellas. Siguiendo el ejemplo anterior, haremos un método que sume las dos propiedades que tiene el objeto.

Para ello dentro de la función constructor ponemos una nueva línea:

```
this.suma=sumar
```

Aquí indicamos que el nuevo método se llamara `suma` y que la función que controla este método es la función `sumar`, la cual la crearemos a continuación. Vemos aquí el código de la función constructor seguido del código de la función `sumar`.

```
function Operar (a,b) {
   this.n1 = a;
   this.n2 = b;
   this.suma=sumar;
   }
function sumar() {
   miSuma = this.n1+this.n2;
   return miSuma;
   }
```

En la función de referencia (`sumar`), sumamos las dos propiedades. Utilizamos la palabra `this` para hacer referencia a las mismas. Debemos devolver el resultado mediante un `return` para que el método funcione.

El método puede aplicarse a cualquiera de los objetos de esta clase. Por ejemplo lo aplicamos al objeto instanciado anteriormente y sacamos el resultado en una ventana de alerta

```
alert(operar1.suma());
```

Esto dará como resultado una ventana de alerta que mostrará la suma de los dos números que tiene el objeto como propiedades.

Puede ser que la función empleada para crear el método necesite algún dato adicional que lo pasaremos en el argumento. En este caso el método necesitará también que se le pase en el argumento ese dato para que funcione correctamente.

7.5. La propiedad prototype

Esta es una propiedad de la clase **`Object`** que nos permite crear un nuevo objeto sin contenido y asignarle después los métodos y propiedades.

Después de crear una nueva clase con una función constructor podemos añadirle nuevos métodos y propiedades fuera de la función constructor mediante la propiedad prototype.

Podemos crear también una nueva clase de objetos mediante una función constructor sin propiedades ni métodos, y añadírselos después con la propiedad prototype.

Al ser esta una propiedad de la clase **`Object`**, es heredadada por el resto de objetos de javascript, lo cual nos permite tambien poder crear nuevos métodos y propiedades para los objetos ya definidos en javascript.

7.5.1. Asignar nuevas propiedades con prototype

Una función sin contenido puede convertirse en una función constructor al asignarle propiedades y métodos mediante `prototype`. por ejemplo :

```
function Calcular() {}
```

Asignamos nuevas propiedades a esta función, mediante la propiedad `prototype`

```
function Calcular() {}
Calcular.prototype.n1 = 0;
Calcular.prototype.n2;
```

Este código convierte la función `Calcular` en una nueva clase de objetos, que tiene dos propiedades `n1` y `n2`. La propiedad prototype permite crear nuevas propiedades. Podemos asignar un valor a la propiedad, el cual será su valor por defecto, o podemos no asignarle ningún valor en principio.

Para comprobar lo anterior podemos después instanciar un objeto de la clase, asignar un nuevo valor a las propiedades, y comprobarlas mediante una ventana de alerta, tal como hacemos en este código:

```
calculo1=new Calcular();
calculo1.n1=3;
calculo1.n2=5;
alert(calculo1.n1+" , "+calculo1.n2);
```

7.5.2. Asignar nuevos métodos con prototype

Crear un nuevo método para una clase de objetos se hace de forma similar, sólo que antes debemos crear la función del método, y después incluirla en la clase con `prototype`. Seguimos con el ejemplo anterior para crearle un método.

```
function sumar() {
   miSuma=this.n1+this.n2
        return miSuma
   }
Calcular.prototype.suma=sumar;
```

Creamos primero la función, en la cual podemos hacer referencia a las propiedades que tiene el objeto mediante la palabra `this`. La función debe llevar siempre un `return` que devuelve el resultado buscado.

Después añadimos la función al objeto como un método; para ello después de la propiedad `prototype` escribimos un punto y el nombre del método, y le asignamos como valor el nombre de la función (sin paréntesis).

Siguiendo con el ejemplo anterior, podemos comprobar que el método funciona, aplicándoselo al objeto que ya habíamos instanciado:

```
alert(calculo1.suma());
```

7.6. Prototype en objetos predefinidos: propiedades.

Podemos crear nuevas propiedades en las clases de objetos predefinidos de javascript mediante la propiedad prototype:

```
Array.prototype.nombre="miArray";
```

Podemos después usar esta propiedad en cualquier array que tengamos.

```
Array.prototype.nombre="miArray";
semana = ["L","M","X","J","V","S","D"];
semana.nombre="Semana";
alert(semana.nombre)
```

El resultado es un ventana de alerta con la palabra "Semana", es decir, sacamos en una ventana de alerta la nueva propiedad aplicada a un objeto concreto.

