
JS
JS elemental

¿Será posible resumir todo lo que necesitamos saber sobre JavaScript en unas pocas páginas?

La respuesta es NO.

Pero eso no significa que no debamos comenzar aunque sea de modo básico, sin pretensiones, tratando de no limitarse a copiar y pegar tal o cual cosa.

Hay preguntas que sólo se pueden responder con generalidades porque son preguntas demasiado específicas: *quiero que tal cosa haga tal otra y se vea de cierto modo*.

Esos detalles son la clave de todo porque son infinitos; habrá tantos como ideas existan. El diseño web es más que nada una artesanía, se hace a mano, no existe lo universal, todo depende del contexto, de lo que uno quiera, todo depende de las excepciones. Si no las hay, es fácil; si las hay, habrá que analizarlas una por una.

Si lo hacemos bien, el resultado será aceptable y nos aliviará trabajo en el futuro.

Índice de contenido

Introducción

JavaScript es un lenguaje que asusta a más de uno y eso no es un problema siempre y cuando nos limitemos a seguir las instrucciones de quien explica cierta cosa y nos resignemos a que eso funcione o no funcione pero sin cuestionar demasiado.

Sin embargo, a veces queremos más y está bien que así sea, pero, salvo que alguien descubra un método de aprendizaje instantáneo, no hay más remedio que hacer las cosas paso a paso y ser conscientes de nuestras limitaciones para ... no meter la pata.

JavaScript tiene sus bemoles y hay códigos sencillos y códigos complejos, tanto unos como otros nos permiten ahorrar trabajo o hacer algo que de otra manera sería imposible. Probablemente, lo más común es generar etiquetas HTML o modificar las existentes.

No es algo tan difícil como parece a primera vista y sólo requiere un poco de paciencia y conocer algunas instrucciones elementales.

¿Qué es un *script*? Sin importar el contenido, es una serie de "textos" con instrucciones. Texto plano, sin formato, textos que podemos ver o editar con cualquier programa incluyendo el *block* de notas.

Las rutinas escritas en JavaScript pueden ser incluidas en cualquier parte de la página. Para esto, se debe insertar el código dentro de la etiqueta **<script>**:

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

O se pueden insertar desde un archivo externo que generalmente tiene una extensión *.js*.

¿Qué los diferencia de cualquier otro archivo de texto?

Nada, excepto la extensión que puede ser cualquier otra o no tener ninguna ya que eso sólo es una convención, basta que sea un archivo de texto plano sin formato y que el contenido sea un código válido y no contenga etiquetas HTML. De esta manera, el mismo archivo, guardado como: *miScript.js, miScript.txt, miScript.xxx* o *miScript* funcionará sin mayores problemas.

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

Como si esto fuera poco, los *scripts* también pueden ser parte de algún atributo, como por ejemplo:

```
<a href="javascript:void(0)">click acá</a>
<p onclick="mifuncion();">click acá</p>
```

Muchas veces veremos que en la etiqueta **<script>** se agega el atributo **language** pero esto debe ser evitado ya que es un atributo depreciado. El atributo **type** es suficiente ya que de ese modo se establece el tipo MIME o sea, se le indica al navegador el tipo de archivo a cargar.

Eventualmente, podemos utilizar una etiqueta **<meta>** para definir el tipo general para toda la página y de tal manera, eliminarlo de las etiquetas individuales:

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

Por lo general, los *scripts* se ubican en el **<head>** de la página, esto provoca que al cargarse la página se produzca una demora ya que el navegador debe acceder a esos archivos, leerlos y eventualmente, ejecutarlos. Mientras eso ocurre, el resto de los procesos de carga se detiene. Sin embargo, no hay una regla fija que nos permita decir que todos los *scripts* deben ubicarse en tal o cual lugar de nuestra página, depende de qué hagan, si no se está seguro de eso, mejor seguir las indicaciones de quienes los han desarrollado o probado.

Al igual que en el lenguaje HTML, pueden agregarse comentarios que son ignorados por el navegador. La forma básica de hacerlo es mediante dos barras inclinadas:

```
// este es un comentario simple
```

y si se trata de textos más largos puede usarse esta sintaxis:

```
/*
      Este es un comentario largo
      que ocuparía varias líneas
*/
```

La sintaxis se basa en declaraciones (*statements*); líneas con instrucciones que terminan con un punto y coma que pueden estar en líneas separadas:

```
dato1 = 5;
dato2 = 6;
suma = dato1 + dato2;
```

o en la misma línea:

```
dato1 = 5; dato2 = 6; suma = dato1 + dato2;
```

Si bien es cierto que el punto y coma final no siempre es obligatorio, es mejor acostumbrarse a agregarlo.

VER REFERENCIAS [Librerías: ¿si? ¿no? ¡Socorro!]

Valores y variables

Una variable es una forma de “guardar” un determinado valor que puede ser un número, una cadena de texto escrita entre comillas simples o dobles, etc que se debe “declarar” utilizando la palabra **var**.

Podríamos decir que una variable es un “cajoncito” donde guardamos algo y al que le damos un nombre para saber cuál es cuál ¿Qué podemos guardar? casi cualquier cosa; así, yo puedo decir que:

```
var minumero = 2; // guardo un valor
var undato = "Fulano de Tal"; // guardo un texto
```

Una vez que la hemos declarado, podemos utilizarla, por ejemplo, asignándola a otra variable:

```
var otrodato =  undato; // ambas variables serán iguales
```

o efectuando alguna clase de operación aritmética:

```
var suma = minumero + 100; // suma será una variable que contendrá el número 102
```

Las llamamos variables porque podemos cambiar su contenido:

```
minumero = minumero + suma + 10; // y ahora contendrá el número 114
```

Lo mismo podríamos hacer con las cadenas de texto:

```
undato = "el nombre es " + undato;
```

y el “cajoncito” ahora contendrá el valor *"el nombre es Fulano de Tal"*.

Una variable puede ser declarada sin dato alguno, de ese modo, “reservamos” ese nombre. Además, podemos declarar varias en una sola línea, separándolas con comas:

```
var dato1, dato2, dato3;
```

Es muy importante tener en cuenta que JavaScript es sensible a minúsculas y mayúsculas así que estas tres variables, serán diferentes:

```
var midato, MIDATO, MiDato;
```

Al igual que el HTML o el CSS, los espacios en blanco no se tienen en cuenta a menos que se utilicen como valores y estén entre comillas. En este ejemplo, ambos son iguales:

```
var midato =        "un texto";
var midato = "un texto";
```

Las reglas para dar nombre a las variables o funciones son relativamente amplias pero tienen limitaciones.

El primer carácter siempre debe ser una letra, un guión bajo (**_**) o el sino dolar (**$**); el resto pueden ser letras, números, guiones o el sino dólar (**$**):

```
minombre = "Fulano";
MiNombre = "Fulano";
miNombre = "Fulano";
_miNombre = "Fulano";
$mi_nombre = "Fulano";
```

Todos esos ejemplos serían aceptados pero definirían variables distintas.

VER REFERENCIAS [Los caracteres “raros”]

Operadores

El operador básico es el signo igual (=) y es fácil de entender su uso:

```
var dato = 5;
```

Tampoco es complicado entender que hay operadores aritméticos que se utilizan como en cualquier calculadora: suma (+), resta (-), multiplicación (*), división(/) y paréntesis para indicar el orden de esas operaciones:

```
var numero = 5;
var dato = (10 + numero) * 10; // será 150
```

Cuando los datos son cadenas de texto, la "suma" se llama concatenación pero el operador es el mismo:

```
var palabra = "cualquiera";
var dato = "un texto " + palabra; // será "un texto cualquiera"
```

Ese tipo de combinaciones donde entremezclamos valores, variables y operadores es muy práctica porque no nos obliga a tener que definir las cosas previamente pero puede provocar confusión y resultados erróneos cuando tratamos de evaluar distintos tipos de datos.

Tal como siempre se ha dicho, es imposible sumar peras y manzanas. Por ejemplo, si una variable es un número, otra es un texto y tratamos de sumarlas, el resultado será imprevisible:

```
var dato1 = 20; // este es un número
var dato2 = "20"; // este es un texto
var suma = dato1 +  dato2; // no dará como resultado el número 40 sino "2020"
```

Si bien esto parece que sería algo que nunca podría ocurrir salvo que sea un error, en realidad, suele ser una situación bastante habitual y, justamente por eso es que JavaScript posee funciones que nos permiten convertir un tipo de dato en otro.

Number() hace que una cadena de texto sea tratada como número:

```
var dato1 = 20; // este es un número
var dato2 = "20"; // este es un texto
var suma = dato1 +  Number(dato2); //el resultado será el número 40
```

A la inversa **String()** convierte un número en una cadena de texto:

```
var suma = String(dato1) +  dato2; //el resultado será el texto "2020"
```

De todos modos, cuando comparamos tipos distintos las cadenas de texto serán convertidas a números (si es una cadena vacía será cero) pero, si esa cadena es imposible de convertir, el resultado será el valor **NaN**.

Normalmente, comparamos dos cadenas de texto para ver si son iguales o distintas pero, podrían usarse otros operadores como mayor que o menor que. En ese caso, se tendrá en cuenta el orden alfabético:

```
"abecedario" < "diccionario"; // será true
"zoologico" < "diccionario"; // será false
```

El operador de módulo se indica con el carácter **%** y devuelve el resto de una división:

```
var resto = dato1 % 3; // sería 2 porque es el resto de 20/3
```

Las sumas y restas poseen una alternativa cuando se trata de incrementar o decrementar algo en una unidad. Por ejemplo:

```
var incremento, decremento;
var dato = 1;
incremento = dato + 1;
decremento = dato - 1;
```

podrían simplificarse de este modo:

```
incremento = dato++; // será 2
decremento = dato--; // será 0
```

Y si sólo quisiéramos operar sobre ese dato sin asignarlo a otra variable:

```
dato++; // dato será 2
dato--; // dato será 0
```

Hay más operadores sofisticados pero no abusemos.

VER REFERENCIAS [Operadores especiales]

Tipos de datos

Hemos visto que en las variables se guardan datos y que hay que tener cuidado de no mezclarlos ya que en este lenguaje no es necesario crearlas estableciendo su tipo; este, puede ser modificado una y otra vez:

```
var dato; // lo declaramos sin contenido (undefined)
dato = 1234; // ahora tiene un número (Number)
dato = dato * 10; // ahora tiene otro número (Number)
dato = "un texto"; // ahora es un texto (String)
```

Los datos de tipo **String** son textos, cadenas de caracteres que deben estar siempre entre comillas simples o dobles indistintamente.

Si queremos que el texto en si mismo tenga comillas podemos hacerlo fácilmente. Si el *string* está entre comillas dobles, la interna será simple y viceversa.

```
var texto1 = "esto es un 'ejemplo' de texto";
var texto2 = 'esto es un "ejemplo" de texto';
```

La propiedad **length** nos permite conocer la longitud de esa cadena de texto:

```
var texto = "esto es un 'ejemplo' de texto";
alert(texto.length); // nos mostrará 29
```

VER REFERENCIAS [Métodos de los strings]

Los datos de tipo **Number** son números, positivos o negativos, enteros o decimales o escritos en notación científica:

```
var num1 = 1234;
var num2 = 3.1416; // el máximo número de decimales es 17
var num3 = -10;
var num4 = 12e4;
var num5 = 12e-4;
var num6 = 0x3F; // número en formato hexadecimal (63)
var num7 = 0x3F90; // número en formato hexadecimal (16272)
```

Cuando el resultado de un cálculo supera el máximo admitido por JavaScript o se intenta realizar una división por cero el resultado será **Infinity** que no es un mensaje de error sino un número.

Si se intenta realizar un cálculo con valores que no son números, se devolverá **NaN** (*Not a Number*).

VER REFERENCIAS [Number y Math]

Los datos de tipo **Boolean** sólo son dos: *true* (verdadero) o *false* (falso):

```
var respuesta = true;
var respuesta = false;
```

Una técnica importante en JavaScript es el uso de datos de tipo *array* (arreglos o matrices).

Un *array* es un conjunto de valores asociados a una variable y la forma sencilla de crearlas es enumerarar esos valores separados por comas y entre corchetes:

```
var ejemplo = ["primero", "segundo", "tercero"];
```

Allí definimos un dato de tipo *array* con tres elementos que en ese caso son textos pero podría ser cualquier otro tipo. A cada uno esos datos se accede mediante su índice es decir, el número de orden en que han sido escritos, comenzando a contar desde cero:

```
ejemplo[0] = "primero";
ejemplo[1] = "segundo";
ejemplo[2] = "tercero";
```

Un dato de tipo **Object** es un caso especial al que ya nos referiremos más adelante pero, básicamente se trata de un conjunto de "propiedades" que se crea utilizando llaves y donde se enumeran esas "propiedades" dándoles un nombre y un valor a cada una de ellas separándolas con comas.

```
var obj = {nombre:"Fulano",id:"1234567",n:10};
```

o para que se vea más claro:

```
var obj = {
        nombre : "Fulano",
        id : "1234567",
        n : 10
};
```

En ese ejemplo, el dato contiene tres propiedades y cada una de ellas consta de dos partes, la primera es el nombre que le damos (puede ser cualquiera), luego dos puntos y por último el valor. Podemos acceder a cada propiedad de dos maneras:

```
var dato = obj.nombre; // será "Fulano"
```

o bien:

```
var dato = obj["nombre"]; // será "Fulano"
```

Y podemos hacer lo mismo con cualquier otra propiedad:

```
var dato = obj.id; será el String "1234567"
var dato = obj.n; // será el número 10
```

Ya vimos que no es necesario que una variable tenga un valor inicial:

```
var dato;
```

En ese caso, se dice que su tipo es **undefined** (indefinido) y nada impediría que asignáramos ese "valor" a una variable aunque eso no haría que dejara de estar indefinida:

```
var dato = undefined;
```

Otro valor que puede confundir es **null** (nulo) que es lo que nos responderá JavaScript cuando queremos usar algo que no existe, algo que ni siquiera es indefinido.

Por cierto, como nada impide que cometamos errores es común que hagamos alguna operación y el resultado que se nos muestre sea **NaN** (*Not a Number*). JavaScript nos dice que estamos operando con uno o más valores que no son números ni pueden ser convertidos a números.

El operador **typeof** nos permite conocer el tipo de dato al que pertenece una variable y de ese modo, chequearlo si es necesario:

```
typeof 10; devuelve Number
typeof "un texto"; devuelve String
typeof false; devuelve boolean
typeof {nombre:"Fulano", id:"1234567"}; devuelve object
typeof mifuncion; devuelve function
```

si la variable no se ha declarado o no existe devuelve **undefined**.
si la variable es **NaN** devuelve **number**
si la variable es **null** devuelve **object**
si la variable es una fecha devuelve **object**
si la variable es un *array* devuelve **object**

VER REFERENCIAS [Fechas]

Condiciones

En JavaScript, cuando queremos comparar si dos valores son idénticos, usamos dos caracteres igual (==) pero si queremos ver si son distintos, le anteponemos un signo de admiración (!=); en ambos casos, el resultado será *true* (si lo son) o *false* (si no lo son).

Como en álgebra, también podemos comparar si algo es menor o mayor que. Suponiendo que tenemos que una variable *dato* tiene un valor de *1*:

```
dato == 1; // devuelve true
dato != 1; // devuelve false
dato > 8; // devuelve false
dato < 8; // devuelve true
dato >= 8; // devuelve false
dato <= 8; // devuelve true
```

Utilizando un tercer signo de igual no sólo comparamos los valores sino también el tipo de dato:

```
dato === "5"; // devuelve false porque un dato es un número y el otro es un texto
dato === 5; // devuelve true
dato !== "5"; // devuelve true porque un dato es un número y el otro es un texto
dato !== 5; // devuelve false
```

¿Y para que comparar cosas? Porque los llamados códigos condicionales son fundamentales a la hora de utilizar cualquier lenguaje de programación, sin ellos … nada sería posible.

¿Y que es un código condicional? Es un conjunto de instrucciones que se ejecutarán o no dependiendo de un dato o conjunto de datos y para determinar eso, "comparamos" ese dato con algo.

La instrucción que nos permite hacer eso es **if** (si) y la usamos de este modo:

```
if(condicion){
    // acá hacemos algo
}
```

Entonces, todo lo que está entre las llaves se ejecutará siempre que la condición sea verdadera.

Para establecer esa condición usamos los llamados operadores lógicos que nos permiten evaluar utilizando AND (**&&**), OR (**||**) NOT (!).

Supongamos que dato1 = 2 y dato2 = 10:

```
(dato1 < 10) && (dato2 == 1)
```

será *false* porque el primer término es verdadero pero el segundo no lo es y AND sólo devuelve *true* si ambos son verdaderos

```
(dato1 < 10) || (dato2 == 1)
```

será *true* porque el primer término es verdadero pero el segundo no lo es y OR devuelve *true* si uno cualquiera es verdadero

El otro operador lógico es NOT (!) que devuelve *true* si algo no es verdadero:

```
!(dato1 == dato2)
```

devuelve *true* porque ambas variables son distintas

Entonces, volvamos a las condiciones. Supongamos que queremos que algo se ejecute si una variable tiene un valor superior a 10:

```
if (numero>10) {
    // esto se ejecuta si la variable numero es mayor que 10
}
```

Pero podemos agregar una instrucción más llamada **else** que se ejecutará si la condición es falsa:

```
if (numero>10) {
    // esto se ejecuta si es verdad (la variable  numero es mayor que 10)
} else {
    // esto se ejecuta si es falso (la variable  numero es menor o igual a 10)
}
```

Muchas veces veremos que ciertas condiciones **if** se escriben sin llaves pero ese una costumbre poco recomendable porque el código se hace menos legible.

Como muchas otras instrucciones, podemos anidar las condiciones cuantas veces se nos ocurra aunque siempre debemos revisar bien la secuencia para no equivocarnos:

```
if (numero>10) {
    // esto se ejecuta si es verdad (la variable numero es mayor que 10)
} else {
    if(numero==0){
        // esto se ejecuta si la segunda condición es verdadera (la variable es cero)
    } else {
        // esto se ejecuta si la otra condición es falsa
        // ya sabemos que la variable es menor a 10 y no es cero)
    }
}
```

El mismo ejemplo lo podemos simplificar usando **else if**:

```
if (numero>10) {
    // esto se ejecuta si es verdad (la variable numero es mayor que 10)
} else if (numero==0){
    // esto se ejecuta si la segunda condición es verdadera (la variable es igual a 0)
} else {
    // esto se ejecuta si la otra condición es falsa
    // ya sabemos que la variable es menor a 10 y no es cero)
}
```

Las condiciones pueden ser sencillas como las de los ejemplos anteriores o más complejas:

```
var numero = 123;
var palabra = "hola";

if (numero>10 && (palabra!="texto" || palabra!="Fulano") {
        // esto se ejecuta si la condición es verdad
}
```

¿Se ejecutará o no?

Veamos la condición; está separada en dos partes con el operador AND:

```
numero>10
        &&
(palabra!="texto" || palabra!="Fulano")
```

así que se ejecutará si la variable *numero* es mayor que 10 (en el ejemplo lo es) pero, el AND nos obliga a ver si la segunda parte también es cierta.

En esa segunda parte, usamos OR y dice que palabra no debe tener el valor *"texto"* ni el valor *"Fulano"* y sabemos que tiene el valor *"hola"* por lo tanto, también es cierta.

Moraleja, la primera condición es verdadera (*true*) y la segunda también por lo tanto, las instrucciones dentro del **if** se van a ejecutar.

Cuando tenemos que evaluar varias condiciones, en lugar de utilizar **if...else** repetidamente, podemos recurrir a la función **switch** que permite distribuir las condiciones de modo más claro. La sintaxis elemental es:

```
switch(condicion) {
        case resultado1:
                // código a ejecutar
                break;
        case resultado2:
                // código a ejecutar
                break;
        default:
                // código a ejecutar
}
```

La condición es cualquiera, tal como la usaríamos con **if**; luego, enumeramos con **case** los posibles resultados de esa condición y dentro de cada uno de ellas agregamos las instrucciones para ese caso y terminamos con la palabra **break** que termina la función **switch**.