Sin embargo esto no funciona igual con todos los objetos de javascript, ya que para los objetos de las clases Number, String y Boolean, las propiedades que crea el programador son de sólo lectura.

En esta clases, sólo si el objeto ha sido creado mediante la forma general de crear objetos (por ejemplo `texto = new String("hola");`), se puede modificar la propiedad con la escritura.

7.7. Prototype en objetos predefinidos: métodos.

Podemos crear nuevos métodos para las clases predefinidas de javascript mediante la propiedad prototype. Estos funcionan siempre con todas las clases, ya sea con objetos creados con el método general o simplemente por asignación a una variable.

En los objetos creados por el programador, cuando creamos un método hacemos referencia a sus propiedades mediante la forma `this.propiedad`. Sin embargo aquí no queremos acceder a una propiedad, sino al objeto en sí, por lo que la forma más simple de acceder a él es igualar una variable a la palabra `this`.

Veamos un ejemplo en el que creamos un método en el que indicamos que un número representa una cantidad de dinero en la divisa que le indicamos.

```
function divisa(logo="") {
   dinero=this;
   dinero = dinero.toFixed(2) + logo
   return dinero
   }
Number.prototype.moneda=divisa;
```

Hemos creado el método `moneda()`. Éste redondea el número que le pasamos a dos decimales y le añade el símbolo de la moneda si se lo pasamos como argumento. En caso de no pasar ningún argumento no devuelve nada, tal como indicamos en el argumento: (`logo=""`). Podemos aplicar este método a algunas variables con números, obteniendo los resultados que indicamos.

```
n1=5.247; n1=n1.moneda("€"); // resultado: 5.25€
n2=7.293; n2=n2.moneda("$"); // resultado: 7.29$
n3=6.5; n3=n3.moneda(); // resultado: 6.50
```

8. Expresiones Regulares

8.1. Definición

Las expresiones regulares son objetos que sirven para comprobar si un determinado elemento se ajusta a unas determinadas características.

En javascript se utilizan para saber si una cadena de texto se ajusta a un determinado patrón. La expresión regular marca el patrón a seguir, y luego mediante los métodos y

propiedades comprobamos si la cadena se ajusta a ese patrón y realizamos acciones como buscar, remplazar, etc.

Las expresiones regulares utilizan un lenguaje propio.

No son exclusivas de javascript, otros lenguajes de programación como Java o PHP también las usan.

8.2. Crear un objeto de la clase RegExp

Las expresiones regulares tienen una sintaxis propia. Toda expresión regular empieza y acaba por una barra inclinada: `/ (código) /`. Esto las distingue el resto de variables, por lo que podemos crearlas al definir la variable:

```
var exp = /texto/
```

También podemos crearlas mediante el método general para crear objetos.

```
var exp = new RegExp(texto)
```

En este caso no debemos poner el código entre barras inclinadas.

8.3. Comprobar patrones

8.3.1. Comprobar un texto

El caso más simple es comprobar si una determinada cadena está contenida en otra. En la expresión indicamos la cadena a comprobar.

```
exp = /martes/
```

Podemos comprobar ahora si un determinado texto contiene la palabra "martes". Aunque veremos más adelante los métodos, utilizamos de momento el método `test()`.

```
resultado = exp.test(texto)
```

En el ejemplo `exp` es la expresión regular, y `texto` es la cadena de texto. El método devuelve un valor booleano: `true` si hay coincidencia, o `false` si no la hay.

8.3.2. Comprobar grupos de caracteres

Si lo que se quiere comprobar es que la cadena contenga alguno de los cractere indicados en un grupo éstos se pondrán entre corchetes:

```
exp = /[0123456789]/;
```

Aquí se buscará si el texto contiene algún número.

Cuando los elementos de dentro del corchete forman grupos homogéneos, podemos poner la primera y la última y entre medio un guión:

```
exp = /[a-z]/;
```

Aquí se buscará si el texto contiene alguna de las letras del alfabeto.