En la última usamos la palabra **default** indicando que allí se ejecutará cualquier condición no enumerada. Un ejemplo:

```
var nombre = "Fulano";
switch(nombre) {
        case "":
                alert("no existe");
                break;
        case "Fulano":
                alert("el nombre es el que se buscaba");
                break;
        default:
                alert("no se encontró el nombre buscado");
                // código a ejecutar
}
```

¿Y si queremos que cierto código se ejecute con varias condiciones? Los pondríamos en varios case uno debajo del otro:

```
case "Fulano":
case "Mengano":
case "Zutano":
        alert("el nombre es el que se buscaba");
        break;
```

Una forma especial de establecer condicionales es el llamado operador ternario que es una simplificación de **if...else**.

```
var dato = (condición) ? verdadero : falso;
```

El operador ternario evalúa la condición; si la condición es verdadera devuelve el primer valor y si es falsa el segundo.

Un ejemplo:

```
var numero = 20;
var resultado = (numero>100) ? "mayor" : “menor”;
```

La variable *resultado* contendrá la palabra *"menor"*.

Ver, mostrar, probar

Podemos tratar de entender qué es una variable o los tipos de datos y hasta podemos tratar de comprender las condiciones leyendo línea por línea y deducir qué hacen pero llega un momento en que eso se torna demasiado teórico, no vemos nada, no podemos probar nada y necesitamos que eso que hemos hecho pueda verificarse de algún modo.

Aunque JavaScript no tiene una función específica para eso, disponemos de algunas alternativas que pueden ayudarnos.

Quizás, la más sencilla y útil es usar la función **alert()** que nos mostrará una ventanita de alerta con un contenido que podemos definir nosotros.

Si escribimos esto:

```
alert("hola");
```

veremos ese texto en la ventanita que podremos cerrar como cualquier otra.

VER REFERENCIAS [Las ventanas de alerta]

En un ejemplo previo nos preguntábamos si se ejecutaría la función o no y no hacía falta deducir nada, hubiera bastado colocar una ventana de alerta:

```
var numero = 123;
var palabra = "hola";

if (numero>10 && (palabra!="texto" || palabra!="Fulano") {
	alert("true: SE EJECUTA");
} else {
	alert("false: NO SE EJECUTA");
}
```

¿Y cómo podríamos probar ese código? El método sencillo sería crear una página *web* sencilla y colocarlo en una etiqueta **<script>**:

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8">
		<meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
	</head>
	<body>
		<script>
			// acá colocamos las instrucciones
		</script>
	</body>
</html>
```

Otro método es utilizar las consolas de depuración que tienen la mayoría de los navegadores.

En ese caso, en lugar de usar la función **alert()** utilizaríamos algo así:

```
console.log(datos);
```

Tanto **alert()** como **console.log()** admiten que el contenido sea un texto o dato específico como el contenido de una variable:

```
var palabra = "hola";
alert(palabra);
```

o combinar ambas cosas:

```
alert(palabra + " : un saludo");
```

En una página *web* podemos utilizar otra función llamada **document.write()** que "escribe" en la página, justo allí donde es ejecutada. Por ejemplo:

```
<script>
      document.write("hola");
</script>
```

Al abrir la página veremos ese texto escrito. Incuso, podríamos complicarlo un poco más, formateando ese texto::

```
document.write("<p style='color:red;text-align:center;'>hola</p>");
```

No es mucho más útil que lo anterior pero ya no solo estamos escribiendo un texto sino que que estamos escribiendo una etiqueta HTML.

¿Para qué podría servir esto? Es obvio que no tiene mucho sentido usarlo de esta manera pero, al igual que con los métodos anteriores, tenemos la posibilidad de escribir cosas "literales" o usar variables.

```
var saludo = = "hola";
document.write(saludo);
```

Bucles

Así como las condiciones permiten que determinadas instrucciones se ejecuten o no, los bucles permiten que se ejecuten una cantidad de veces determinada. Para esto, usamos **for** con la siguiente sintaxis:

```
for (inicio; condicion; paso) {
    // esto se ejecuta varias veces
}
```

donde *inicio* es la variable con el valor inicial y *condicion* es la variable con el valor final; ambas son las que definen la cantidad de veces que se repetirá y *paso* el incremento o decremento de la variable en cada paso (*step*).

Viendo algo concreto es más sencillo:

```
for (var i=0; i<3; i++) {
    alert(i);
}
```

Estamos generando un bucle que se repetirá tres veces ¿cómo lo sabemos? Porque estamos usando una variable como contador (la llamamos *i*) y decimos que ese contador comienza con cero y deberá terminar cuando sea menor que 3.

En cada paso del bucle, la variable se incrementará en 1 porque eso es lo que le estamos diciendo con *i*++

Si ejecutáramos eso veríamos que aparecerían tres ventanas de alerta mostrando los números 0, 1 y 2.

En el ejemplo estamos sumando 1 al contador pero también podemos decrementarlo:

```
for (var i=3; i>0; i--) {
    alert(i);
}
```

Lo que veríamos serían tres ventanas de alerta mostrando los números 3, 2 y 1.

Tampoco es necesario que cada paso del bucle sea 1; podemos definir cualquier otro; por ejemplo:

```
for (var i=0; i<5; i=i+2) {
    alert(i);
}
```

En ese caso, *i*=*i*+2 hace que en cada paso del bucle, la variable *i* se incremente en 2 así, veríamos tres ventanas de alerta mostrando los números 0, 2 y 4.

Un bucle puede ejecutarse tantas veces se desee pero también podemos interrumpirlo cuando lo consideremos necesario. Para esto, existe la instrucción **break**.

```
for (var i=0; i<5; i++) {
	if (condicion) {
		break;
	}
}
```

Otra instrucción que podemos usar dentro de un bucle es **continue**; con ella permitimos que se no se ejecuten más instrucciones pero no se termina sino que se continua con el siguiente valor del contador.

```
for (var i=0; i<5; i++) {
	if (condicion) {
		continue;
	}
	// lo que sigue sólo se ejecutará si la condición es falsa
}
```

Otro tipo de bucle es el que se genera utilizando **while** que también podríamos decir que es un tipo de condicional ya que lo que hace es ejecutar ciertas instrucciones mientras una condición sea verdadera.

```
while (condicion) {
	// código a ejecutar
}
```

Un bucle **do...while** es similar pero debe tenerse cuidado cuando se utiliza ya que las instrucciones internas se ejecutarán por lo menos una vez aunque la condición sea falsa:

```
do {
	// esto se ejecutará al menos una vez
}
while (condicion);
```

Hay un último tipo de bucle llamado **for...in** que se utiliza con los datos de tipo **Object** para acceder a sus propiedades.

```
for (variable in objeto) {
	// código a ejecutar
}
```

Así escrito es confuso pero, lo que hará es recorrer todas las propiedades del objeto y asignar al parámetro *variable* el nombre de cada una.

Un ejemplo concreto; supongamos que tenemos este objeto:

```
var obj = {nombre:"Fulano",id:"1234567",n:10};
```

Utilizaremos el bucle para poder conocer cuales son sus propiedades y que valores tiene cada una de ellas.

```
for (item in obj) {
        var propiedad = item; // este es el nombre de la propiedad
        var valor = obj[item]; // este es el valor de la propiedad
        // en cada paso del bucle, item será  nombre id y n
        // y obj[item] sus valores: "Fulano", "1234567" y 10
}
```

Funciones

Como todo en una página *web*, las instrucciones JavaScript son ejecutadas por los navegadores una a una, en el mismo orden en que han sido escritas.

Esto es importante porque si añadimos un *script*, este comenzará a ejecutarse apenas sea cargado a menos que esas instrucciones estén incluidas dentro de lo que llamamos una función. Por ejemplo:

```
<script>
        alert ("este es un mensaje");
        function ejemploFuncion() {
                alert ("y este es otro mensaje");
        }
</script>
```

Escrito así, la ventana mostrará *"este es un mensaje"* apenas se haya cargado la página y no ocurrirá nada más, el resto, lo que está dentro de una función, permanecerá allí, a la espera que se la ejecute.

Una función no es otra cosa que un grupo de instrucciones que hacen "algo". Y su sintaxis es:

```
function nombre() {
        // acá colocamos las instrucciones
}
```

La creamos usando la palabra **function** seguida del nombre y paréntesis. El nombre puede ser cualquier cosa y sigue las mismas reglas que las variables (letras, números, guiones, etc).

Uno de los detalles más importante de las funciones es que son capaces de recibir datos enviados, procesarlos y devolver otros. A los datos enviados solemos llamarlos parámetros, se enumeran dentro de los paréntesis separados por comas y pueden ser de cualquier tipo.

```
function ejemplo(parametro1, parametro2, ..., parametroN) {
        // acá colocamos las instrucciones
}
```

En el siguiente ejemplo tenemos una función que calcula la superficie de un rectángulo y la muestra:

```
function superficie(ancho,alto) {
        var area = ancho * alto;
        alert("la superficie es: " + area);
}
```

Claro que si escribimos eso no veremos nada en absoluto porque el código dentro de una función sólo se ejecuta cuando es "llamado" por algo. Ese algo puede ser un evento (por ejemplo un *click* en un botón) u otro código JavaScript.

En el ejemplo, podríamos ejecutarla agregando algo así:

```
superficie(10,5);
```

Simplemente, escribimos su nombre y entre paréntesis, los parámetros que requiere que pueden ser valores o variables:

```
var ancho = 10;
var alto = 5;
superficie(ancho,alto);
```

Si usamos variables, no es necesario que sean las mismas que las que se definen en la función, esta, simplemente toma los valores que recibe y los asigna a las variables definidas:

```
var dato1 = 10;
var dato2 = 5;
superficie(dato1,dato2);
```

Esto ocurre porque dentro de la función, las variables definidas son "locales" es decir, no existen fuera de ella.

Veamos que significa esto:

```
<script>
	function superficie(ancho,alto) {
		var area = ancho * alto;
		alert("la superficie es: " + area);
	}
	var dato1 = 10;
	var dato2 = 5;
	superficie(dato1,dato2);
</script>
```

Veremos una ventana de alerta que dirá *"la superficie es: 50"*.

Pongamos una alerta más al final:

```
alert("la variable area es = "+area);
```

No la veremos porque la variable *area* sólo existe dentro de la función (es local) y cuando esta termina, deja de existir. Por el contrario, *dato1* y *dato2* son variables "globales" porque al estar fuera de una función pueden ser usadas en cualquier parte.

Las funciones también pueden devolver un valor.

Supongamos el siguiente ejemplo:

```
function item(indice) {
	var ejemplo = ["primero", "segundo", "tercero"];
	return ejemplo[item];
}
var dato = item(1);
alert (dato);
```

Dentro de esa función tenemos un *array* con datos; el parámetro que le enviamos es un número (el índice a uno de los elementos del *array*) y nos devuelve el dato que mostraremos en una ventana y dirá *"segundo"*.

La instrucciones dentro de una función pueden ser todo lo complejas que sean necesarias e incluso, ejecutar otras funciones; la única limitación es que sólo pueden devolver un único valor usando la palabra **return**.

Cualquier **return** dentro de una función la termina y no es necesario que devuelva un valor. En este ejemplo el primer **return** termina la función y no se ejecuta nada:

```
function ejemplo(dato) {
	return;
	return dato/10;
}
```

En una página *web*, las funciones también pueden ser ejecutadas mediante eventos como **onclick**, **onmouseover**, etc. Para hacer esto, incluimos esos eventos como atributos de una etiqueta HTML:

```
<script>
	function ejemplo() {
		alert("hola");
	}
</script>
<a href="#" onclick="ejemplo();">click acá</a>
```

Otra forma de llamar a una función sería esta:

```
<a href='javascript:ejemplo();">click acá</a>
```

Tampoco es necesario que la etiqueta sea un enlace, pude usarse casi cualquier otra:

```
<span onclick='ejemplo();">click acá</span>
```

Las funciones son útiles porque pueden ser usadas una y otra vez con distintos argumentos.

VER REFERENCIAS [Un poco más sobre las funciones]

Manos a la obra

Hemos visto variables, tipos de datos, condiciones, bucles y funciones; con eso, ya tenemos los instrumentos básicos para poder empezar a hacer pruebas e ir ampliando el conocimiento de este lenguaje a medida que lo vayamos necesitando.

Como estamos trabajando con páginas *web* debemos entender que, cuando esta se carga, se genera algo denominado DOM (Document Object Model); un conjunto de objetos que no son otra cosa que las etiquetas HTML.

Por lo tanto, debería ser posible usar JavaScript para agregar, modificar o leer el contenido de esos objetos, de sus propiedades, sus atributos, métodos y eventos.

Podría parecer confuso pero, para decirlo de manera simple: un objeto es ... como un objeto. Una mesa es un objeto y como tal tiene propiedades, es de madera o de metal, tiene una cierta altura, un color; lo mismo ocurre con las etiquetas HTML, de alguna forma, son objetos rectangulares que tienen propiedades, un ancho, un alto, un color de fondo, están en cierta posición de nuestra página y así como podemos modificar algunas propiedades de una mesa, también podemos modificar algunas propiedades de las etiquetas y para eso debemos indicarles con exactitud a cuál nos estamos refiriendo.

Por ejemplo, en una página *web* puede haber muchas etiquetas ** ** y si nosotros queremos usar JavaScript para cambiar algo de una de ellas, debemos identificarla con claridad y para eso existe el atributo **id** que nos sirve para darle un nombre exclusivo a una etiqueta cualquiera de tal modo que podamos manipularla:

```
<div id="contenedor" style="color: red; width: 100px;">
        un texto cualquiera
</div>
```

Entre otras cosas, el **id** nos permite darle propiedades con CSS con **<style>**:

```
#contenedor {
        /* lista de propiedades */
}
```

pero también nos permitirá acceder a él desde JavaScript usando alguno de los métodos disponibles.

El más común para es llamado **getElementById()** que usamos de este modo:

```
document.getElementById(id);
```

donde el parámetro es el nombre del atributo **id** del elemento que queremos evaluar.

En el ejemplo, podríamos guardar ese dato en una variable de este modo:

```
var obj = document.getElementById("contenedor");
```

y luego leer alguna de sus propiedades CSS como su ancho o su color si estuvieran definidas con el atributo **style**:

```
var elAncho = obj.style.width;
var elColor = obj.style.color;
```

Así como las leemos, también podemos escribirlas o agregarlas:

```
obj.style.color = "green"; // ahora será de color verde en lugar de rojo
```

Esta es la idea básica que se usa para mantener oculto un contenido y mostrarlo cuando se hace *click* en algún botón; simplemente se permuta la propiedad **display** y eso lo podemos hacer con una función:

```
<script>
	function permutar(elID) {
		// en una variable, guardamos el objeto/etiqueta
		var obj = document.getElementById(elID);
		// una condición: ¿esa etiqueta es visible?
		if (obj.style.display == "block") {
			// si la respuesta es SI (true) cambio la ocultamos
			obj.style.display = "none";
		} else {
			// si la respuesta es NO (false) la mostramos
			obj.style.display = "block";
		}
	}
</script>

<botton onclick="permutar('contenedor');">click acá para ver el contenido</a>
<div id="contenedor" style="display:none;">... el contenido ...</div>
```

Al hacer *click*, se ejecutará una función llamada *permutar* a la que le enviaremos el **id** de la etiqueta. Allí, usando una condición evaluamos la propiedad **display** y la invertimos.

Como hemos dicho, esa es la utilidad de las funciones, permitir que un solo código ejecute cosas similares; en este caso, podríamos tener varios ** ** con **id** diferentes y mostrarlos u ocultarlos a voluntad.

Ahora que estamos probando cosas concretas hay que tener cuidado con el orden en que agregamos los *scripts*.

No es cierto que siempre deben ser incluidos en el **<head>** ni es cierto que siempre deben ser incluidos al final de la página; todo depende de lo que hagan.

Una regla elemental es la siguiente: no podemos ejecutar un *script* que modifique nuestra página si eso que vamos a modificar aún no se ha creado. Como una página *web* se crea de manera secuencial, línea por línea tal cual se ve en el código fuente, esto sería erróneo:

```
<script>
	getElementById("ejemplo").style.display="none";
</script>
<div id="ejemplo">.....</div>
```

ya que el ** ** llamado *ejemplo* se crea después del *script* y cuando este se ejecuta, aún no existe.

Lo correcto sería esto:

```
<div id="ejemplo">.....</div>
<script>
	getElementById("ejemplo").style.display="none";
</script>
```

Si se ha entendido el concepto de que las etiquetas pueden tener un atributo llamado **id** que las identifica, y que ese nombre que le damos debe ser único, ya tenemos una de las herramientas básicas para trabajar con JavaScript porque son muchas las cosas que podemos hacer a partir de eso.

Por ejemplo, supongamos que hemos agregado una imagen de este modo:

```
<img id="demo1" src="URL_demo1">
```

Con JavaScript podemos "leer" sus propiedades y una de ellas, la más obvia, es el atributo **src** que contiene la dirección URL de esa imagen que podríamos leer así:

```
var miurl = document.getElementById("demo1").src;
```

y como podemos leerla, también podemos escribirla:

```
document.getElementById("demo1").src = "URL_demo1";
```

entonces, no resultaría complicado crear una función que ejecutamos con un botón para que, al hacer *click* allí, la imagen cambie:

```
<script>
	function cambiar() {
		document.getElementById("demo1").src = "URL_otra_imagen";
	}
</script>

<img id="miimagen1" src="URL_demo1">
<button onclick="cambiar();">click para cambiar la imagen</button>
```

No tiene mucho sentido ¿no? Mejor, usar una función más general y para eso, deberíamos usar más variables es decir, enviar a la función, datos que varíen, en este caso, podríamos usar dos, uno con el **id** para que la función cambie el contenido de cualquier imagen que se nos ocurra y otro con la URL de la imagen que quisiéramos cambiar:

```
<script>
	function cambiar(id, url) {
		document.getElementById(id).src = url;
	}
</script>

<img id="demo2" src="URL_demo2"/>
<img id="demo3" src="URL_demo3"/>
<button onclick="cambiar('demo2','URL_una_imagen);">cambio una</button>
<button onclick="cambiar('demo3','URL_otra_imagen');">cambio otra</button>
```

Siempre necesitamos "algo" que ejecute la función, un "evento", por ejemplo, un *click* en alguna parte pero, no hace falta que sea un enlace, también puede ser la misma imagen:

```
<img onclick="cambiar('demo4','URL_ejemplo');" id="demo4" src="URL_demo4">
```

¿Es necesario usar siempre un **id**? Pues si y no, depende de lo que quisiéramos hacer. En este caso, podríamos hacer una función que no requiera ese dato porque en JavaScript hay varias formas de identificar las etiquetas y si nos basamos en el último ejemplo, donde el evento está en la etiqueta misma, en realidad, podemos decirle a nuestra función que "esa" es la etiqueta a la que nos referimos y para decirle eso, usamos la palabra **this**:

```
<script>
	function cambiar(cual, url) {
		cual.src = url;
	}
</script>

<img onclick="cambiar(this,'URL_otra_imagen');" src="URL_mi_imagen">
```

Bueno, está bien pero ¿y si quiero restaurar la imagen original? Como la magia no existe y el navegador no piensa, lo que debemos hacer es decírselo explícitamente, métodos hay miles, cada uno para un caso específico, para una necesidad particular.

Una forma simple sería usar otros eventos, en lugar de usar **onclick** podríamos usar **onmouseover** y **onmouseout** que se ejecutan cuando ponemos el puntero del ratón encima de una etiqueta o cuando lo quitamos:

```
<img
	onmouseover="cambiar(this,'URL_imagen_2');"
	onmouseout="cambiar(this,'URL_imagen_1');"
	src="URL_imagen_1"
>
```

¿Y si queremos usar el **onclick** si o si? En este ejemplo, la función leerá cuál es la dirección URL de la imagen que está visible y luego, usando condiciones, la cambiará; si es la primera pondrá la segunda y si es la segunda pondrá la primera:

```
<script>
	function permutarimagen(cual,url) {
		var imagen1 = "URL_imagen_original";
		var imagen2 = "URL_otra_imagen";
		var imagenactual = cual.src;
		if(imagenactual==imagen1) {
			cual.src = imagen2;
		} else  {
			cual.src = imagen1;
		}
	}
</script>

<img onclick="permutarimagen(this);" src="URL_imagen_original">
```

VER REFERENCIAS [Propiedades de las etiquetas HTML]

El objeto document: leer y escribir

Como vimos, con **getElementById()** podemos "leer" una etiqueta HTML e incluso, cambiar algunas de sus características pero JavaScript nos permite cambiar su contenido y para eso existe **document.write()**; sin embargo, esto no es suficiente y difícilmente lo podremos utilizar.

Otra forma mucho más amigable es usar la propiedad **innerHTML**; con ella no sólo podemos leer el contenido de un elemento HTML sino también cambiarlo. Un ejemplo:

```
<div id="ejemplo">contenido original</div>

<script>
	var contenedor = document.getElementById("ejemplo");
	contenedor.innerHTML = "cualquier otro texto";
</script>
```

Entonces, tenemos que la forma clásica de hacer algo con una etiqueta es usar:

```
document.getElementById(id);
```

y sus propiedades genéricas son:

```
document.getElementById(id).atributo; // lee el contenido de ese atributo
document.getElementById(id).atributo = valor; // escribe un atributo

document.getElementById(id).innerHTML ; // lee el contenido de esa etiqueta
document.getElementById(id).innerHTML = valor ; // escribe un contenido

document.getElementById(id).style.propiedad; // lee el contenido de esa propiedad
document.getElementById(id).style.propiedad = valor; // escribe una propiedad
```

Hasta acá hemos identificado elementos HTML utilizando su **id** pero, hay otras maneras de hacerlo aunque son un poco más complejas ya que no tendremos un único resultado sino varios en un *array*.

La función **getElementsByTagName()** nos devuelve la lista de etiquetas de cierto tipo que se indica en el parámetro.

Por ejemplo, si tuviéramos algunos párrafos, imágenes y cualquier otro HTML:

```
<p>el primer texto</p>
<div><p>el segundo texto está es un div</p></div>
<p class="demo">el tercer texto</p>
<img src="URL_imagen1">
<p>el cuarto texto</p>
<img class="demo" src="URL_imagen2">
```

Podríamos listar esos elementos:

```
<script>
	// guardamos la lista de etiquetas <p>
	var listaParrafos = document.getElementsByTagName("p");
	// como es un array, podemos saber el contenido de cualquier item
	alert(listaParrafos[1].innerHTML);
	/*
		listaParrafos[1] = "el primer texto"
		listaParrafos[1] = "el segundo texto está es un div"
		listaParrafos[1] = "el tercer texto"
		listaParrafos[1] = "el cuarto texto"
	*/
	// y lo mismo con las imágenes
	var listaImagenes = document.getElementsByTagName("img");
	alert(listaImagenes[0].src);
	/*
		listaImagenes[0] = "URL_imagen1"
		listaImagenes[1] = "URL_imagen2"
	*/
	// y el <div>
	var listaDivs = document.getElementsByTagName("div");
	alert(listaDivs[0].innerHTML);
	/*
		listaDivs[0] = "<p>el segundo texto está es un div</p>"
		noten que el contenido del <div> incluye la etiqueta <p>
	*/
</script>
```

La función **getElementsByClassName(clase)** es similar excepto que lo que se obtiene es una lista con cualquier etiqueta que tenga una clase determinada.

```
var listaDemo = document.getElementsByClassName("demo");
alert(listaDemo[0].innerHTML);
```

La ventana dirá *"el tercer texto"* pero ¿qué dirá si queremos ver el segundo item?

```
alert(listaDemo[1].innerHTML);
```

Nada de nada ya que la clase se encuentra en una etiqueta **<img>** y por lo tanto, en ella no hay contenido alguno aunque podríamos leer su dirección URL:

```
alert(listaDemo[0].src);
```

Esto puede ser un problema pero hay otra posibilidad que es usar la función **querySelectorAll()** que nos permite seleccionar los elementos de manera más especifica, usando su clase, atributos, etc.

```
var listaExacta = document.querySelectorAll ("p.demo");
alert(listaExacta[0].innerHTML);
```

sólo devolverá una lista con las etiquetas **<p>** que tengan la clase *.demo*.

Por último, JavaScript permite acceder a una serie de las llamadas colecciones que no son otra cosa que las listas de determinado tipo de etiquetas HTML que se pueden consultar del mismo modo que en los casos anteriores ya que son *arrays*:

document.embeds enumera las etiquetas **<embed>**
document.forms enumera las etiquetas **<form>**
document.images enumera las etiquetas **<img>**
document.links enumera las etiquetas **<a>** y **<area>** que tengan el atributo **href**
document.scripts enumera las etiquetas **<script>**

Otras colecciones son más específicas:

document.anchors enumera las etiquetas **<a>** que tengan el atributo **name**
document.title devuelve el contenido de la etiqueta **<title>**

Y estas otras son muy genéricas:

document.documentElement permite identificar la etiqueta **<html>**
document.head permite identificar la etiqueta **<head>**
document.body permite identificar la etiqueta **<body>**

Si bien las colecciones pueden ser útiles en ciertas condiciones, las que podríamos emplear con más frecuencia son **document.head** y **document.body** ya que nos van a permitir agregar contenidos de forma dinámica.

Más propiedades del objeto document

El objeto **document** también posee propiedades que nos permiten acceder a información más general.

Por ejemplo, hay una serie de propiedades que devuelven la dirección URL de la página: **document.URL** y **document.documentURI**.

En la práctica, es probable que no veamos diferencias entre ambas ya que darán el mismo resultado pero, **document.URL** sólo devuelve la dirección de documentos HTML, en cambio, **documentURI** se utiliza cuando el documento no es HTML (por ejemplo XML).

La propiedad **document.domain** devuelve el dominio o subdominio de la página y excluye el protocolo y cualquier archivo interno.

Por ejemplo: *http://subdominio,.dominio.com/paginaweb.html*
devolverá *"subdominio,.dominio.com"*

La propiedad **document.lastModified** devuelve la fecha y hora en que el documento ha sido modificado por última vez en formato: *"mes/dia/año hora:minutos:segundos"*.

La propiedad **document.readyState** devuelve un texto que indica el estado en que se encuentra la carga de un documento y posee uno de estos valores:

uninitialized si aún no ha comenzado la carga
loading si se está cargando
loaded si ha sido cargado
interactive si se ha cargado lo suficiente como para que el usuario haga algo
complete cuando se ha completado la carga

La propiedad **document.referrer** devuelve la dirección URL de la página desde donde se ha cargado el documento.

Así que, consultando el dato de **document.referrer** sabremos si se ha ingresado desde otro sitio, desde algún enlace externo o si se llegó desde un buscador en cuyo caso también nos dirá qué estaba buscando el usuario.

¿Para que podría servir esto? Muchos lo utilizan para mostrar determinado contenido en función del acceso; por ejemplo, es común que ciertos sitios coloquen publicidad cuando se ingresa a través de buscadores pero no lo hagan si se ingresa de otro modo.

```
if (document.referrer!="") {
        alert("gracias por visitarnos desde " + document.referrer);
}
```

Verificamos que ese dato exista ya que si se ingresa de modo directo o a través de los marcadores, devolverá una cadena vacía.

Otra forma de usarlo es para redirigir a otra página:

```
if (document.referrer = 'http://deDondeVienen") {
        location.href = "http://aDondeRedirigimos";
}
```

Supongamos que tenemos una lista con las direcciones web que nos interese detectar y una lista de mensajes asociados a cada una de esas direcciones:

```
var referencias = [
        "//www.google.com",
        "//bing.com",
        "//twitter.com"
];

var mensaje = [
        "utilizando el buscador de Google",
        "utilizando el buscador de Microsoft",
        "desde un enlace de Twitter"
];

// verificamos si el visitante llega desde alguna de ellas
for (i in referencias) {
        if(document.referrer.indexOf(referencias[i])>-1) {
                alert(mensaje[i]);
        }
}
```

La propiedad **document.cookie** devuelve los nombres y valores de las *cookies*.

Una *cookie* es un dato que guarda el navegador en el dispositivo del usuario y, de ese modo, la página, puede acceder a ciertos datos almacenados previamente.

VER REFERENCIA [Cookies]

Eventos

"Los eventos son eso que pasa cuando algo pasa."

En el caso de las páginas *web* pueden ser pulsar un botón del ratón, moverlo, enviar un formulario, tocar una tecla, etc; y son lo que nos permite interactuar con ella, agregándolos como atributos en una etiqueta y dándoles un "valor" que será una o más instrucciones de JavaScript; por ejemplo:

```
<button onclick="alert('hola')">click acá</button>
```

Otra forma de asignar un evento a una etiqueta es usar JavaScript mismo:

```
<button id="ejemplo">click acá</button>

<script>
	function saludar(){
		alert("hola")
	}
	document.getElementById("ejemplo").onclick = saludar;
</script>
```

Ya hemos visto que el evento más común es **onclick** y con él podemos ejecutar una función pero también colocar instrucciones de modo directo. En este ejemplo tres botones que cambian el color de fondo de la página:

```
<button onclick="document.bgColor='lightblue'">lightblue</button>
<button onclick="document.bgColor='lightyellow'">lightyellow</button>
<button onclick="document.bgColor='lightgreen'">lightgreen</button>
```

Además de **onclick** hay muchos eventos que responden a acciones ejecutadas con el ratón.

Por ejemplo, acá estamos usando **onmouseover** y **onmouseout**. El primero se activa cuando se pasa el cursor del ratón encima del elemento y el segundo cuando se lo quita.

```
<a href="#" onmouseover="this.style.color='red';" onmouseout="this.style.color='';">ver</a>
```

En este caso, simplemente cambiamos el color y luego lo eliminamos.

Otros dos eventos sencillos de comprender son **onload** y **onunload**. El primero es utilizado para ejecutar una rutina JavaScript después que la página o imagen haya sido cargada por completo:

```
<body onload="alert('hola')">

<img src="URL_imagen" onload="alert('la imagen ha sido cargada')">
```

El segundo se lanza inmediatamente después que se abandona la página:

```
<body onunload="alert('adios')">
```

Los eventos pueden ejecutar varias acciones simultáneamente. Esto se consigue separando cada una con un punto y coma. Por ejemplo:

```
<button onclick="alert('hola'); alert('adios');">ejemplo</button>
```

VER REFERENCIAS [Eventos en JavaScript]

Window

Así como existe un objeto **document** que es la referencia del DOM (*Document Object Model*) y nos permite acceder a las etiquetas HTML, también existe un objeto **window** que es una referencia a la ventana del navegador.

De hecho, el objeto **document** es parte del objeto **window** por lo tanto, **document.getElementById()** y **window.document.getElementById()** son exactamente lo mismo.

Como todo objeto, **window** posee propiedades, alguna de las cuales son muy útiles. Por ejemplo **window.innerWidth** y **window.innerHeight** devuelven el ancho y alto de la ventana del navegador expresada en pixeles sin incluir barras de desplazamiento y herramientas; es decir, el espacio real donde se muestra nuestra página *web*.

Además, **window.outerWidth** y **window.outerHeight** devuelven los mismos datos pero incluyendo esas barras y bordes.

Las propiedades de **window** son muchísimas y no todas ellas están estandarizadas, algunas, sólo son accesibles en determinados navegadores lo cual hace que todo sea un poco más confuso que lo habitual pero a no desesperar, sólo unas pocas tienen una utilidad práctica. Por ejemplo:

window.history contiene los datos de las páginas visitadas

VER REFERENCIAS [Historial]

window.navigator contiene los datos del navegador

VER REFERENCIAS [Detectar el navegador]

A su vez, todas ellas poseen propiedades y métodos específicos que no tiene sentido enumerar porque sólo agregaríamos más confusión. Lo recomendable es saber que existen y, llegado el caso, buscar información más específica cuando lo creamos necesario. De todos modos, algunos ejemplos pueden aclarar el asunto.

Con **window.open()** podemos abrir una ventana con un documento:

```
window.open("URL_pagina","nombre_ventana");
```

o bien mostrar un documento que hayamos creado con JavaScript:

```
var demo = open("","nueva_ventana");
demo.document.write("<html><head></head><body>EJEMPLO</body></html>");
```

En este ejemplo el primer parámetro de **window.open** esta vacío porque no hay dirección URL y podemos escribir en ella; de esta forma se genera y muestra un nuevo documento HTML.

Si se quisieran insertar imágenes en esa nueva ventana hay que asegurarse de poner las propiedades **height** y **width** en la etiqueta **<img>**; de otra forma no se verán o la página se desvanecerá de alguna manera.

Con **window.close()** podemos cerrar una ventana que hayamos creado previamente.

Uno de los eventos de **window** más utilizados es **window.onload** que nos permite ejecutar instrucciones JavaScript sólo cuando la página haya sido cargada y de ese modo, evitar que algo termine en un error al no haberse creado el elemento HTML.

```
window.onload = function() {
    // esto se ejecutará cuando la página haya terminado de cargarse
}
```

El objeto **window.screen** contiene los datos de la pantalla del usuario (y no debe confundirse con la ventana del navegador).

window.screen.width y **window.screen.height** son el ancho y alto de la pantalla expresada en pixeles

window.screen.availWidth y **window.screen.availHeight** son el ancho y alto de la pantalla descontando bordes, barras etc.

Y como muchas de las propiedades y métodos de window, podemos consultarlas de modo directo:

```
alert("ancho = " + screen.width + "/ alto = " + screen.height;
```

El objeto **window.location** contiene los datos con la ubicación de la página actual y las propiedades más útiles son:

window.location.href devuelve la URL completa de la página
window.location.hostname devuelve el nombre del dominio
window.location.pathname devuelve el *path* (sin el dominio)
window.location.protocol devuelve el protocolo (*http*: o *https*:)

window.location además, nos permite redirigir una página, es decir, cargar otra; para eso, basta establecer la dirección URL ya sea cambiando el valor de la propiedad **href**:

```
window.location.href = "URL_pagina";
```

o empleando el método **assign()**:

```
window.location.assign("URL_pagina");
```

VER REFERENCIAS [Más sobre el objeto window]

Arrays, arreglos o como se llamen

Ya vimos un ejemplo de este tipo de datos y si bien parecen complicados, en realidad, basta saber que un *array* es una lista de datos que se guardan en una sola variable y a los cuales podemos acceder mediante su número de orden.

```
var ejemplo = ["primero", "segundo", "tercero"];
```

Lo más importante a tener en cuenta es que esos números de orden comienzan siempre con cero así que si tenemos tres items, estos serán el 0 el 1 y el 2.

```
ejemplo[0] = "primero";
ejemplo[1] = "segundo";
ejemplo[2] = "tercero";
```

Un mismo *array* puede contener datos de distinto tipo (valores, textos) e incluso otros *arrays*.

Aunque la forma más sencilla de crear un *array* es la que se muestra en el ejemplo, enumerar los valores entre corchetes y separarlos con comas, también se pueden crear usando **new Array()**:

```
var ejemplo = new Array("primero", "segundo", "tercero");
```

¿Y si no conocemos los items que irán porque los calcularemos? Podemos crearla vacía:

```
otro_ejemplo = [];
```

Por supuesto, así como podemos leer su contenido, también podemos modificarlo:

```
ejemplo[2] = "último item";
```

Para añadir un item podríamos hacer lo mismo pero esto puede complicarnos la vida ya que, si el número que ponemos se saltea índices, quedarán datos indefinidos; lo mejor, entonces, es usar el método **push()** que lo agregará al final sin que necesitemos saber cuál es la cantidad de items que haya:

```
ejemplo.push('agregado');
```

¿Y si queremos saber la cantidad de items? Para es existe la propiedad **length**:

```
ejemplo.length; // devolverá el número 3
```

Usando **length** podemos crear un bucle para leer un *array*:

```
for (var i = 0; i < ejemplo.length; i++) {
   alert(ejemplo[i]);
}
```

Noten que el bucle comienza con *i=0* y termina con *i<ejemplo.length* porque el último índice es siempre un número menos que la cantidad total.

Los *arrays* poseen varios métodos con los cuales podemos manipularlas; ya vimos que **push()** agrega un nuevo item al final; por el contrario, con **pop()** podemos eliminar el último item:

```
ejemplo.pop(); // se elimina el item "tercero"
```

Usando una variable podemos eliminar ese item y guardar su valor:

```
var dato = ejemplo.pop(); // se elimina el item "tercero" y se guarda en dato
```

Los métodos **unshift()** y **shift()** hacen lo mismo que **push()** y **pop()** pero agregan o eliminan un item al inicio del *array*, desplazando los otros:

```
ejemplo.unshift("agregar otro"); // agrega un item al inicio
ejemplo.shift(); // elimina el primer item
```

¿Y si queremos agregar o eliminar algo en cualquier otra posición? Para eso existe **splice()**; podemos agregar uno:

```
ejemplo.splice(1, 0, "texto 1");
```

o varios al mismo tiempo:

```
ejemplo.splice(1, 0, "texto 1", "texto 2");
```

Y de modo similar, podemos eliminar uno o varios items:

```
ejemplo.splice(1, 1); // elimina el segundo item
```

En ambos casos, el primer parámetro define el índice (la posición) donde agregamos el o los items; el segundo, siempre es cero cuando agregamos o la cantidad cuando eliminamos; el tercero sólo se usa al agregar y es donde se enumeran los nuevos valores, separados con comas.

slice() copia parte de un *array* y la guarda en otra:

```
var ejemplo = ["primero", "segundo", "tercero", "cuarto"];
var otro = ejemplo.slice(1);
```

El parámetro indica a partir de cuál índice se copia y en el ejemplo, el a*rray* otro contendrá *["segundo", "tercero", "cuarto"]* y el *array* ejemplo permanecerá sin modificar.

Eventualmente, puede tener un segundo parámetro indicando hasta qué item copiar pero sin incluirlo:

```
var ejemplo = ["primero", "segundo", "tercero", "cuarto"];
var otro = ejemplo.slice(1,3);
```

En este caso, el *array* otro contendrá *["segundo", "tercero"]* porque extraemos desde el item 1 hasta el item 3 pero sin incluir este último; o sea, extraemos los items 1 y 2.

concat() crear un array uniendo otras:

```
var arr1 = [1,2];
var arr2 = [3,4];
var nueva = arr1.concat(arr2); // contendrá [1,2,3,4]
```

y como la cantidad de *arrays* a unir es infinita, basta separarlas con comas:

```
var arr3 = [5,6];
var otra = arr1.concat(arr2,arr3); // contendrá [1,2,3,4,5,6]
```

¿Qué pasará si las unimos al revés?

```
var arr1 = [1,2];
var arr2 = [3,4];
var nueva = arr2.concat(arr1); // contendrá [3,4,1,2]
```

Moraleja: se unen en el orden en que lo indiquemos:

```
var nueva = arr2.concat(arr3,arr1); // contendrá [3,4,5,6,1,2]
```

toString() convierte un array en una cadena de texto donde cada item será separado por una coma:

```
var arr1 = [1,2,3];
var texto1 = arr1.toString(); // contendrá "1,2,3"

var arr2 = ["primero", "segundo", "tercero"];
var texto2 = arr2.toString(); // contendrá "primero,segundo,tercero"
```

join() hace lo mismo pero admite un parámetro en el cual podemos establecer el separador:

```
var arr3 = [1,2,3];
var texto3 = arr3.join(" / "); // contendrá "1 / 2 / 3"
```

reverse() invierte el orden de un *array*:

```
var arr1 = [1,2,3];
arr1.reverse(); // ahora contendrá [3,2,1];
```

sort() ordena un array alfabéticamente:

```
var ejemplo = ["zoo", "agua", "mundo"];
ejemplo.sort(); // ahor acontendrá ["agua", "mundo", "zoo"]
```

Si queremos ordenar valores numéricos hay que emplear métodos más “sofisticados”:

```
var numeros = [3, 2, 1, 4];
numeros.sort(function(a, b){return a – b}); // contendrá [1, 2, 3, 4]
numeros.sort(function(a, b){return b - a}); // contendrá [4, 3, 1, 1]
```

Para JavaScript *arrays* y *objects* son el mismo tipo de dato y sólo se diferencian en que los primeros usan números como índices y los segundos usan textos.