Para incluir el guión en la expresión anterior podemos ponerlo al final, o también al principio, pero con una barra inclinada inversa que indica que es un carácter de escape.

```
exp = /[a-z-]/;
exp = /[\-a-z]/;
```

Estas dos RegExp buscan si el texto contiene cualquier letra del abecedario o un guión.

Los caracteres de escape tienen la misma función que en los textos, y sirven para incluir caracteres que en el código de las RegExp se usan también para otras cosas.

8.4. Clases de caracteres

Para los casos más habituales hay unos caracteres especiales que indican si la cadena contiene un determinado tipo de caracteres:

- `/\w/` : Contiene caracteres alfanuméricos.
- `/\W/` : Devuelve false si sólo contiene caracteres alfanuméricos.
- `/\d/` : Contiene digitos o cifras
- `/\D/` : Devuelve false si sólo contiene dígitos.
- `/\s/` : Contiene alguno de los caracteres que definen espacios en blanco (espacio, tabulador, retorno de carro ...)
- `/\S/` : Devuelve false si sólo contiene caracteres que definen espacios en blanco.

Tenemos también la expresión `/./` que indica "cualquier caracter" y que se utiliza para saber si una cadena no está vacía.

Los signos `^` y `$` al principio y final de la RegExp, indican que ésta debe coincidir completamente con la cadena de texto (no sólo una parte).

8.5. Cuantificadores

Además del patrón o texto a buscar, hay unos signos que ponemos detrás y que indican las veces que debe aparecer el texto para ajustarse al patrón. Son los cuantificadores:

- `/a+/` : La cadena debe contener el patrón (letra "a") una o más veces.
- `/a*/` : El patrón (letra a) debe aparecer cero o más veces.
- `/a?/` : El patón debe aparecer cero o una vez.
- `/a{3}/` : El patón debe aparecer exactamente tres veces (en lugar del 3 puede ser cualquier otro número).
- `/a{3,8}/` : El patón debe aparecer entre 3 y 8 veces (ambos inclusive).

8.6. Modificadores

Los modificadores o "flags" son unas letras con un significado especial que se ponen detrás de la expresión regular, y matizan la forma de buscar. Estos son los siguientes:

- **g :** `/a/g` : Explora la cadena hasta el final.
- **i :** `/a/i` : No distingue entre mayúsculas y minúsculas.
- **m :** `/a/m` : Permite usar varios ^ y $ en la cadena.
- **s :** `/a/s` : Incluye el salto de línea en el comodín punto .
- **x :** `/a/x` : ignora los espacios en el patrón .

Si escribimos más de un modificador en una expresión regular, debemos ponerlos en el mismo orden que aparecen arriba, es decir, en orden alfabético.

8.7. Métodos de RegExp.

La clase RegExp tiene varios métodos. Además también podemos usar algunos métodos de la clase String con las expresiones regulares.

Métodos de RegExp

Nombre	Ejemplo y explicación
test()	`buscar = exp.test(texto)` Devuelve un valor booleano que indica si la expresión (exp) está contenida o no en la cadena(`texto`).
compile()	`compilado = exp.compile(exp);` Convierte la expresión en un formato interno para que la ejecución sea más rápida. Por ejemplo, esto permite un uso más eficiente de RegExp en bucles.
exec()	`buscar = expresion.exec(texto);` busca la `expresion` en el `texto`, y devuelve el primer texto que concuerda con la expresión buscada. Si no encuentra ninguna coincidencia, entonces devuleve null.

El método `exec` tiene además dos propiedades: `.index`, que nos indica la posición en la que se encuentra la cadena buscada; y `.input`, que devuelve la cadena completa en la que estamos realizando la búsqueda.

8.8. Métodos de String para RegExp.

Los siguientes métodos de la clase String pueden usarse para trabajar con expresiones regulares. En ellos la RegExp se pasa siempre como argumento.

Métodos de String para RegExp

Nombre	Ejemplo y explicación
search()	`buscar = texto.search(expresion)` Busca la `expresión` en el `texto`. si la encuentra devuelve un número indicando la posición del primer carácter, empezando a contar desde 0, si no la encuentra devuelve -1.
split()	`miArray = texto.split(expresion);` Transforma el `texto` en un array, la `expresion` indica cual es el delimitador que separa los elementos del array.
replace()	`cadena = texto.replace(expresion,nuevo_texto)` Devuelve el `texto` original, en la que se ha remplazado las coincidencias encontradas por la `expresion`, por un `nuevo_texto` pasado como segundo argumento.