```
var arr = [1,2,3];
var obj = {nombre : "Fulano", id : "1234567", n : 10};

typeof arr; // devolverá object
typeof obj; // también devolverá object
```

El método **isArray()** nos permite detectar si la variable es un *array* o no lo es:

```
Array.isArray(arr); // devuelve true porque el parámetro es un array
Array.isArray(obj); // devuelve false porque el parámetro no lo es
```

indexOf() devuelve el número índice de cierto item:

```
var ejemplo = ["primero", "segundo", "tercero", "cuarto", "quinto"];
var dato = ejemplo.indexOf("segundo"); // contendrá el número 1
var otro = ejemplo.indexOf("quinto"); // contendrá -1 porque no existe
```

Podríamos indicar desde dónde comenzar la búsqueda indicando ese índice como segundo parámetro:

```
var otro = ejemplo.indexOf("segundo", 2);
```

En ese ejemplo devolverá *-1* porque empezamos a buscar desde el item *"tercero"*.

La búsqueda la podemos hacer al revés, empezando desde el último item usando **lastIndexOf()**:

```
var otro = ejemplo.lastIndexOf("segundo");
```

Obviamente, si en el *array* sólo hay un dato igual al que buscamos, hacerlo desde el inicio o desde el final dará el mismo resultado. Sin embargo, si hay más de uno si ya que en ambos casos, se devolverá el primero que se encuentre:

```
var ejemplo = ["primero", "segundo", "tercero", "segundo", "quinto"];
ejemplo.indexOf("segundo"); // contendrá el número 1
ejemplo.lastIndexOf("segundo"); // contendrá el número 3
```

VER REFERENCIAS [Otros métodos de los arrays]

Esa cosa llamada objeto

Ya vimos que tratamos a las etiquetas HTML como objetos y este tipo de dato es algo que también podemos crear a voluntad.

Los objetos son elementos que pueden contener conjuntos de pares de nombres claves y valores asociados. Es decir, son *arrays* donde en lugar de haber números de índice, hay nombres claves que pueden ser cualquier palabra y cuyo valor puede ser cualquier tipo de dato: un número, una cadena, un *array*, una función o incluso otro objeto.

Así como existen objetos predefinidos por el mismo lenguaje y objetos asociados a los elementos del documento, también pueden ser creados en forma manual.

Se crean como cualquier variable, dándoles un nombre y enumerando su contenido entre llaves:

```
var obj = {
        nombre : "Fulano",
        saluda : function() {
                alert("hola");
        },
        id : "1234567",
        n : 10
};
```

o simplemente, dejándolos vacíos para agregarles contenido dinámicamente:

```
var obj = {};
```

Las propiedades son los “nombres” de cada clave de un objeto a las cuales accedemos separando el nombre del objeto del nombre de la propiedad con un punto o bien con el nombre de la propiedad entre corchetes y comillas:

```
var dato1 = obj.nombre; // contendrá el string "Fulano"
var dato2 = obj.n; // contendrá el número 10

var dato1 = obj["nombre"]; // contendrá el string "Fulano"
var dato2 = obj["n"]; // contendrá el número 10
```

Un método es una función asociada a un objeto y ya vimos que aquellos propios de JavaScript como **document**, tienen métodos predefinidos. En el caso de los objetos creados por nosotros, también podemos definir métodos propios:

```
obj.saluda(); // mostrará una ventana con el texto "hola"
```

Es muy pero muy importante entender que, a diferencia de otro tipo de datos, “copiar” objetos no es lo mismo que “copiar” cualquier variable. Por ejemplo:

```
var a = 10;
```

```
var b = a;
b = b + 50;
```

Al ejecutar eso, primero copiamos el valor de *a* en *b* y luego incrementamos el valor de *b*. El resultado será que *a* seguirá siendo 10 y b será 60. Es decir, al copiar *a* en *b*, hemos duplicado las variables preservando la primera.

Por el contrario:

```
var a = {nombre : "Fulano", id : "1234567", n : 10};
var b = a;
b.nombre = "cualquier otro";
```

Al ejecutar eso, primero copiamos el valor de *a* en b y luego cambiamos una propiedad de *b*. El resultado será que tanto *a* como *b* habrán cambiado. Es decir, ambas son en realidad el mismo objeto con distinto nombre y las modificaciones se reflejarán en ambos.

Así como podemos cambiar el valor de cualquier propiedad, también podemos agregarlas con la misma sintaxis:

```
var obj = {nombre : "Fulano", id : "1234567", n : 10};
obj.extra = "cualquier otra cosa";
```

Para eliminar una propiedad usamos la palabra **delete**:

```
delete obj.extra;
```

Hay un tipo de bucle especial que nos permite recorrer las propiedades de un objeto:

```
var obj = {nombre : "Fulano", id : "1234567", n : 10};
for (item in obj) {
        alert(obj[item]);
}
```

El bucle tomará cada propiedad, la llamará *item* (o como más nos guste) y mostrará su valor: *"Fulano", "1234567"* y *10*.

¿Y que pasaría si ejecutáramos esto?

```
for (item in obj) {
    alert(item);
}
```

No nos mostraría su valor sino el nombre de la propiedad: *"nombre", "id"* y *"n"*.

Si bien podemos crear un objeto de modo literal, a veces, cuando necesitamos que este tipo de datos sea utilizado en distintas variables, es mejor crear ese tipo con una función:

```
function mi_objeto(dato1, dato2, dato3) {
        this.nombre = dato1;
        this.id = dato2;
        this.n= dato3;
}
```

Que luego, podríamos usar así:

```
var objDEMO = new mi_objeto("Fulano", "1234567", 10);
var objOTRO = new mi_objeto("Mengano", "9876543", 22);
```

Este tipo de solución tiene un inconveniente, no es posible agregar nuevas propiedades o métodos dinámicamente, para hacerlo, debemos agregarlas en la función.

Manejando las etiquetas HTML

Crear, agregar, eliminar o modificar etiquetas o sus atributos es la base de casi cualquier rutina de JavaScript así que nos toca ver alguno de los métodos que tiene el objeto **document** para hacer esto.

El método **createElement()** crea una etiqueta HTML indicando su tipo como parámetro:

```
var parrafo = document.createElement("p"); // crea una etiqueta <p>
```

Hasta ahí, la etiqueta está vacía; si quisiéramos agregarle algún texto usaríamos el método **createTextNode()** así que seguimos:

```
var contenido = document.createTextNode("texto de ejemplo");
```

Ahora ya hemos creado ambas cosas pero una debe estar dentro de la otra y para eso usamos el método **appendChild()**:

```
parrafo.appendChild(contenido);
```

Y nos falta agregarla a la página *web* así que otra vez usamos **appendChild()**. Por ejemplo, así la agregamos al final del **<body>**:

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

También podemos insertar ese nuevo elemento en cualquier otro lado, para ello, basta identificarlo con **getElementById()**, **getElementsByTagName()**, etc.

Suponiendo que tenemos un ** ** cuyo atributo **id** es ejemplo, esto lo insertaría dentro:

```
document.getElementById("ejemplo").appendChild(parrafo);
```

Si queremos eliminar un elemento usamos **removeChild()**:

```
parrafo.removeChild(contenido);
```

Para reemplazar un elemento usamos **replaceChild()** pero necesitamos dos pasos:

```
var nuevo = document.createTextNode("otro texto");
parrafo.replaceChild(nuevo, contenido);
```

Todos esos métodos se utilizan muy a menudo y se basan en el concepto de nodo.

Todas las etiquetas HTML son nodos porque se anidan; están unas dentro de otras y basta identificar una cualquiera para acceder a sus propiedades y métodos.

Veamos un ejemplo:

```
<header>EJEMPLO</header>
<div id="ejemplo" class="contenedor">
        <h3>título</h3>
        <p>el primer texto</p>
        <a id="enlace" href="#">
                <img src="URL_imagen">
        </a>
        <p>el segundo texto</p>
</div>
<footer>ACA TERMINAMOS</footer>
```

Identificamos un elemento mediante su **id**:

```
var elDIV = document.getElementById("ejemplo");
```

A partir de acá podemos obtener algunos datos y en todos los casos usamos la misma sintaxis:

```
elemento.propiedad;
```

Primero, vamos a verificar que el elemento tenga nodos:

```
elDIV.hasChildNodes(); //  devolverá true si los tiene o false si no los tiene
```

Con **children()** obtendremos un *array* (colección) con cada nodo dentro del ** **:

```
var c = elDIV.children; // devolverá [h3,p,a,p]
var len = elDIV.children.length; // devolverá la cantidad (4)
var uno = elDIV.children[1]; // devolverá la primera etiqueta <p>
```

El contenido (los elementos hijo) de ese ** ** pueden ser identificados usando **children()** y el número índice del *array* o el método **item()**:

```
var uno = c.item(2); // devolverá lo mismo que elDIV.children[1]
```

También hay dos propiedades que nos permiten saber cuál es el primero y el último: **firstChild** y **lastChild** aunque esto no siempre será útil porque tomarán como nodos, los espacios en blanco y las tabulaciones; por eso, es mejor usar **firstElementChild** y **lastElementChild**.

Además, otras dos propiedades permiten saber cuales son los elementos que están "arriba" y "abajo": **previousSibling** y **nextSibling** pero otra vez, como en el caso anterior es mejor usar **previousElementSibling** y **elDIV.nextElementSibling**.

```
elDIV.firstElementChild; // devolverá la etiqueta <h3>
elDIV.lastElementChild; // devolverá la segunda etiqueta <p>
elDIV.nextElementSibling; // devolverá la etiqueta <header>
elDIV.previousElementSibling; // devolverá la etiqueta <footer>
```

Las propiedades **parentNode** o **parentElement** devolverán el elemento padre:

```
var elENLACE = document.getElementById("enlace");
elENLACE.parentNode; // devolverá la etiqueta <div id="ejemplo">
```

Hay que tener muy presente que en todos esos casos, lo que se devuelve es un objeto; si sólo quisiéramos saber cuál es la etiqueta, usaríamos **tagName**:

```
elDIV.tagName; devolverá el string "DIV"
elENLACE.tagName; devolverá el string "A"
```

La propiedad **textContent** lee o modifica el texto de un nodo:

```
elDIV.children[1].textContent; // devolverá "el primer texto"
elDIV.children[1].textContent = "nuevo primer texto"; // cambiará el texto
```

También podemos "copiar y pegar" elementos, para eso está el método **cloneNode()** que admite dos parámetros: *true* si queremos incluir su contenido y *false* (es el valor por defecto) si no queremos:

```
var copia = elDIV.children[1].cloneNode(true); // clonamos la primer etiqueta <p>
elDIV.appendChild(copia); // la agregamos al final del <div>
// aparecería otro <p>el primer texto</p> antes de </div>
```

Si hubiéramos usado **cloneNode()** o **cloneNode(false)** sólo veríamos ** ** ya que no se copiara el contenido.

Podemos verificar si un elemento contiene a otro usando el método **contains()**:

```
elDIV.contains(elENLACE); // devolverá true
```

Otras propiedades muy usadas son las que muestran las dimensiones y la posición:

elemento.clientWidth y **elemento.clientHeight** devuelven el ancho y alto del elemento incluyendo su *padding*.

Cuando los elementos tienen alguna propiedad **overflow** que hace que su dimensión se reduzca, esas dos propiedades sólo devolverán el tamaño de la parte visible. Para conocer el tamaño total, usaríamos **elemento.scrollWidth** y **elemento.scrollHeight**.

elemento.offsetWidth y **elemento.offsetHeight** devuelven el ancho y alto del elemento incluyendo su *padding* y bordes.

elemento.clientLeft y **elemento.clientTop** devuelven la distancia horizontal y vertical del elemento con respecto a su propio borde.

elemento.offsetLeft y **elemento.offsetTop** devuelven la distancia horizontal y vertical del elemento con respecto al borde de la ventana.

Al igual que en un caso anterior, si los elementos tienen alguna propiedad **overflow**, los valores sólo indicarán la parte visible. Para conocer la posición real usaríamos **element.scrollLeft** y **element.scrollTop**.

Manejando los atributos HTML

Lo mismo que hemos hecho con las etiquetas HTML podemos hacerlo con sus atributos.

Primero vamos a verificar si una etiqueta tienen atributos y esto podemos hacerlo de dos manera, consultado si tienen cualquier atributo con **hasAttributes()**:

```
elDIV.hasAttributes(); // será true porque tiene un atributo id y otro class
elENLACE.hasAttributes(); // será true porque tiene un atributo id y otro href
elDIV.children[0].hasAttributes(); // será false porque <h3> no tiene atributos
```

La segunda opción es usar **hasAttribute()** colocando como parámetro un atributo:

```
elDIV.hasAttribute("id"); // será true porque tiene un atributo id
elDIV.hasAttribute("class"); // será false porque no tiene un atributo class
```

Los atributos los creamos o modificamos con **setAttribute()** que requiere dos parámetros, el primero es el nombre del atributo y el segundo su valor:

```
elDIV.setAttribute("title","este es el título");
```

del mismo modo lo podemos eliminar con **removeAttribute()**:

```
elDIV.removeAttribute("title");
```

y leer cualquier atributo con **getAttribute()**:

```
elENLACE.children[0].getAttribute("src"); // nos devolverá la URL de la imagen
```

Si quisiéramos leer todos los atributos usaríamos **attributes** que nos devolvería un *array* con la lista:

```
elENLACE.attributes; // devolvería [href="#", id="enlace"]
elENLACE.attributes[1]; // devolvería id="enlace"
```

Algunos atributos específico pueden accederse de manera directa porque hay propiedades para ellos que simplifican la tarea a la hora de escribir códigos. Veamos alguno de ellos:

elemento.className lee o modifica el atributo **class**
elemento.contentEditable lee o modifica el atributo **contenteditable**
elemento.dir lee o modifica el atributo **dir**
elemento.id lee o modifica el atributo **id**
elemento.lang lee o modifica el atributo **lang**
elemento.title lee o modifica el atributo **title**
elemento.style lee o modifica el atributo **style**

Esta última merece alguna explicación extra porque es muy importante entender su uso ya que el CSS de una página es fundamental.

El objeto **elemento.style** es el estilo particular de una etiqueta determinada.

Supongamos que la etiqueta **<h3>** tuviera un atributo **style**:

```
<h3 style="color:red;font-size:24px;">título</h3>
```

Podríamos leer ese atributo con:

```
var estilo = elDIV.children[0].style;
```

nos devolvería un *array* con dos datos:

```
elDIV.children[0].style[0]; // devolverá "color"
elDIV.children[0].style[1]; // devolverá "font-size"
```

que no nos servirían de mucho pero cada propiedad CSS tiene su contraparte en JavaScript que, por lo general, tienen el mismo nombre aunque si la propiedad CSS usa guiones, estos desaparecen y la primera letra de la segunda palabra comienza con mayúsculas. Por ejemplo:

backgroundColor equivale a **background-color**
marginLeft equivale a **margin-left**

Entonces, para cambiar el color y el tamaño de la fuente de esa etiqueta simplemente usamos esto:

```
elDIV.children[0].style.color = "black";
elDIV.children[0].style.fontSize = "64px";
```

Si la propiedad CSS no existe, podemos agregarla:

```
elDIV.children[0].style.border = "2px solid red";
```

y para eliminarla, la dejamos vacía:

```
elDIV.children[0].style.border = "";
```

VER REFERENCIAS [Las propiedades CSS en JavaScript]

Manejando los eventos HTML

Manipulamos etiquetas, atributos y también se pueden "simular" algunos eventos:

elemento.click() simula un *click* del ratón
elemento.focus() establece el foco del elemento
elemento.blur() elimina el foco del elemento

Si eso no es suficiente, el método **addEventListener()** es la forma de adosar eventos a cualquier elemento y la sintaxis es esta:

```
elemento.addEventListener(evento, funcion);
```

donde *evento* es el nombre del evento sin el prefijo *on* (*click*, *keypress*, etc) y *funcion* el nombre de la función que deseamos ejecutar.

Otra sintaxis posible es esta:

```
elemento.evento = function(){ /* código a ejecutar */ }
```

donde *evento* es el nombre del evento con el prefijo *on* (*onclick*, *onkeypress*, etc).

Cuando queremos agregar parámetros a la función la sintaxis es similar:

```
elemento.addEventListener(evento, function(){myFunction(parametros);});
```

Un ejemplo:

```
elDIV.children[0].onclick = function(){
	alert("HOLA");
}
```

o la variante que hará exactamente lo mismo:

```
function ejemplo(){
	alert("HOLA");
}

elDIV.addEventListener("click", function(){
	ejemplo();
});
```

En ambos casos, cuando se haga *click* en la etiqueta **<h3>** con el título, se abrirá una ventana de alerta.

Los eventos no sólo se pueden agregar a las etiquetas HTML sino a la misma ventana que ya hemos visto que también es un objeto llamado **window**.

Si pusiéramos esto:

```
window.addEventListener("click", ejemplo);
```

La ventana de alerta se mostraría cuando hiciéramos *click* en cualquier parte, algo que no resultaría muy útil así que deberíamos eliminar ese evento adosado y para eso utilizamos el método **removeEventListener()** que tiene una sintaxis similar:

```
elemento.removeEventListener(evento, funcion);
```

Así que en el ejemplo, esto quitaría ese último evento:

```
window.removeEventListener("click", ejemplo);
```

Parece fácil pero eliminar eventos agregados tiene una limitación ya que sólo podemos hacerlo si lo hemos adosado con esa sintaxis; si lo hubiéramos agregado así, no funcionaría:

```
window.addEventListener("click", function(){
        ejemplo();
});
```

Cuando lanzamos eventos, la función "recibe" al objeto y este, puede ser referenciado con la palabra **this**; por ejemplo:

```
elDIV.children[0].onclick = function(){
        this.style.backgroundColor = "yellow";
}
```

o bien:

```
elDIV.children[0].addEventListener("click", function(){
        this.style.backgroundColor = "yellow";
});
```

En ese caso, this es la etiqueta **<h3>** y su color de fondo será amarillo cuando se hace *click* en el título.

Sin embargo esta sintaxis no funcionaría porque la función "no sabe" qué referencia la palabra **this**:

```
function ejemplo(){
        this.style.backgroundColor = "yellow";
};

elDIV.children[0].addEventListener("click", function(){ejemplo();});
```

Para usar **this**, deberíamos enviarlo como parámetro y tratarla como cualquier otra variable dentro de la función:

```
elDIV.children[0].addEventListener("click", function(){ejemplo(this);});

function ejemplo(elem){
        elem.style.backgroundColor = "yellow";
};
```

VER REFERENCIAS [Eventos en JavaScript]

Timers

En JavaScript, el manejo de los *timers* es fundamental porque a veces es necesario ejecutar algo pasado cierto tiempo o repetir una acción a intervalos regulares.

Para esto hay dos métodos específicos del objeto **window**: **setTimeout()** ejecuta una función pasado cierto tiempo y **setInterval()** ejecuta una función repetidamente, cada cierto tiempo.

La sintaxis de **setTimeout()** es la siguiente:

```
setTimeout(funcion, milisegundos);
```

donde *funcion* es el nombre de la función a ejecutar y *milisegundos* el tiempo (1 segundo = 1000 milisegundos).