8.9. Propiedades de RegExp

Las propiedades del objeto RegExp se usan directamente sobre el objeto `RegExp`, por tanto se escribirán de la forma: `RegExp.propiedad`

Son de sólo lectura, sin embargo se actualizan automáticamente cada vez que se emplea un método, ya sea de RegExp o de String; por lo que el valor obtenido corresponderá al último método empleado.

Propiedades de RegExp

Propiedad	Ejemplo y explicación
$1 ... $9	`indice = RegExp.$1` Índices que contienen las partes agrupadas con paréntesis en el patrón de búsqueda.
input	`texto = RegExp.input` Devuelve la última cadena que se ha explorado mediante un método.
lastMatch	`texto = RegExp.lastMatch` Devuelve la última coincidencia encontrada.
multiline	`comprobar = RegExp.multiline` Devuelve una variable booleana que indica si la cadena explorada incluye saltos de línea.
lastParent	`texto = RegExp.lastParent` Devuelve la última coincidencia encontrada con un patrón entre paréntesis.
leftContext	`texto = RegExp.leftContext` En la última cadena explorada devuelve desde el principio hasta la última coincidencia hallada.
rightContext	`texto = RegExp.rightContext` En la última cadena explorada devuelve desde la última coincidencia hasta el final.

8.10. Expresiones Regulares habituales

Los ejemplos vistos hasta ahora tienen expresiones regulares sencillas. Sin embargo la cosa puede complicarse cuando tratamos de hacer expresiones regulares que se ajusten a patrones más complejos, para comprobar que algunas cadenas están escritas correctamente.

En estos casos lo más sencillo es recurrir a una lista con las RegExp más habituales. Veamos algunas de ellas:

- **E-mail** : `/^[\w]+@{1}[\w]+\.[a-z]{2,3}$/`
- **Página web** : `/^http[s]?:\/\/[\w]+([\.]+[\w]+)+$/`
- **Nombre propio de persona** : `/^([A-ZÁÉÍÓÚ]{1}[a-zñáéíóú]+[\s]*)+$/`
- **Núm. teléfono de España** : `/^[9|6]{1}([\d]{2}[-]*){3}[\d]{2}$/`

9. Cookies

9.1. Definición

Las cookies son datos de la página que pueden guardarse en el ordenador del usuario, de manera que éste pueda recuperarlos

Al cerrar la página se perderán los datos que hay en ella, a no ser que los guardemos en cookies. En javascript podemos guardar cualquier dato o variable en una cookie. Al volver a abrir la página las cookies guardadas estarán disponibles para ser usadas.

Las cookies se guardan en el ordenador del usuario, y concretamente en el navegador que se está usando. Para ello el navegador debe tener habilitadas las cookies. Normalmente solemos encontrar el botón para habilitar cookies en la pestaña de "herramientas/opciones", o en "configuración": aunque lo normal es que éstas estén ya habilitadas por defecto.

Las cookies se guardan en el navegador, por lo que para recuperarlas debemos usar no sólo el mismo ordenador, sino también el mismo navegador en que las hemos creado. Sin embargo una cookie creada en una página puede verse en otra distinta.

Hay ciertas limitaciones en el uso de cookies. Un usuario no puede almacenar más de 300 cookies, y el tamaño máximo de cada cookie es de 4000 bytes.

9.2. Guardar cookies

Para crear una nueva cookie usaremos la propiedad `document.cookie` de la siguiente manera:

```
document.cookie = "nombre=valor";
```

A cada cookie le asignamos un nombre y un valor. Con el nombre identificamos la cookie entre otras que podamos tener, y en el valor guardamos el dato que nos interesa.

En el ejemplo anterior hemos pasado los datos en forma de cadena de texto, por lo que lo escrito es lo que guardamos. Sin embargo podemos pasar los datos mediante variables si los sacamos de la cadena, por ejemplo:

```
nombre = "color";
valor = "rojo";
document.cookie = nombre+"="+valor;
```

De esta manera podemos guardar cualquier variable en una cookie, y además guardar su nombre en una variable.