```
function ejemplo() {
	alert('HOLA');
}

setTimeout(ejemplo, 2000);
```

En el ejemplo, la ventana con el saludo aparecerá dos segundos después de haberse cargado la página.

Eventualmente, podemos agregar parámetros que se transferirán a la función, agregándolos al final; por ejemplo, acá estamos transfiriendo el texto como parámetro y recibiéndolo en la función:

```
function ejemplo(texto) {
	alert(texto);
}

setTimeout(ejemplo, 2000, "hola otra vez");
```

también lo podemos escribir así:

```
setTimeout(function(){ejemplo("hola otra vez"); }, 2000);
```

No es obligatorio que la función sea externa, podemos incluir todo en la misma instrucción de este modo:

```
setTimeout(function(){alert("hola")}, 2000);
```

Además, como es una función, devuelve un dato que es el un número que identifica al timer así que podríamos usar este tipo de sintaxis:

```
var demoTIMER = setTimeout(ejemplo, 2000);
```

Con el método **clearTimeout()** borramos un *timer* creado con **setTimeout()** y es para eso que se utiliza una variable que debe ser global, es decir, no debe ser declarada dentro de una función para que, de ese modo, podamos acceder a ella en cualquier parte del código:

```
var demoTIMER;
setTimeout(ejemplo, 2000);
clearTimeout(demoTIMER);
```

En el ejemplo no pasará nada porque eliminamos el *timer* inmediatamente y de ese modo, cancelamos su ejecución y el parámetro que colocamos es el nombre de la variable.

El método **setInterval()** tiene las variantes de sintaxis idénticas pero, debe tenerse en cuenta que hace algo distinto ya que la función que se indique, se repetirá una y otra vez:

```
setInterval(funcion, milisegundos);
```

Para detener este tipo de *timer* debemos usar **clearInterval()** con el nombre de la función como parámetro.

El uso de **setInterval()** puede causar ciertos problemas porque se está ejecutando de modo constante; es por ese motivo que usar funciones indirectas da mejor resultado.

Por ejemplo, supongamos que tenemos que verificar algún tipo de dato cada cinco segundos; usamos dos funciones:

```
function ejecutarTIMER(){
	setTimeout(verificar,50000);
}

function verificar(){
	// verificamos que no lo hayamos detenido
	if(stopTIMER==true){
		// esta detenido así que no hacemos nada
		return;
	}
	// ejecutamos la verificación
	// .....
	// y reiniciamos el timer
	ejecutarTIMER();
}

var stopTIMER = false;
ejecutarTIMER();
```

De ese modo, sólo necesitamos el método **setTimeout()** que se ejecutará de modo indirecto con *ejecutarTIMER* y detendremos colocando *stopTIMER=true*.

El teclado y el ratón

Los eventos relacionados con el teclado son bastante complicados ya que no están estandarizados. De todos modos tenemos que resolverlo de la mejor manera posible y para eso, primero debemos recordar que cuando se pulsa una tecla no se ejecuta un evento sino tres: **keydown**, **keypress** y **keyup** que traduciríamos como, la tecla baja, queda abajo y sube.

Saber qué tecla se ha pulsado tampoco es tan sencillo porque depende de lo que quisiéramos saber, una cosa es la tecla en si misma (d y D son la misma) y otra cosa es saber el código del carácter (d y D no son iguales). Además, como si esto fuera poco, también existen combinaciones de teclas usando Alt Ctrl o cualquier otra sin contar que todo eso lo podemos mezclar con los *clicks* del ratón.

Para ver las diferencias entre las distintas propiedades vanos a utilizar una etiqueta **<input>** con un evento **onclick**:

```
<input type="text" onkeydown="teclado(event)">
```

Las primeras propiedades a evaluar son las que nos permiten saber si se han oprimido algunas de las teclas especiales:

altKey devuelve true si se oprime la tecla ALT
ctrlKey devuelve true si se oprime la tecla CTRL
shiftKey devuelve true si se oprime la tecla SHIFT

```
function teclado(event) {
        if (event.altKey) {
                alert("tecla ALT");
        } else if (event.ctrlKey) {
                alert("tecla CTRL");
        } else if (event.shiftKey) {
                alert("tecla SHIFT");
        } else {
                alert("cualquier otra tecla");
    }
}
```

La propiedad **charCode** devuelve el código Unicode del carácter pero, sólo en el evento **keypress** ya que en los otros, el resultado será siempre cero.

Las propiedades **keyCode** y **which** devuelven el código ASCII del carácter en el evento **keypress** y el código Unicode de la tecla en los eventos **keydown** y **keyup** pero su resultado varía según el navegador.

Entonces, ¿queda claro que el tema es confuso? Por suerte, las nuevas versiones van colocando las cosas en su lugar y se recomienda olvidarse de todo ese galimatías y utilizar una propiedad sencilla que debería haber existido siempre.

La propiedad **key** devuelve una cadena de texto con el nombre de la tecla oprimida; así de simple:

```
function teclado(event) {
	alert(event.key);
}
```

Si pulsamos la tecla a devolverá a y si pulsamos A devolverá A. Además, todas las teclas especiales tienen su nombre (*CapsLock*, *Enter*, *Escape*, *Control*, *ArrowUp*, *Home*, *Shift*, *Alt*, *F1*, *Insert*, etc).

Definitivamente, esta es la solución más lógica y deberíamos olvidarnos de cualquier otra.

De todos modos, cuando usamos los eventos del teclado debemos tener en cuenta que se ejecutan con mucha rapidez y que mantener pulsada una tecla puede implicar que se produzcan efectos indeseados.

Una forma de evitar esto es establecer alguna clase de *timer* que impida que se procese un evento antes de haber transcurrido determinado tiempo:

```
var keydelay = false; // definimos una variable global

function teclado(event) {
	if(keydelay) {
		// si el flag es true no hacemos nada
		return;
	}
	// esta en false y lo ponemos en true
	keydelay = true;
	// y ejecutamos una demora de 50 milisegundos antes de ponerlo en false
	setTimeout(function(){keydelay=false}, 50);
	// cuando termina la demora evaluamos el teclado y verificamos las teclas
	alert(event.key);
}
```

Los eventos relacionados con el ratón también poseen propiedades a las que podemos acceder.

La propiedad **button** devuelve un número que indica cuál botón ha sido oprimido:

0 botón izquierdo
1 botón central
2 botón derecho

La propiedad **which** hace lo mismo pero los valores devueltos son otros:

0 ninguno
1 botón izquierdo
2 botón central
3 botón derecho

La propiedad **detail** devuelve la cantidad de veces que se ha hecho *click*.

Hay propiedades que nos permiten conocer las coordenadas gráficas donde se posiciona el ratón o se ejecuta algún evento asociado.

screenX y **screenY** indican la posición respecto a la pantalla del dispositivo; además, **clientX** y **clientY** indican la posición respecto a la ventana del navegador.

Los errores

JavaScript no difiere de cualquier otro lenguaje, habrá errores, será inevitable y el problema intentar que no ocurran (algo que evidentemente uno tratará de hacer siempre) sino descubrir dónde está el error y que lo ha provocado porque las causas pueden ser diversas.

La instrucción **try...catch** nos ayuda a crear códigos donde se verifica la existencia de errores y nos advierte.

```
try {
	// esto se ejecutará si todo esta ok
} catch(err) {
	// esto se ejecutará si hay algún error
}
```

¿Para qué usar algo así? Porque en muchos casos, cuando JavaScript encuentra un error, el resto del código no se ejecuta y se genera un mensaje de error que normalmente no es visible para el usuario.

Veamos un ejemplo; supongamos que queremos ejecutar una función pero esta no existe:

```
mifuncion();
```

No pasará nada pero no sabremos que pasa. Ahora, probemos lo mismo de este modo:

```
try {
	mifuncion();
} catch(err) {
	alert(err.message);
}
```

Veremos una ventana de alerta con el texto *"mifuncion is not defined"* ya que le estamos diciendo que ante cualquier error al tratar de ejecutar el bloque que se encuentra dentro de **try{}** se ejecute el bloque dentro de **catch{}** y en este caso le pedimos que nos muestre el mensaje de error que se encuentra en el objeto **err**.

El objeto **err** tiene una propiedad fundamental llamado **message** que es un texto con la descripción del error pero también se está experimentando con otro que no funciona en todos los navegadores pero es muy útil. La propiedad se llama **lineNumber** y devuelve el número de línea donde se produce el error.

Así que podríamos ampliar nuestro ejemplo para que nos guiara un poco más:

```
var texto =  err.message + "\nen la línea " + err.lineNumber;
alert(texto);
```

Con la instrucción **finally{}** agregamos otra alternativa, un bloque de código que se ejecutará al final sin importar si previamente se ejecutó **try{}** o **catch{}**

```
try {
        // esto se ejecutará si todo esta ok
} catch(err) {
        // esto se ejecutará si hay algún error
} finally {
    // esto se ejecutará al final
}
```

Utilizando **throw** podemos definir errores personales. En este caso, no lo usaríamos para evaluar errores de sintaxis incorrecta o problemas similares sino a la evaluación de datos que queremos controlar o verificar.

La sintaxis es sencilla y el valor puede ser cualquier tipo de dato:

```
throw "texto a mostrar";
```

Veamos un ejemplo:

```
var nombre = "Mengano";
try {
        if(nombre=="") {
                throw "el nombre es obligatorio";
        }
        if(nombre!="Fulano") {
                throw "el nombre es incorrecto";
        }
        alert(nombre);
} catch(err) {
       alert(err);
}
```

Como la variable nombre no es la que necesitamos, se mostrará una ventana de alerta con el texto indicado en **throw**; si nombre es un *string* vacío, el mensaje será otro.

Hay que tener en cuenta que si la variable nombre no existiera, el mensaje de error sería *"ReferenceError: nombre is not defined"*.

Cuando cargamos una página web, en la barra de estado solemos ver el proceso de carga, es decir, los archivos externos que se van agregando. En algunos de ellos, al terminar la carga se nos mostrará la palabra *Listo* pero en otros es posible que se muestre un ícono de advertencia que nos indica un error en alguno de los *scripts*.

Que algunos navegadores no muestren ese ícono no significa que no haya errores así que, de alguna manera tiene su utilidad aunque, para un usuario normal es algo irrelevante y para quien tiene un sitio, suele ser molesto o “poco estético”.

Para un usuario que navega, es un dato que no le interesa ya que nada puede hacer y para un administrador de un sitio, la información que brinda es bastante escasa. Sin las herramientas adecuadas, buscar el error es casi imposible y con las herramientas adecuadas puede resultar engorroso.

Para colmo, algunos de esos errores son imposibles de solucionar ya que son generados por *scripts* sobre los que no tenemos control o no podemos modificar.

Bien, para quien no quiera que eso se muestre, podemos agregar un par de líneas antes de y ocultar las advertencias:

```
function noMostrarErrores() {return true;}
window.onerror=noMostrarErrores;
```

Obviamente, esto no impedirá que el error siga existiendo así que debe tomarse sólo como una forma de esconder la mayoría de ellos bajo la alfombra. Una forma poco sutil pero efectiva.

Ajax

AJAX (*Asynchronous JavaScript and XML*) no es un lenguaje sino una forma de intercambiar datos con un servidor, sin tener que recargar la página. Para eso, se utiliza un objeto de JavaScript llamado **XMLHttpRequest**.

Tiene un nombre raro, de esos que asusta pero simplemente creamos uno como lo podemos hacer con cualquier otro, usando **new** y asignándolo a una variable:

```
var objAJAX = new XMLHttpRequest();
```

Como todo objeto, tiene métodos y propiedades. Los métodos **open()** y **send()** son los que usaremos para solicitar (*request*) que el servidor nos envíe ciertos datos.

El método open() tiene tres parámetros:

```
objAJAX.open(metodo, archivo, FLAGasync);
```

El método es un *string* que puede ser *GET* o *POST*; *archivo* es la dirección URL del archivo que procesará el pedido y *FLAGasync* es *true* si la solicitud se procesará de modo asincrónico o *false* si no lo será.

El método **send()** envía la solicitud que en el caso de GET no tiene parámetros y en el caso de POST el parámetro es un *string* con los datos que se envían.

```
objAJAX.send();
```

Usar *GET* o *POST* depende del tipo de datos a enviar y su volumen. Usamos *GET* cuando no hay datos a enviar o son pocos ya que que en ese caso, los datos deben estar en la dirección URL definida con **open()**.

Por ejemplo, supongamos que requerimos que se nos devuelvan ciertos datos de un usuario que está registrado en una base de datos en el servidor; enviamos su nombre de este modo:

```
objAJAX.open("GET", "archivo.php?nombre=Fulano&id=1234567890", true);
```

Usamos *POST* si la cantidad de datos a enviar tienen un gran tamaño o si necesitamos que haya cierta seguridad en ese intercambio. En ese caso, los datos se incluyen en **send()**:

```
objAJAX.send("nombre=Fulano&id=1234567890");
```

Hay que tener en cuenta que si usamos POST, también debemos agregar un *header HTTP* que indique el tipo de datos que se están enviando y eso se hace con **setRequestHeader()**.

```
objAJAX.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
```

La forma estándar de usar AJAX es asincrónicamente, es decir, procesar esa solicitud, recibir los datos y ejecutar alguna acción sin que se interrumpa el flujo natural de la pagina, procesando todo eso en segundo plano sin tener que esperar que el servidor responda.

Sin embargo, en muchos casos no queda más remedio que esperar esa respuesta para continuar y actuar en consecuencia, entonces es cuando colocamos el valor de *false* en el tercer parámetro de **open()**.

Decimos que AJAX "solicita algo" a un archivo que está en el servidor y este archivo puede ser de cualquier tipo, uno que procese datos como PHP o cualquier otro lenguaje, o uno que simplemente contenga datos ya sean textos planos o XML.

Sean lo que sean, necesitamos saber que "responde" el servidor y para eso hay dos propiedades:

```
var datos1 = objAJAX.responseText; // recibe los datos como una cadena de texto
var datos2 = objAJAX.responseXML; // recibe los datos en formato XML
```

Cuando usamos **open()** con asincronismo *false*, el navegador esperará la respuesta u continuará inmediatamente después de **send()** así que es allí donde deberíamos procesar los datos recibidos:

```
objAJAX.open("GET", "archivo", false);
objAJAX.send();
var datos = objAJAX.responseText;
```

En cambio, cuando la solicitud es asincrónica, deberemos consultar el evento **onreadystatechange** ya que este se ejecutará cuando se haya recibido la respuesta.

```
objAJAX.onreadystatechange = function() {
    // respuesta recibida
};
objAJAX.open("GET", "archivo", true);
objAJAX.send();
```

Pero eso no es suficiente ya que no sabemos si la respuesta es válida; por eso existe la propiedad **readyState** que nos permite saber el estado de la solicitud:

0 indica que no se ha inicializado
1 indica que se ha establecido la conexión
2 indica que el servidor ha recibido la solicitud
3 indica que se esta procesando
4 indica que se ha recibido la respuesta

Además, la propiedad **status** nos devuelve un código; por ejemplo:

200 indica que todo salió bien
404 indica que la página no se encontró
500 indica un error interno del servidor

Teniendo esto en cuenta, al forma tradicional de manejar las respuestas es usar verificar si **readyState** es 4 y **status** es 200 lo que indica que todo está bien y hemos recibido alguna respuesta que podemos procesar:

```
objAJAX.onreadystatechange = function() {
        // respuesta recibida
        if (objAJAX.readyState == 4 && objAJAX.status == 200) {
                // todo OK
                var datos = objAJAX.responseText;
        }
};
```

Obviamente, el uso más importante de AJAX es intercambiar información entre una página *web* y un servidor usando lenguajes como PHP o ASP pero eso está lejos de lo elemental así que nos limitaremos a algo menos espectacular como, por ejemplo, cargar cierto texto o algo similar.

Para eso, lo más simple es tener alguna clase de función genérica que permita solicitar que se nos envíen datos.

```
function procAJAX(url) {
        var objAJAX = new XMLHttpRequest();
        objAJAX.open("GET", url, true);
        objAJAX.onreadystatechange = function () {
                if (objAJAX .readyState == 4 && objAJAX.status == "200") {
                        procesarRESPUESTA(objAJAX .responseText);
                }
        };
        objAJAX.send();
}

function procesarRESPUESTA(respuesta){
        alert(respuesta);
}
```

Simplemente, probemos con un archivo de texto con cualquier contenido al que llamaremos *texto.txt*; al ejecutar:

```
procAJAX("texto.txt")
```

se mostrará una ventana de alerta con el texto de ese archivo.

¿Qué pasaría si ese archivo contuviera HTML? Lo mismo, una vez que recibimos la respuesta, como es un *string*, podríamos usarlo para agregarlo a una etiqueta existente

```
document.getElementById("ejemplo").innerHTML = DATOS;
```

JSON

El formato JSON (*JavaScript Object Notation*) es una de las formas más simples de intercambiar datos entre una página y otra ya sea que se encuentren en el mismo dominio o no.

Su sintaxis es idéntica a los objetos JavaScript aunque un archivo en formato JSON es simplemente un archivo de texto. Creamos uno de ejemplo con este contenido:

```
{
    "nombre": "Fulano",
    "id": "1234567890",
    "detalles": {
        "cantidad": 2,
        "color": "#00ff0f"
    }
}
```

y lo guardamos con el exótico nombre de *json.json* o como se quiera.

Como se ve, los datos se encuentran en pares (nombre:valor) y sólo se diferencian a los objetos de JavaScript en que el nombre siempre debe estar entre comillas.

¿Y cómo lo podemos leer eso desde nuestra página? Usando AJAX así que nos basaremos en lo dicho anteriormente y crearemos una función genérica que procese esa solicitud:

```
function procJSON(url) {
        var objAJAX = new XMLHttpRequest();
        objAJAX.overrideMimeType("application/json");
        objAJAX.open("GET", url, true);
        objAJAX.onreadystatechange = function () {
                if (objAJAX .readyState == 4 && objAJAX.status == "200") {
                        procesarRESPUESTA(objAJAX .responseText);
                }
        };
        objAJAX.send();
}
```

Y como la respuesta recibida es un simple *string*, lo convertimos usando la función **JSON.parse()** de JavaScript:

```
function procesarRESPUESTA(respuesta){
        var datosJSON = JSON.parse(respuesta);
}
```

Si seguimos el ejemplo, ejecutamos:

```
procJSON("json.json");
```

Eso es todo, ahora, tendremos un objeto llamado *datosJSON* que podemos consultar y usar para lo que se nos ocurra:

datosJSON.nombre será "Fulano"
datosJSON.id será "1234567890"
datosJSON.detalles.cantidad será 2
datosJSON.detalles.color será "#00ff0f"

Hay muchos servicios externos que utilizan este formato para enviarnos información que podemos aprovechar; en general, tienen un API (*Application Programming Interface*), que es lo que genera el JSON basándose en ciertos datos que solicitamos y que ellos establecen.

Suelen ser de uso gratuito aunque en la mayoría es obligatorio registrarse y conseguir una clave para acceder a ellos. Por ejemplo, el API de YouTube tiene muchas alternativas pero vamos a usar la más sencilla, le solicitaremos información de un vídeo específico:

https://www.googleapis.com/youtube/v3/videos?id=uB_2yjlkdk8&key=APIKEY&part=snippet

El JSON que nos devolvería sería algo como esto:


```
{
   "kind": "youtube#videoListResponse",
   "etag": "\"I_8xdZu766_FSaexEaDXTIfEWc0/i_gIDmI_mpzu1B0IKi-U5DSoAHI\"",
   "pageInfo": {
      "totalResults": 1,
      "resultsPerPage": 1
   },
   "items": [
      {
         "kind": "youtube#video",
         "etag": "\"I_8xdZu766_FSaexEaDXTIfEWc0/0WoL3K2-xQjYy7MSlT2g0AZe-4Y\"",
         "id": "uB_2yjlkdk8",
         "snippet": {
            "publishedAt": "2014-12-02T15:00:52.000Z",
            "channelId": "UCNcgF3a3mrTZqqop9C17iTQ",
            "title": "Chuck Berry - Back In The USA",
            "description": "Chuck Berry - Back In The USA\n\n ...",
            "thumbnails": {
               "default": {
                  "url": "https://i.ytimg.com/vi/uB_2yjlkdk8/default.jpg",
                  "width": 120,
                  "height": 90
               },
               "medium": {
                  "url": "https://i.ytimg.com/vi/uB_2yjlkdk8/mqdefault.jpg",
                  "width": 320,
                  "height": 180
               },
               "high": {
```

```
                "url": "https://i.ytimg.com/vi/uB_2yjIkdk8/hqdefault.jpg",
                "width": 480,
                "height": 360
            },
            "standard": {
                "url": "https://i.ytimg.com/vi/uB_2yjIkdk8/sddefault.jpg",
                "width": 640,
                "height": 480
            },
            "maxres": {
                "url": "https://i.ytimg.com/vi/uB_2yjIkdk8/maxresdefault.jpg",
                "width": 1280,
                "height": 720
            }
        },
        "channelTitle": "Nostalga Records",
        "tags": [
            "Chuck Berry",
            "Back in the USA",
            "Oldies"
        ],
        "categoryId": "24",
        "liveBroadcastContent": "none",
        "localized": {
            "title": "Chuck Berry - Back In The USA",
            "description": "Chuck Berry - Back In The USA\n\n..."
        }
    }
  }
 ]

}
```


¿Cómo leemos semejante cosa? Del mismo modo que lo hemos hecho con el ejemplo simple pero claro, debemos saber cuál es la estructura de ese archivo y qué significa cada dato. Por lo general, ambas cosas siempre son explicadas por el sitio que proporciona el servicio; en este caso, vamos a ver que hay cuatro elementos principales:


```
{
   "kind": "youtube#videoListResponse",
   "etag": "\"l_8xdZu766_FSaexEaDXTlfEWc0/i_gIDmI_mpzu1B0IKi-U5DSoAHI\"",
   "pageInfo": {...},
   "items": [{...}]
}
```


Dos de ellos, *datosJSON.kind* y *datosJSON.etag* son simples, contienen los textos tal cual se ven. El tercero, *datosJSON.pageInfo* tiene dos valores:

datosJSON.pageInfo.totalResults es 1 porque solo hemos pedido un vídeo
datosJSON.pageInfo.resultsPerPage es 1 porque no hay más paginas

datosJSON.items es distinto, si vemos el JSON notaremos que su contenido está entre corchetes indicando que es *array* de datos que, en este caso solo contiene un elemento y eso quizás confunde pero, es así porque el mismo API se usa para realizar búsquedas y por lo tanto, otro tipo de solicitud podría devolver varios resultados.

De todos modos, basta usar *datosJSON.items[0]* y continuar leyendo: por ejemplo:

datosJSON.items[0].id contendrá el id del vídeo que solicitamos: "*uB_2yjIkdk8*"

datosJSON.items[0].snippet tiene más datos internos que debemos seguir recolectando como por ejemplo:

datosJSON.items[0].snippet.publishedAt es la fecha de publicación
datosJSON.items[0].snippet.title es el título del vídeo
datosJSON.items[0].snippet.description es la descripción
datosJSON.items[0].snippet.tags es un *array* con las etiquetas

Y así seguimos con los datos de las miniaturas por defecto:

datosJSON.items[0].snippet.thumbnails.default.url
datosJSON.items[0].snippet.thumbnails.default.width
datosJSON.items[0].snippet.thumbnails.default.height

o las que tienen la máxima resolución

datosJSON.items[0].snippet.thumbnails.maxres.url
datosJSON.items[0].snippet.thumbnails.maxres.width
datosJSON.items[0].snippet.thumbnails.maxres.height

Ejemplos surtidos

Cargar scripts de manera dinámica

La idea es cargar archivos de JavaScript de manera condicional, sólo cuando nosotros queremos o lo consideremos necesario y, de esta manera, aliviar el peso de las páginas web.

La teoría de esto es muy simple y sólo requerimos un poco de código:

```
function cargarJS(archivo) {
        var nuevo = document.createElement("script");
        nuevo.setAttribute("type", "text/javascript");
        nuevo.setAttribute("src", archivo);
        document.getElementsByTagName("head")[0].appendChild(nuevo);
}
```

Y el archivo lo cargaremos de este modo:

```
cargarJS("URL_archivo");
```

También es usual que el evento **onload** se utilice para cargar *scripts* externos:

```
window.onload=function(){
        var nuevo = document.createElement("script");
        nuevo.setAttribute("type", "text/javascript");
        nuevo.setAttribute("src", archivo);
        document.getElementsByTagName("head")[0].appendChild(nuevo);
}
```

Lo mismo podría hacerse con hojas de estilo, en ese caso, bastaría crear otra etiqueta y sus atributos:

```
function cargarCSS(archivo) {
        var nuevo = document.createElement("link");
        nuevo.setAttribute("rel", "stylesheet");
       nuevo.setAttribute("type", "text/css");
        nuevo.setAttribute("href", archivo);
        document.getElementsByTagName("head")[0].appendChild(nuevo);
}
```

Abrir todos los vínculos en otra ventana

Para eso necesitamos que todos los enlaces tenga el atributo *target="_blank"* pero, si no lo hemos puesto o queremos simplificar la tarea a la hora de escribir el código HTML, podemos hacerlo con JavaScript:

```
function allBLANK() {
	var enlaces = document.getElementsByTagName("a");
	for (var i=0; i<enlaces.length; i++) {
		enlaces[i].target = "_blank";
	}
}
```

Aunque eso podría ser problemático si tenemos enlaces internos que quisiéramos que se abrieran en la misma página entonces, podríamos agregar el atributo sólo si el enlace es a una página externa que comienza con *http://* o *https://*

```
function externalLINKS() {
	var enlaces = document.querySelectorAll('a[href^="http"]', 'a[href^="https"]');
	for (var i=0; i<enlaces.length; i++) {
	enlaces.target = "_blank";
	}
}
```

Y si aún tenemos dudas porque pueden haber otras alternativas, usemos algún atributo personal en los enlaces externos ya que de ese modo, la selección no será automática y podremos personalizarlos gráficamente:

```
<style>
	a[rel="externo"] {
		/* propiedades */
	}
</style>

<script>
	function externalLINKS() {
		var enlaces = document.getElementsByTagName("a");
		for (var i=0; i<enlaces.length; i++) {
			var e = enlaces[i];
			if (e.getAttribute("href") && e.getAttribute("rel") == "external")
			enlaces.target = "_blank";
			}
		}
	}
	window.onload = externalLINKS;
</script>

<a href="direccion_URL" rel="external">vínculo externo</a>
<a href="direccion_URL">vínculo normal</a>
```

Limpiar caracteres indeseados de un texto

Todo dependerá de lo que consideremos caracteres indeseados.

1. Eliminar caracteres < y > para evitar el uso de etiquetas HTML:

```
var regex = /(<([^>]+)>)/ig;
texto = texto.replace(regex,"");
```

2. Cambiar saltos de línea **\n** por etiquetas **
**:

```
texto = texto.replace(/\n/g," <br>");
```

3. Cambiar etiquetas **
** por saltos de línea **\n**:

```
var regex = /<br\s*[\/]?>/gi;
texto = texto.replace(regex, "\n");
```

4. Reemplazar letras acentuadas áéíóú por aeiou:

```
var acentos = {'\u00e1':'a', '\u00e9':'e', '\u00ed':'i', '\u00f3':'o', '\u00fa':'u'}
for (var val in acentos){
        texto = texto.replace(new RegExp(val, "g"), acentos[val]);
}
```

5. Eliminar espacios duplicados:

```
texto = texto.replace(/\s{2,}/g, ' ');
```

6. Eliminar caracteres de idiomas extraños:

```
var nuevoTEXTO =  texto;
for(i=0;i<t.length;i++){
        c = texto.charCodeAt(i);
        if(c>31 && c<256){
                nuevoTEXTO = nuevoTEXTO + String.fromCharCode(c);
        }
}
```

7. Eliminar comillas dobles o simples:

```
texto = texto.replace(/"/g,''); // dobles

texto = texto.replace(/'/g,''); // simples
```

8.Eliminar comillas simples y dobles:

```
texto = texto.replace(/[\'"]/g, '');
```

9. Capitalizar la primera letra de cada palabra

```
function capitalizeSTR(texto){
        return texto.replace(/\w\S*/g, function(texto) {
                return texto.charAt(0).toUpperCase() + texto.substr(1).toLowerCase();
        });
}
```

Generar una demora

```
function delay(tiempo){
	var date = new Date();
	var currDate = null;
	do {
		currDate = new Date();
	} while(currDate-date<tiempo);
}
```

Ejecutar utilizando como parámetro una cantidad de milisegundos (1segundo = 1000 milisegundos):

```
delay(milisegundos);
```

Ampliar imágenes de manera sencilla

La función tomará los datos de la etiqueta **<img>** y al hacer *click* se mostrará en base a los parámetros de ancho y alto mínimos y máximos enviados:

```
function zoom(minANCHO, minALTO, maxANCHO, maxALTO, laIMAGEN){
        ancho = laIMAGEN.style.width;
        alto = laIMAGEN.style.height;
        if((ancho==maxANCHO)||(alto==maxALTO)){
                nuevoANCHO = minANCHO;
                nuevoALTO = minALTO;
        } else {
                nuevoANCHO = maxANCHO;
                nuevoALTO = maxALTO;
        }
        laIMAGEN.style.width = nuevoANCHO;
        laIMAGEN.style.height = nuevoALTO;
}
```

Y luego, lo agregamos en cualquier imagen:

```
<img src="URL_imagen" width="valor" height="valor"
onclick="zoom('valor','valor','valor','valor',this);">
```

Reemplazar direcciones por enlaces

Buscamos direcciones en el texto y las convertimos en enlaces utilizando expresiones regulares:

```
function crearLINKS(t) {
	var urlRegEx = /((([A-Za-z]{3,9}:(?:\/\/)?)(?:[\-;:&=\+\$,\w]+@)?[A-Za-z0-9\.\-]+|
(?:www\.|[\-;:&=\+\$,\w]+@)[A-Za-z0-9\.\-]+)((?:\/[\+~%\/\.\w\-]*)?\??(?:[\-\+=&;
%@\.\w]*)#?(?:[\.\!\/\\\w]*))?)/g;
	var matches;
	matches = t.match(urlRegEx);
	if (matches) {
		for (i=0; i<matches.length; i++) {
			t = t.replace(matches[i], '<a href="'+matches[i]+'">'+matches[i]+'</a>');
		}
	}
	return t;
}
```

Verificar la carga de una imagen

Para determinar las dimensiones de una imagen debemos esperar que el navegador la cargue por lo tanto, si no las conocemos de antemano, en ciertas condiciones es necesario utilizar alguna función extra antes de mostrarlas.

En este caso, tenemos dos funciones. La primera, carga la imagen en la memoria y, usa los eventos **onload** y **complete** para verificar que hay terminado de ser cargada. Cuando ocurre esto, se ejecuta una segunda función que en este ejemplo, simplemente muestra sus dimensiones.

```
function waitImagen(src, callback) {
	var imagen = new Image();
	imagen.src = src;
	if (imagen.complete) {
		callback(imagen);
		imagen.onload = function() {};
	} else {
		imagen.onload = function() {
			callback(imagen);
			imagen.onload = function() {};
		}
		imagen.onerror = function() {
			alert("error al cargar la imagen");
		}
	}
}

function imageREADY(imagen) {
	// la imagen esta disponible y podemos mostrarla
	alert(imagen.width + "x" + imagen.height);
}
```

Esto lo ejecutamos de este modo:

```
waitImagen("URL_imagen",imageREADY);
```

Funciones para generar números aleatorios

```
/* devuelve un numero aleatorio entre cero y un valor máximo
        max = valor máximo
*/
function getRND(max) {
        return Math.floor(Math.random() * max);
}

/* devuelve un numero aleatorio en determinado rango incluyendo los límites
        min y max son los límites
*/
function getRNDrange(min,max) {
   return Math.floor(Math.random() * (max - min + 1)) + min;
}

/* devuelve el indice de un item aleatorio de un array
        arr es el array a evaluar
*/
function getRNDitem(arr) {
        return Math.floor(Math.random()*arr.length);
}

/* devuelve true si se genera un número aleatorio mayor que cierto "porcentaje"
        porcentaje = pseudo-porcentaje
*/
function isALEA(porcentaje){
        var alea = Math.floor(Math.random() * 100);
        return (alea > (100 - porcentaje)) ? true : false;
}
```

Función genérica para agregar página a favoritos

Poner un enlace para que los visitantes puedan agregar la página a los favoritos puede no ser algo tan sencillo como parece ya que los navegadores utilizan códigos diferentes.

Un método sencillo podría ser este:

```
function agregarFavoritos(url, titulo) {
	if (window.sidebar&&window.sidebar.addPanel){
		window.sidebar.addPanel(titulo,url,"");
	} else {
		window.external.AddFavorite(url,titulo)
	}
}
```

Este otro método hace lo mismo pero amplia las posibilidades:

```
function agregarFavoritos(url, titulo) {
	if (window.sidebar && (navigator.appName.indexOf('Explorer') == -1) &&
(navigator.appName.indexOf('Opera') == -1)) {
		window.sidebar.addPanel(titulo,url,'');
	} else if (window.opera && window.print) {
		var fav = document.createElement('a');
		fav.setAttribute('rel','sidebar');
		fav.setAttribute('href',url);
		fav.setAttribute('title',titulo);
		fav.click();
	} else if( document.all) {
		window.external.AddFavorite(url,titulo);
	}
}
```

En ambos casos los parámetros son el titulo con el texto que se mostrará en el enlace agregado y la dirección URL de la página.

Fechas relativas

Una fecha relativa, en lugar de mostrar día, mes y año de cierta entrada, lo que indica es el tiempo transcurrido desde su publicación hasta el momento actual; por ejemplo: “hace 2 minutos” o “dos días atrás” o “hace un año”; cualquier texto que a uno se le ocurra.

Para que este tipo de cosas funcione correctamente lo que debemos usar es una fecha con un formato conocido para poder poder hacer un cálculo preciso que por lo general es algo así: *2010-03-25T15:30:20-03:00*.

¿Qué es eso? El año, el mes, el día; seguido de la letra T que indica la hora, los minutos y los segundos, seguido de un valor que es la zona horaria.

Suponiendo que la variable *dato* contenga esa fecha:

```
var fecha= new Date(dato).getTime()/1000; // esa fecha en segundos
var ahora = new Date().getTime()/1000; // la fecha actual en segundos
var segundos = ahora – fecha; // la diferencia entre ambas fechas
```

Teniendo el valor de los *segundos*, ahora es cuestión de usar aritmética para calcular las fracciones es decir, expresar segundos de otra manera.

```
minutos = segundos / 60
horas = segundos / 60*60 = segundos / 3600
dias = segundos / 60*60*24 = segundos / 86400
semanas = segundos / 60*60*24*7 = segundos / 604800
```

Los meses son más complicados de calcular porque tienen diferente cantidad de días pero, como esto no dee ser algo tan exacto, lo simplificamos asumiendo que todos tienen 31:

```
meses = segundos / 60*60*24*31 = segundos / 2678400
```

Algo similar pasaría con los años bisiestos pero acá no se trata de precisión sino de redondeos así que:

```
años = segundos / 60*60*24*365 = segundos / 31536000
```

```
/* convierte una fecha absoluta en una fecha relativa y devuelve un string
    dato = fecha a evaluar
*/
function fecharelativa(dato) {
    var salida = "";
    dato = dato.replace("a.m.", "AM");
    dato = dato.replace("p.m.", "PM");
    var fecha = new Date(dato).getTime()/1000;
    var ahora = new Date().getTime()/1000;
    if(fecha<=ahora) {
```

```
            var segundos = ahora - fecha;
            var minutos = (parseInt(segundos / 60)).toString();
            var horas = (parseInt(segundos / 3600)).toString();
            var dias = (parseInt(segundos / 86400)).toString();
            var semanas = (parseInt(segundos / 604800)).toString();
            var meses = (parseInt(segundos / 2678400)).toString();
            var anios = (parseInt(segundos / 31536000)).toString();
            if (minutos < 1) {
                  salida = "hace menos de un minuto";
            } else if(minutos == 1) {
                  salida = "hace poco más de un minuto";
            } else if(horas < 1) {
                  salida = "hace " + minutos + " minutos";
            } else if(horas == 1) {
                  salida = "hace poco más de una hora";
            } else if(dias < 1) {
                  salida = "hace " + horas + " horas";
            } else if(dias == 1) {
                  salida = "hace poco más de un día";
            } else if(semanas < 1) {
                  salida = "hace menos de una semana";
            } else if(semanas == 1) {
                  salida = "hace una semana";
            } else if(meses < 1) {
                  salida = "hace " + dias + " días";
            } else if(meses == 1) {
                  salida = "hace un mes";
            } else if(anios < 1) {
                  salida = "hace " + meses + " meses";
            } else if(anios == 1) {
                  salida = "hace un año";
            } else {
                  salida = "hace " + anios + " años";
            }
        }
      return salida;
}
```

Cambiar una palabra por una imagen

Una de las posibilidades del JavaScript es que podemos cambiar cosas muy rápidamente sin que esto sea evidente para los visitantes.

Con ese criterio es que es posible modificar cualquier tipo de texto.

Supongamos que nos gustaría que cada vez que hacemos referencia a nuestro propio sitio, en lugar de aparecer un enlace con un texto quisiéramos que se muestre una imagen como un logo.

Es sencillo de hacer, escribimos el código HTML y listo:

```
<p>
        ...etiam rhoncus iaculis magna ac accumsan ...
        <a href="URL_pagina"><img src="URL_imagen"></a>
         ...et eros mollis hendrerit ...
</p>
```

No hay problema pero ... hay que escribirlo ... eso mismo lo podríamos hacer de manera automática y evitarnos el trabajo de recordar el código; para eso, sólo necesitamos de una función cuya estructura elemental sería algo así:

```
function reemplazarXlogo(id) {
        var bodyText = document.getElementById(id);
        var elTexto = bodyText.innerHTML;
        elTexto = elTexto.replace(/PALABRA/g,'CODIGO_HTML');
        bodyText.innerHTML = elTexto;
}
```

¿Qué hará eso? Una vez que la página es cargada, buscará dentro de algún elemento (el body, un div, un párrafo), una *PALABRA* y la reemplazará por un código HTML que deberemos definir nosotros.

¿Qué palabra? Cualquiera pero es recomendable que usemos una palabra inventada, algo que podamos recordar con facilidad y que no tenga otro posible uso. Por ejemplo, podría usar *MILOGO* y hacer que en lugar de verse esa palabra, se viera una imagen que fuera un enlace a mi sitio:

```
var html = '<a href="http://misitio.com"><img src="URL_imagen"></a>';
elTexto = elTexto.replace(/MILOGO/g,html);
```

El HTML entonces lo podríamos escribir así:

```
<p>
        ...etiam rhoncus iaculis magna ac accumsan MILOGO et eros mollis hendrerit ...
</p>
```

Cambiar los colores de manera dinámica

Una forma sencilla de cambiar los colores de modo general es buscar ciertas etiquetas dentro de algún contenedor (body, div, etc) , guardarlas en una lista y usar un bucle para cambiar el estilo de todas ellas al mismo tiempo.

Supongamos que lo que queremos cambiar está dentro de un ** ** cuyo **id** es *#contenedor*:

```
// cambiar el color de los enlaces buscando todas las etiquetas <a>
function cambiarColorEnlaces(elcolor) {
	var contenedor = document.getElementById("contenedor");
	var listaENLACES = contenedor.getElementsByTagName("a");
	for(var i=0; i<listaENLACES .length; ++i) {
		listaENLACES[i].style.color = elcolor;
	}
}

// cambio el color de fondo de todas las etiquetas <span>
function cambiarColoFondo(elcolor) {
	var contenedor = document.getElementById("contenedor");
	var listaSPANS = contenedor.getElementsByTagName("span");
	for(var i=0; i<listaSPANS .length; ++i) {
		listaSPANS[i].style.backgroundColor = elcolor;
	}
}

// cambiar el color del borde de ese <div>
function cambiarColorBorde(elcolor) {
	var contenedor = document.getElementById("contenedor");
	contenedor.style.borderColor = elcolor;
}

// cambiar todo eso al mismo tiempo
function cambiarTodos(elcolor) {
	var contenedor = document.getElementById("contenedor");
	var listaENLACES = contenedor.getElementsByTagName("a");
	for(var i=0; i<listaENLACES .length; ++i) {
		listaENLACES[i].style.color = elcolor;
	}
	var listaSPANS = contenedor.getElementsByTagName("span");
	for(var i=0; i<listaSPANS .length; ++i) {
		listaSPANS[i].style.backgroundColor = elcolor;
	}
	contenedor.style.borderColor = elcolor;
}

<a href="javascript:void(0);" onclick="cambiarTodos('#FF0')"> todos amarillos </a>
```

Crear una galería de imágenes paso a paso

Usar JavaScript para cambiar una imagen por otra es tan sencillo como cambiar cualquier otra propiedad.

La idea entonces es usar eso para poder armar una galería de imágenes donde, con un *click*, vayamos mostrando una cantidad indefinida.

Supongamos que tenemos un HTML así:

```
<div class="demoSW">
        <img id="misfotos" src="URL_imagen_1">
        <span id="mistextos"> la primera de las imagenes </span>
        <div><a href="#">anterior</a> | <a href="#">siguiente</a></div>
</div>
```

Lo que vamos a hacer es crear dos *arrays*, la primera contendrá la dirección URL de una serie de imágenes y la segunda, los textos asociados a cada una de esas imágenes:

```
var foto = ["URL_img1", "URL_img2", "URL_img3", "URL_img4", "URL_img5"];

var texto = ["la primera de las imágenes", "esta es la segunda", "esta es al tercera", "y casi vamos a terminar", "la ultima de las imágenes"];
```

Ahora, vamos a crear una función para manipular las etiquetas **<img>** y **** a las que hemos identificado como *misfotos* y *mistextos* con respectivos atributos **id**.

A la función la llamaremos *mover* y le enviaremos un dato. Para decirle que queremos ir hacia atrás le enviaremos un *-1* y para decirle que queremos ir hacia adelante, le enviaremos un *1* así que *mover(-1)* debería mostrar la imagen anterior y *mover(1)* debería mostrar la imagen siguiente.

Para que el ejemplo funcione, bastará cambiar los enlaces y llamar a la función que vamos a crear:

```
<a href="#" onclick="mover(-1);">anterior</a>
.....
<a href="#" onclick="mover(1);">siguiente</a>
```

Pero necesitamos un dato más, debemos llevar la cuenta, saber cuál es la imagen que estamos viendo para poder saber cuál es la siguiente y cuál es la anterior así que la guardaremos en alguna parte y al inicio, su valor será cero ya que la imagen visible es la primera.