Para cambiar el dato almacenado en una cookie basta rescribirla cambiando sólo el dato, pero no el nombre:

```
document.cookie = "color=rojo"; //cookie inicial
document.cookie = "color=verde"; //Cambiamos el valor de la cookie
```

Las cookies tal como las tenemos hasta ahora no pueden ser guardadas, pues desaparecen al cerrar la página. Para poder conservarlas debemos ponerles una **fecha de caducidad**. Esto lo hacemos añadiéndole el siguiente código:

```
document.cookie = "color=rojo;expires=31 Dec 2020 23:59:59 GMT"
```

Después del nombre y el valor, escribimos el punto y coma, la palabra `expires` y el signo igual, a continuación escribimos la fecha de caducidad. La fecha irá siempre en el formato que tiene en el ejemplo, es decir: día del mes, mes abreviado en inglés, año completo, hora minuto y segundo separados por signos de dos puntos, y el literal `GMT`

Todo lo anterior, incluido el nombre y el valor, está incluido dentro de una cadena de texto. Por lo tanto si lo tenemos guardado en variables, debemos sacar las variables de la cadena, para unirlas a ella mediante el signo + . Ejemplo:

```
nombre = "color";
valor = "rojo";
fecha = "31 Dec 2020 23:59:59 GMT";
document.cookie = nombre+"="+valor+";expires="+fecha;
```

La cookie dejará de funcionar cuando le llegue su fecha de caducidad. Entonces será borrada de la lista de cookies.

Por lo tanto para **borrar** una cookie basta con cambiarle la fecha de caducidad a una fecha anterior a la actual.

9.3. Leer cookies

No hay un método concreto para acceder a un valor determinado de una cookie, por lo que seguiremos una serie de pasos.

En primer lugar accedemos a todas las cookies mediante la instrucción:

```
misCookies = doument.cookie;
```

La variable `misCookies` contendrá todas las cookies que tenga guardadas el usuario en el ordenador mediante javascript. Estas se mostrarán en una cadena de texto que tiene el siguiente formato:

```
"nombre1=valor1;nombre2=valor2;/* ... */nombreN=valorN"
```

El siguiente paso sería convertir esta cadena de texto en un array en el que cada elemento sea una cookie, para ello utilizamos el método `split()`.

```
listaCookies = misCookies.split(";")
```

Como separador utilizamos el punto y coma, que es el elemento que separa las cookies dentro de la cadena. De esta manera se forma un array en el que cada elemento es una cookie distinta.

El siguiente paso es buscar la cookie en el array. Nosotros conocemos el nombre e la cookie, y queremos buscar su valor. Crearemos un bucle que busque la cookie dentro del array. Utilizaremos para ello el método `search()`. Por ejemplo buscamos una cookie llamada `"color"`.

```
for (i in listaCookies) {
    busca = listaCookies[i].search("color");
    if (busca > -1) {micookie=lista[i]}
    }
```

Tenemos ya aislada la cookie en una cadena de texto (variable `micookie`), sin embargo en ésta se muestra tanto el nombre como su valor separados por el signo igual (`"nombre=valor"`).

Lo que nos interesa aquí es obtener el valor, para ello utilizamos los métodos `indexOf` y `substring()` para aislar el valor de la cookie.

Con `indexOf` localizamos la posición en la cadena del signo igual, que es el que separa los dos valores:

```
pos = micookie.indexOf("=");
```

Esto nos devuelve el número de posición del signo igual. Para tener una cadena que tenga solamente lo que hay detrás del signo igual, utilizamos el método `substring()`

```
micookie = micookie.substring(pos+1);
```

Con esto obtenemos el valor de la cookie buscada.

Todo este proceso podemos ponerlo en una función en la que pasamos como argumento el nombre de la cookie y nos devuelve su valor. La función será la siguiente:

```
function leerCookie(nombre) {
         var lista = document.cookie.split(";");
         for (i in lista) {
             var busca = lista[i].search(nombre);
             if (busca > -1) {micookie=lista[i]}
             }
         var pos = micookie.indexOf("=");
         var valor = micookie.substring(pos+1);
         return valor;
         }
```

Con esta función podemos leer fácilmente cualquier cookie, no tenemos más que llamarla y pasar como argumento el nombre de la cookie para obtener su valor:

```
dato = leerCookie("color");
```

9.4. Almacenamiento Local en HTML5

El nuevo HTML5 resuelve el problema mediante un nuevo método, el del almacenamiento local. Este crea una serie de métodos que almacenan los de datos en el ordenador del usuario de una manera más eficaz, dejando las cookies un tanto obsoletas. Para ver más sobre esto nos remitimos al manual de HTML5.

Contenidos