```
var cualvemos = 0;
```

En este caso, el bucle será infinito porque pasaremos de la última a la primera cuando usemos el enlace *siguiente* y de la primera a la última cuando usemos el enlace *anterior*.

```
function mover(direccion) {
        var laimagen = document.getElementById("misfotos"); // la etiqueta <img>
        var eltexto = document.getElementById("mistextos"); // la etiqueta <span>
        // ¿cuál es el indice de la última imágenes en nuestra array?
        var ultima = foto.length - 1; // en el ejemplo, será el 4
        // verificamos cuál sería la imagen a mostrar
        var auxiliar = cualvemos + direccion; // se sumará 1 o se restará 1 al índice
        // si el resultado es menor que cero muestra la última
        if(auxiliar < 0) { auxiliar = ultima; }
        // si el resultado es mayor que la última muestra la primera
        if(auxiliar > ultima) { auxiliar = 0; }
        // listo, ahora ya podemos cambiar el dato sin problemas
        cualvemos = auxiliar;
        // ponemos la dirección URL de la imagen en la etiqueta <img>
        laimagen.src = foto[cualvemos];
        // ponemos el texto en la etiqueta <span>
        eltexto.innerHTML = texto[cualvemos];
}
```

Simulando marquees

Desde tiempos inmemoriales existió la etiqueta **<marquee>** pero, como todo en este mundo, su existencia ha sido declarada inaceptable y el HTML5 la ha depreciado y recomienda no utilizarla.

De todos modos, podemos simularla utilizando JavaScript y todos felices.

La idea básica de este ejemplo es tener un elemento con un ancho fijo y un texto largo que se desplazará dentro de modo continuo.

```
<style>
	# divDEMO {
		height: 1.5em;
		overflow: hidden;
		white-space: nowrap;
		width: 200px;
	}
</style>
<div id="divDEMO"></div>

<script>
	var texto="... este es un ejemplo ... ";
	var timer;
	function marquee() {
		document.getElementById("divDEMO").textContent = texto;
		texto = texto.substring(1, texto.length) + texto.charAt(0);
		timer = setTimeout('simumarquee()',100);
	}
</script>
```

Si queremos que se ejecute, usamos:

```
marquee();
```

y si queremos detenerlo usamos:

```
clearTimeout(timer);
```

Mostrar el tiempo de carga de una página

Para quienes gustan de mostrar el tiempo que tarda en cargarse una página web, la solución es sencilla; basta un pequeño *script* que se ejecute con **onload** y algún contenedor donde escribir el resultado:

```
<div id="mostrarTiempo">Cargando ...</div>

<script>
	var startTime = new Date();
	function showElapsedTime() {
		var endTime = new Date();
		var elapsedTime = Number(endTime - startTime);
		var texto = Number(elapsedTime/1000) + " segundos";
		document.getElementById("mostrarTiempo").innerHTML = texto;
	}
	onload=function() {showElapsedTime();}
</script>
```

Eso es todo.

Acordeones

Siempre que queremos ejecutar alguna acción sobre una etiqueta debemos hacer *click* en ella y para interactuar, el CSS no es suficiente, necesitamos JavaScript para que el navegador actué así que, para crear acordeones, hay que combinar ambas cosas.

Un acordeón no es mucho más que una serie de enlaces que nos permiten mostrar y ocultar contenidos de tal manera que sólo uno de ellos es visible; cuando se muestra el seleccionado, los demás se ocultan y de ese modo, el espacio utilizado se optimiza.

Hay muchos métodos para crear esto y este sólo es una demostración del concepto básico.

```
<div id="demoA">
	<a onclick="demoacordeon('primero');" href="#">primero</a>
	<div id="primero" class="visible">
		<p> ..... el contenido ..... </p>
	</div>
	<a onclick="demoacordeon('segundo');" href="#">segundo</a>
	<div id="segundo" class="novisible">
		<p> ..... el contenido ..... </p>
	</div>
	<a onclick="demoacordeon('tercero');" href="#">tercero</a>
	<div id="tercero" class="novisible">
		<p> ..... el contenido ..... </p>
	</div>
</div>
```

Dentro de un ** ** contenedor, ponemos una sucesión de enlaces y otros ** ** con los contenidos, identificados con un **id** exclusivo. Cada enlace ejecutará la función de JavaScript enviando ese atributo como dato.

Para simplificar todo, cada contenido tiene una clase CSS. Si es visible es la clase *.visible* y para deslumbrar con nuestra originalidad, cuando no es visible la llamamos *.novisible*:

```
<style>
#demoA { /* al div contenedor lo dimensionamos y centramos */
	margin: 0 auto;
	width: 500px;
}
#demoA a { /* establecemos las propiedades de los enlaces o botones */
	background-color: #456;
	display: block; font-size: 16px;
	height: 2em;
	line-height: 2em;
	margin: 1px 0;
	padding-left: 20px;
}
#demoA a:hover {color: #eee;}
```

```
/* los contenidos */
#demoA div {
	overflow:hidden;
	text-align: center;
	transition: all 1s ease-in-out;
 }
#demoA div.visible {
	/* el visible tiene borde y una cierta altura */
	border: 1px solid #456;
	height:100px;
}
#demoA div.novisible {
	/* el oculto no tiene borde y su altura es cero */
	border: 1px solid transparent;
	height:0px;
 }
</style>

<script>
	var abierto = "primero"; // definimos el id del que estará abierto al inicio
	function demoacordeon(cual) {
		var mostrar = document.getElementById(cual); // el que vamos a mostrar
		var actual = document.getElementById(abierto); // el que vamos a ocultar
		if(mostrar==actual) {
			// si es el mismo no hacemos nada
			return false;
		}
		// caso contrario permutamos sus clases
		actual.className = "novisible";
		mostrar.className = "visible";
		abierto = cual; // y guardamos el que está abierto
	}
</script>
```

overflow si overflow no

Las propiedades **overflow** nos permiten limitar el espacio en el cual se mostrará algo y de ese modo, poner mucho contenido en un espacio pequeño donde podremos desplazarnos con barras.

Siempre es bueno tener definida esta propiedad en todos los contenedores cuyo ancho deseamos controlar pero, siempre es demasiado tiempo y a veces, en ciertas condiciones, esto nos impide hacer determinadas cosas. Por ejemplo, una alternativa interesante es que algunos sectores se muestran en alguna clase de etiqueta con barras de *scroll* pero, cuando colocamos el puntero del ratón encima, el contenedor se expande avanzando hacia la derecha para facilitar su lectura.

Para hacer esto, basta usar algún *script* que quite esa propiedad temporalmente y luego la restaure.

Este es un ejemplo donde se permuta el ancho del contenedor cada vez que hacemos *click*:

```
<style>
	#demo {
		overflow-x: scroll;
		overflow-y: auto;
		white-space: pre;
		width: 500px;
	}
</style>

<div id="demo" onclick="demoExpandir(this);"> ..... el contenido ..... </div>

<script>
	function demoExpandir(objdemo) {
		var estado = objdemo.style.overflowX;
		if(estado=="visible") {
			objdemo.style.width = "500px";
			objdemo.style.overflowX = "scroll";
		} else {
			objdemo.style.width = "100%";
			objdemo.style.overflowX = "visible";
		}
	}
</script>
```

Slideshow simple

Esta rutina crea un *slideshow* que permite colocar cualquier cantidad de imágenes que se reproducirán a una velocidad dada.

Comenzamos con un contenedor donde se desee que aparezcan las imágenes y definimos cuál será la primera:

```
<div><img id="slideshow" src="URL_imagen1"></div>
```

Y ahora el *script*:

```
<script>
	// la velocidad está expresada en milisegundos (1000 = 1 segundo)
	var velocidad = 1000;
	// en un array colocamos las imágenes a mostrar
	var listaIMAGENES = ["URL_imagen_1", "URL_imagen_2", ... , "URL_imagen_n"];
	var cualIMG = 0; // indice a la primera imagen
	var slideshow = document.getElementById("slideshow");
	function ejecutar() {
		slideshow.src=listaIMAGENES[cualIMG]
		if (cualIMG<listaIMAGENES.length-1) {
			cualIMG++;
		} else {
			cualIMG = 0;
		}
		setTimeout("ejecutar()",velocidad)
	}
	ejecutar();
</script>
```

Sonidos en cualquier parte

Que se puedan agregar sonidos a los enlaces quizás no es algo que atraiga demasiado pero, por suerte, hasta ahora, era algo que estaba bastante limitado porque salvo Internet Explorer, el resto de los navegadores no poseían etiquetas para esto.

El problema, si es que uno ama algunos silencios, es que ahora, esas etiquetas comienza a estar disponibles y han dejado al viejo **<sound>** atrasado.

La etiqueta en cuestión se llama **<audio>** que es muy simple de usar:

```
<audio controls="controls" preload="auto">
	<source src="URL_archivo.mp3"></source>
</audio>
```

Eso escrito tal cual, mostraría el reproductor pero basta eliminar el atributo **controls** para que podamos reproducir el audio sin necesidad de tener visible ese reproductor y por lo tanto, usando JavaScript podemos agregar eventos a una etiqueta cualquiera.

```
<audio id="misonido" preload="auto">
	<source src="URL_archivo.mp3"></source>
</audio>
```

Ahora, usaremos el evento mouseover en una etiqueta cualquiera como un enlace::

```
<a href="#" onmouseover="hacerruido("misonido");">probar el sonido</a>
```

Y sonará si usamos una función que le pase el id de la etiqueta **<audio>**:

```
function hacerruido(id) {
	var audio = document.getElementById(id);
	audio.play();
}
```

Sólo espero que no lo utilicen en demasía.

Simple scroll del fondo

¿Es posible hacer que la imagen de fondo de una página web se mueva con un *scroll*?

Pués, la verdad, sí. Digamos que es una técnica del tiempo en que las cosas que había en internet debían "moverse" todo el tiempo y nos fascinaba que así fuera aunque, por suerte ya se ha abandonado el bamboleo.

Para crear un efecto *scroll* del fondo sólo se necesita una imagen y un pequeño *script*.

Colocamos la imagen a utilizar en el CSS del **<body>**:

```
body {background:#FFFFFF url(URL_imagen) repeat scroll 0 0;}
```

Y luego, el *script*:

```
<script>
	var laPosicionVertical = 0;
	function scrollFondo(maximo) {
		laPosicionVertical = laPosicionVertical + 1;
		if (laPosicionVertical > maximo) {
			laPosicionVertical = 0;
		}
		document.body.style.backgroundPosition = '0 ' + laPosicionVertical + 'px';
	}
	window.onload= function(){
		var scrollTimer = window.setInterval('scrollFondo(500)', 64);
	}
</script>
```

En el llamado a la función pueden cambiarse dos datos que en este ejemplo son:

500 es la altura máxima, generalmente, es similar a la altura de la imagen a usar
64 es la velocidad, un número menor hará un *scroll* más rápido y un número mayor un *scroll* más lento.

Insertar vídeos con sencillez

Es algo habitual que uno quiera automatizar ciertas tareas, sobre todo, aquellas cosas que son rutinarias o repetitivas; por ejemplo, escribir cierto tipo de código o darle forma a cierto tipo de contenido sin tener que estar escribiendo códigos extensos. No es algo que pueda ser respondido de modo genérico, las necesidades son infinitas y por lo tanto, las soluciones también.

La mayoría de los servicios que alojan vídeos nos permiten insertarlos en una página web y nos dan el código correspondiente que en la enorme mayoría de los casos suele ser una etiqueta **<iframe>** donde va variando un dato:

```
<iframe src="URL" width="valor" height="valor"></iframe>
```

donde esa URL es siempre la misma pero varía un dato, el identificador o el nombre del video; por ejemplo:

*http://www.dailymotion.com/embed/video/**dato***
*https://www.facebook.com/video/embed?video_id=**dato***
*http://player.vimeo.com/video/**dato**?title=0&byline=0&portrait=0*
http://vk.com/video_ext.php?dato&hd=1
*http://www.youtube.com/embed/**dato***
*https://vine.co/v/**dato**/embed/simple?related=0&audio=1*

Usemos JavaScript y creemos una función para que podamos insertar los vídeos de distintos servicios de manera más automática. En lugar de insertar el código que ellos nos dan, simplemente colocamos el dato variable; por ejemplo:

```
<div class="mivideo" rel="youtube" id="4fk2prKnYnI"></div>
<div class="mivideo" rel="vimeo" id="33440713"></div>
```

Cada uno de ellos es un ** ** cuyo atributo **class** es *.mivideo*, donde colocamos dos atributos extras: **rel** indicará el tipo de servicio e **id** será el identificador del vídeo que es el que vemos en las páginas de esos servicios.

La clase deberíamos definirla aunque sea con propiedades mínimas:

```
<style>
	.mivideo {
		margin: 0 auto;
		text-align: center;
	}
</style>
```

Veamos el código:

```
var videos = document.getElementsByClassName("mivideo");
for (var i=0; i<videos.length; i++) {
	var laURL = "";
```

```
        var videoID = videos[i].getAttribute("id");
        var servicio = videos[i].getAttribute("rel");
        // de acuerdo al tipo de servicio, armamos la dirección URL
        if(servicio=="youtube") {
                laURL = "http://www.youtube.com/embed/" + videoID;
        } else if(servicio=="vimeo") {
                laURL = "http://player.vimeo.com/video/" + videoID + "?
title=0&byline=0&portrait=0";
        }
        if(laURL) {
                // y creamos el código HTML que en este caso será una etiqueta <iframe>
                var salida = "<iframe width='480' height='360' src='" + laURL + "'></iframe>";
                // y con esto, colocamos ese código HTML en el <div> que estaba vacío
                videos[i].innerHTML = salida;
        }
}
```

En el ejemplo hemos establecido que el tamaño de esos vídeos es de 480x360 pero, quizás, lo que sería mejor es que se vieran de un tamaño más chico o más grande dependiendo de donde se los incrusta y eso, implica establecer los atributos **width** y **height** de cada caso.

```
<iframe width="480" height="360" src="URL_video" ></iframe>
```

Sin embargo, podemos usar el *script* para que este se adapte adoptando siempre el ancho máximo disponible del ** ** donde lo mostraremos.

Para eso, deberemos leer el ancho del contenedor, tomar los valores de **width** y **height** del vídeo, calcular la proporción (*aspect radio*), establecerá una nueva altura y cambiar los dos datos de la etiqueta, por los valores correspondientes. Para esto cambiaremos un poco los estilo, definiendo un ancho cualquiera y manteniendo oculto el vídeo hasta que hayamos establecido el cambio;

```
<style>
        .mivideo {
                margin: 0 auto;
                text-align: center;
                visibility: hidden;
                width: 300px;
        }
</style>
```

Y en el código anterior, agregaremos el código al final de la condición *if(laURL) {}*, es decir, luego de escribir el **<iframe>** en el ** ** que, recordemos, se mantendrá oculto hasta que lo digamos.

```
var maxw = videos[i].offsetWidth; // es el ancho del <div> contenedor
var elIFRAME = videos[i].firstChild; // este es el <iframe> que acabamos de agregar
var w = elIFRAME.getAttribute("width"); // el ancho del <iframe>
var h = elIFRAME.getAttribute("height"); // el alto del <iframe>
var ar = h/w;  // la proporción entre ambos (aspect radio)
```

```
elIFRAME.width = maxw; // el nuevo ancho del <iframe> es el del <div> contenedor
elIFRAME.height = maxw * ar; // y la altura es proporcional
// como ya está todo listo, lo podemos mostrar
videos[i].style.visibility = "visible";
```

El botón derecho: Peleando contra el viento

¿Es posible proteger el contenido de lo que publicamos para evitar que sea "copiado"?

La respuesta más sincera es NO porque en realidad, si alguien quiere copiar algo lo hará de todos modos. Si hablamos de imágenes, copiarlas es demasiado fácil. En ese sentido, la regla más sencilla y también la que suena más brutal es: "*si no quieres que las imágenes sean accesibles, no las subas a internet*" aunque el servicio te jure que tu privacidad está super-ultra-garantizada.

Sin embargo, hay algunas alternativas que funcionarán pero sólo de manera limitada. Las marcas de agua (*watermarking*) realizadas a mano, usando CSS, JavaScript o con algún *software*; suelen ser poco atractivas y tampoco solucionan nada o quedan muy mal:

Lo que muchos hacen es deshabilitar el botón derecho del ratón de tal manera que no pueda accederse al menú contextual del navegador y muestran un mensaje:

```
function inhabilitar(){
        alert ('Función inhabilitada. Perdonen las molestias.');
        return false;
}
document.oncontextmenu=inhabilitar;
```

Es cierto, se deshabilitará el ratón pero podríamos usar las herramientas de edición del navegador para copiar.

Un poco más útil es inhabilitar el uso del botón derecho del ratón en una imagen concreta. Podemos hacer lo mismo agregando el código en cada imagen:

```
<img oncontextmenu="return false" oncopy="return false" src="URL_imagen">
```

Pero deshabilitar el botón derecho no soluciona los problemas y además, provoca otros porque nosotros tampoco podremos usarlo y los sitios con esa función bloqueada no son agradables ya que el botón derecho lo empleamos para muchas otras cosas que nada tienen que ver con las copias.

"Disclaimer: Copying websites is not cool... but disabling right click isn't either."

¿Y el código fuente? Es lo mismo, no hay mucho que hacer al respecto. Hay *software* que realiza esa supuesta "protección" pero esa protección absoluta del contenido HTML no existe.

Algunos otros trucos son posibles pero ...

Se puede encriptar el código con algún programa pero, como el decodificador debe estar incluido en la misma página para que los navegadores puedan mostrar el sitio, el método parece un poco absurdo aunque para la gran mayoría de los usuarios será complejo de leer.

Eliminar espacios y saltos de línea lo transformará en un código engorroso de entender y editar así que no es muy razonable porque también nos complicará la vida a nosotros.

Insertar líneas en blanco para que el código se muestre *muuuuuuuy* abajo; casi casi ridículo pero hay quien lo usa.

Deshabilitar el cacheo de la imágenes pero eso sólo pude hacerse con determinados servidores.

Entonces, no debemos pensar si vale la pena que nos enojemos con el viento:

"Recuerden que la protección excesiva y las limitaciones que imponemos pueden generar que los visitantes, simplemente, no regresen."

VER REFERENCIAS [Los scripts ofuscados]

Evitar que usen iframes de nuestro sitio

Estamos acostumbrados a ver etiquetas **<iframe>** por todos lados. Es una técnica sencilla y como cualquier otra, tiene sus ventajas y desventajas.

Un **<iframe>** es una ventana a otro sitio, digamos que con ella, creamos un agujero en nuestra página y allí mostramos el contenido de otra página, algo que está en otra parte y sobre lo cual, por supuesto, no tenemos control.

Esta etiqueta tiene algunos atributos que pueden usarse y casi como todas, acepta estilos CSS pero, todo eso, sólo afecta al marco de la ventana en si misma y no a su contenido.

Por supuesto, como todas las demás, no es una etiqueta ni mala ni buena pero, un problema que ha surgido en los últimos tiempos es que hay sitios que incrustan otros y lo hacen con el solo propósito de aprovecharse de su contenido.

Es sencillo; creo un sitio en cualquier servidor gratuito que me permita hacer una página web, lo rodeo de publicidad, le pongo mi nombre y adentro meto otro sitio. Negocio redondo. Para no caer en la paranoia, la solución es simple;

```
if (top.location != self.location) {
        top.location = self.location;
}
```

Lo que hace eso es redireccionar cualquier sitio que coloque un **<iframe>** con nuestra URL y lo envíe de vuelta a casa, es decir a nuestra página.

Cargar una imagen local

Vamos a usar la etiqueta **<input>** de tipo file para seleccionar la imagen y lanzar la función:

```
<input onchange="preview(this.files[0])" type="file">

<div id="imagen"></div>

function preview(file){
	if(!file || !file.type.match(/image.*/)){
		alert("el archivo no es una imagen");
		return;
	}
	if(window.FileReader){
		reader=new FileReader();
		reader.onloadend = function(e){
			// y la mostramos
			var elDIV = document.getElementById("imagen");
			elDIV.innerHTML = '<img src="'+e.target.result+'">';
		};
		reader.readAsDataURL(file);
	} else {
		alert("error desconocido");
	}
}
```

En este caso llamamos *e* al objeto que contendrá varios datos útiles.

e.total es el tamaño de la imagen en bytes
e.timeStamp es la fecha de creación de la imagen expresada en milisegundo
e.type será *"loadend"* cuando la imagen se haya cargado
e.target.result es la imagen en formato data URI: *"..."*

Transiciones CSS y JavaScript

El único evento asociado con las transiciones es **transitionend** que se ejecuta cuando esta se ha completado.

Este evento aún no está estandarizado así que muchas veces se usan prefijos como **webkitTransitionEnd**, **oTransitionEnd** y **MSTransitionEnd**.

Su uso es muy sencillo, suponiendo que tenemos una animación en cierta etiqueta que identificamos con un id, podemos ejecutar una función cuando la transición termine:

```
document.getElementById(id).addEventListener("transitionend", ejemplo);

function ejemplo() {
	alert("transición terminada");
}
```

¿Para que serviría esto? Por ejemplo, podemos tener un un elemento cualquiera con una transición CSS definida que no tendrá efecto alguno a menos que le agreguemos una clase y esto lo haremos al hacer *click* en él:

```
<style>
	#ejemplo {
		background: black;
		height: 200px;
		transition: background 1s;
		width: 200px;
	}
	#ejemplo.efecto {
		background: red;
	}
</style>

<div id="ejemplo" onclick="ejecutar()"></div>

<script>
	var elDIV = document.getElementById("ejemplo");
	elDIV.addEventListener("transitionend", terminar);
	function ejecutar(){
		elDIV.className = "efecto"; // iniciamos la transición
	}
	function terminar() {
		elDIV.className = ""; // terminó la transición
	}
</script>
```

Animaciones CSS y JavaScript

Las animaciones CSS se pueden asociar a tres eventos de JavaScript:

animationstart se ejecuta cuando la animación comienza
animationiteration se ejecuta cuando la animación se repite
animationend se ejecuta cuando la animación se ha completado

En este ejemplo, al hacer *click* en el elemento ** ** comienza la animación y cada vez que se repite, cambia de color:

```
<style>
	#ejemplo {
		background: black;
		height: 100px;
		position: relative;
		width: 100px;
	}
	#ejemplo.efecto {
		animation: 4s linear 0s normal none infinite running mover;
	}
	@keyframes mover {
		from {left: 0;}
		to {left: 500px;}
	}
</style>

<div id="ejemplo" onclick="iniciar()"></div>

<script>
	var elDIV = document.getElementById("ejemplo");
	elDIV.addEventListener("animationstart", iniciar);
	elDIV.addEventListener("animationiteration", repetir);
	elDIV.addEventListener("animationend", terminar);
	function iniciar() {
		elDIV.className = "efecto"; // iniciamos la animación
	}
	function repetir() {
		if(elDIV.style.backgroundColor=="red"){
			elDIV.style.backgroundColor = "black";
		} else {
			elDIV.style.backgroundColor = "red";
		}
	}
</script>
```

Obviamente, como la animación es infinita nunca termina así que el evento **animationend** jamás se ejecuta.

Manejando iframes

Si dos páginas están en el mismo dominio, podemos cargar una dentro de otra usando la etiqueta **<iframe>** y, desde cualquiera de ellas, acceder a los datos de ambas o ejecutar funciones.

Para eso, disponemos de dos propiedades, **contentWindow** y **parent**.

Veamos un ejemplo sencillo, por un lado, tendremos nuestra página principal:

```
<div id="ejemplo">
	<iframe id="" src="iframe.html"></iframe>
</div>
<p onclick="ejecutarIFRAME();">cambiar</p>

<script>
	var elDIV = document.getElementById("ejemplo");
	var elIFRAME = elDIV.children[0];
	function cerrarIFRAME(){
		elDIV.innerHTML = "";
	}
	function cambiar(){
		elIFRAME.contentWindow.cambiarCOLOR();
	}

</script>
```

Y por otro, un archivo llamado *iframe.html* que tendrá algo así:

```
<div id="elemento" style="width:200px;height:200px;">
	<p onclick="cerrar();">cerrrar</p>
</div>

<script>
	function cambiarCOLOR(){
		document.getElementById("elemento").style.backgroundColor = "blue";

	}
	function cerrar(){
		parent.cerrarIFRAME ();
	}
</script>
```

Si abriéramos nuestra página principal, veríamos el contenido de la segunda insertada.

Dentro del *iframe* al hacer *click* en *cerrar* se ejecuta la función *cerrarIFRAME()* que está en la página principal y elimina la etiqueta.

Al revés, en la página principal al hacer *click* en *cambiar* se ejecuta la función *cambiarCOLOR()* que está en la página insertada.

Quiere decir que, desde la página principal, podemos ejecutar cualquier función que esté en el *iframe* utilizando:

```
objeto.contentWindow.nombre_funcion();
```

o acceder a cualquier variable o dato definido:

```
objeto.contentWindow.nombre_variable;
```

Y desde el *iframe*, podemos ejecutar cualquier función que esté en la página principal, utilizando:

```
parent.nombre_funcion();
```

o acceder a cualquier variable o dato definido con:

```
parent. nombre_variable;
```

encode y decode URLs

Las funciones globales **encodeURI()** y **decodeURI()** codifican y decodifican una dirección URL.

Codificarla significa cambiar ciertos caracteres que no deberían ser parte de una dirección, tales como espacios, letras acentuadas, etc.

```
var laURL = "http://misitio.com/pagina web.html?ejemplo=ábcdé";
var dato = encodeURI(laURL);
```

El resultado sería:

http://misitio.com/pagina%20web.html?ejemplo=%C3%A1bcd%C3%A9

```
decodeURI(dato);
```

volvería a darnos la dirección original.

Las funciones **encodeURIComponent()** y **decodeURIComponent()** son similares excepto que también codifican otros caracteres como / ? : @ & = + $ #

```
var laURL = "http://misitio.com/pagina web.html?ejemplo=ábcdé";
var dato = encodeURIComponent(laURL);
```

El resultado sería:

http%3A%2F%2Fmisitio.com%2Fpagina%20web.html%3Fejemplo%3D%C3%A1bcd%C3%A9

```
decodeURIComponent(dato);
```

volvería a darnos la dirección original.

¿Por que usar una u otra? Depende de lo que se quiere hacer; se puede decir que **encodeURI** debería usarse para codificar direcciones URL completas y **encodeURIComponent** para codificar *strings* que pueden ser parte de una dirección aunque también usamos esta función para enviar datos vía Ajax.

Crear una ventana modal propia

Una ventana modal no es algo tan sofisticado como se cree. El concepto básico es sencillo y si no queremos nada excesivamente complejo ni con muchas opciones, crear una propia no es cosa de magia negra sino de entender dos o tres conceptos elementales y probar a ver qué sale.

Cuando hablamos de ventanas modales no debemos confundirnos con las ventanas de tipo *pop-up* que se generan con JavaScript y que no son otra cosa que una instancia del mismo navegador al que abrimos en una ventana nueva con cierta dimensión y en cierta posición usando **window.open()**.

Pero, una ventana modal no es igual, es ... un DIV, es decir, un rectángulo que contiene cosas, una etiqueta HTML como cualquier otra de nuestro sitio.

¿Y que la hace aparentemente distinta? Nada en particular, no hay propiedades especiales, simplemente, se agrega al final de la página y permanece allí, oculta, hasta tanto se la requiere así que vamos y a empezar a crearla colocando el HTML:

```
<div id='MVM'>
	<div id='MVM-inner'>
		<div id='MVM-contenido'> </div>
	</div>
</div>
```

Puesto eso, no pasará absolutamente nada, no veremos nada porque no hay ninguna clase de contenido y, salvo que nosotros hayamos indicado lo contrario, las etiquetas ** ** no poseen propiedades de estilo por defecto así que le pondremos algunas.

```
<style>
	#MVM {
		background-color: #eee;
		left: 0;
		position: fixed;
		top: 0;
		visibility: hidden;
		z-index: 10000;
	}
	#MVM-inner {
		position: relative;
	}
	#MVM-contenido {
		height: 100%;
		width 100%;
	}
</style>
```

Seguimos sin ver nada porque no hay nada que ver y ademas la hemos hecho invisible con **visibility** pero, está allí, arriba a la izquierda.

Como hemos definido que su posición sea absoluta y, como no está contenida dentro de otra etiqueta excepto el **<body>** mismo, las propiedades **left** y **top** cero son el ángulo superior izquierdo de la ventana del navegador.

Entonces, ¿podemos centrarla igual que una ventana *pop-up*? Vemos qué pasa si usamos el ancho y alto del navegador y un poco de aritmética para establecemos los valores:

```
function mostrarMVM(ancho,alto) {
	var MVM = document.getElementById("MVM");
	var MVMCONTENIDO = document.getElementById("MVM-contenido");
	// si está visible, la ocultamos
	if(MVM.style.visibility == "visible") {
		MVM.style.visibility = "hidden";
	}
	// dimensionamo el contenido con los parámetros recibidos
	MVMCONTENIDO.style.width = ancho + "px";
	MVMCONTENIDO.style.height = alto + "px";
	// calculamos el centro
	var x = (window.outerWidth / 2) - (ancho / 2);
	var y = (window.outerHeight / 2) - (alto / 2);
	// posicionamos la ventana modal
	MVM.style.left = x + "px";
	MVM.style.top = y + "px";
	// y la hacemos visible
	MVM.style.visibility = "visible";
}
```

Podríamos probar con:

```
mostrarMVM(300,200);
```

Pero, lo que hace que la ventana modal sea más grande o más pequeña es básicamente su contenido y es sencillo establecer su tamaño si lo conocemos previamente; las complicaciones surgen cuando desconocemos ese tamaño.

Una de las claves de las ventanas modales es lograr que se muestren en el centro mismo de la página, sea cual sea el navegador que se usa y sea cual sea su tamaño o dónde nos encontremos dentro de ella, la segunda de las claves es poder aplicarlas a una etiqueta cualquiera sin tener que escribir mucho código.

Elegimos una etiqueta cualquiera, por ejemplo **** porque no tiene propiedades por defecto, porque es una etiqueta inocua pero podría ser cualquier otra. Sin embargo, hay una forma de conseguir que esa etiqueta que aparentemente nula, se transforme en una etiqueta más "inteligente". Para eso, JavaScript tiene la función muy **addEventListener()** que nos permite agregarle eventos a cualquier tipo de etiqueta y de ese modo. modificarla de manera dinámica, es decir, luego de ser creada.

Agregaremos entonces el evento **onclick** a todas las etiquetas **** que tengan la clase *mimodal* (o el nombre que queramos usar) indicando que cuando se haga *click* en ellas se ejecute la función que llamaremos *demoMODAL()*:

```
window.onload = function() {
	// creamos una lista con todas las etiquetas <span>
	var s = document.getElementsByTagName("span");
	// y las leemos una por una
	for (var i=0; i<s.length; i++) {
		var c = s[i].className; // el contenido del atributo class
		// si es la etiqueta tiene esa clase le agregamos un evento
		if (c=="mimodal") {
			s[i].addEventListener("click", demoMODAL, false);
		}
	}
}
```

Si la funcion *demoMODAL()* fuera esta:

```
function demoMODAL(){
	alert("hola");
}
```

y colocáramos en el HTML algunas de esas etiquetas:

```
<p><span class="mimodal">click acá</span></p>
<p><span>esta no</span></p>
<p><span class="mimodal">click acá</span></p>
```

al hacer *click* en la primera y última, se mostrará una venta de alerta.

Claro que ese ejemplo no es nada interesante así que agregaremos un atributo personal con la dirección URL de una imagen cualquiera:

```
<span class="mimodal" url="URL_imagen">click para ver la imagen</span>
```

Si cambiáramos la función del demo podríamos “leer” ese atributo:

```
function demoMODAL(){
	var laURL = this.getAttribute("url");
	alert(laURL);
}
```

Las ventanas modales perfectas no existen, todas tienen algo que no pueden hacer o limitaciones de alguna clase y no hay manera de abarcar todas las posibilidades ya que son infinitas; es por eso que lo primero a definir luego de tanto código suelto y pedacitos de soluciones, es establecer para qué las necesitamos y cuál es la forma que nos resultará más cómoda de usarlas. El *script* y sus funciones son la forma de obtener eso y no lo inverso.

Como estamos usando esto para mostrar imágenes, lo que debemos saber es su dimensión antes de mostrarlas y para eso, primero la cargaremos con **new Image()** y usaremos los eventos **complete** y **onload** para tenerla en memoria antes de incluirla en la página.

Veamos entonces todas las funciones.

Cuando se termina de cargar la página, buscamos todas las etiquetas **** con la clase *mimodal* y les adosamos un evento onclick:

```
window.onload = function() {
	// creamos una lista con todas las etiquetas <span>
	var s = document.getElementsByTagName("span");
	// y las leemos una por una
	for (var i=0; i<s.length; i++) {
		var c = s[i].className; // el contenido del atributo class
		// si es la etiqueta tiene esa clase le agregamos un evento
		if (c=="mimodal") {
			s[i].addEventListener("click", demoMODAL, false);
		}
	}
}
```

Esta es la función que se ejecutará cada vez que se haga *click* en las etiquetas **** que hemos interceptado.

Allí, leemos la dirección URL que está en el atributo *url* de la etiqueta y ejecutamos la función que "espera" los datos de esa imagen:

```
function demoMODAL(){
	var laURL = this.getAttribute("url"); // nos dirá el contenido del atributo
	waitImagen(laURL,mostrarMVM); // cargamos la imagen
}
```

Esta función genérica cargará la imagen en la memoria, esperará hasta que esté completa y recién entonces enviará los datos y mostrará la modal:

```
function waitImagen(src, callback) {
	var imagen = new Image();
	imagen.src = src;
	if (imagen.complete) {
		callback(imagen); // listo, ya tenemos los datos
		imagen.onload = function() {};
	} else {
		imagen.onload = function() {
			callback(imagen); // listo, ya tenemos los datos
			imagen.onload = function() {};
		}
		imagen.onerror = function() {
			alert("error al cargar la imagen");
		}
	}
}
```

Y finalmente armamos la modal y mostramos la imagen:

```
function mostrarMVM(imagen) {
	// la imagen esta disponible y podemos mostrarla
	var url = imagen.src; // la url de la imagen
	var ancho = imagen.width; // el ancho de la imagen
	var alto =  imagen.height; // el alto de la imagen
	// los objetos de la ventana modal
	var MVM = document.getElementById("MVM");
	var MVMCONTENIDO = document.getElementById("MVM-contenido");
	// si está visible, la ocultamos y borramos su contenido
	if(MVM.style.visibility == "visible") {
		MVM.style.visibility = "hidden";
		MVMCONTENIDO.innerHTML = "";
	}
	// dimensionamos el contenido con el tamaño de la imagen
	MVMCONTENIDO.style.width = ancho + "px";
	MVMCONTENIDO.style.height = alto + "px";
	// agregamos la etiqueta <img> con un evento para poder cerrarla
	MVMCONTENIDO.innerHTML = "<img src="'+url+'" onclick='cerrarMVM();'>";
	// calculamos el centro de la ventana del navegador
	var x = (window.outerWidth / 2) - (ancho / 2);
	var y = (window.outerHeight / 2) - (alto / 2);
	// posicionamos la ventana modal
	MVM.style.left = x + "px";
	MVM.style.top = y + "px";
	// y la hacemos visible
	MVM.style.visibility = "visible";
}
```

Esta función cerrará la ventana modal y eliminará su contenido:

```
function cerrarMVM(v){
	var MVM = document.getElementById("MVM");
	var MVMCONTENIDO = document.getElementById("MVM-contenido");
	MVM.style.visibility = "hidden";
	MVMCONTENIDO.innerHTML = "";
}
```

A partir de eso, cualquier variante es posible si se entiende como funciona ya que sólo hay dos puntos a definir, cuales son los atributos donde agregaremos los datos a usar y cómo escribiremos la salida.

Por ejemplo, hacemos lo mismo pero con un vídeo de YouTube. Definimos que el atributo a usar se llamará *yt* y contendrá el identificador del vídeo.

```
<span class="videomodal" yt="12345678901">un texto o miniatura</span>
```

En ese caso no sería necesario verificar dimensiones porque podríamos utilizar valores fijos y simplemente insertaríamos una etiqueta **<iframe>**. El único detalle a tener en cuenta es que habrá que armar la ventana con algo que permita cerrarla.

Referencias

Operadores especiales

En el caso de incrementos y decrementos como:

```
dato++;
dato--;
```

hay otras simplificaciones posibles que no suelen usarse habitualmente pero vale la pena enumerarlas:

dato1 **-=** dato2 es equivalente a dato1 = dato1 - dato2
dato1 ***=** dato2 es equivalente a dato1 = dato1 * dato2
dato1 **/=** dato2 es equivalente a dato1 = dato1 / dato2
dato1 **%=** dato2 es equivalente a dato1 = dato1 % dato2

dato1 **+=** dato2 es equivalente a dato1 = dato1 + dato2
dato1 **-=** dato2 es equivalente a dato1 = dato1 - dato2
dato1 ***=** dato2 es equivalente a dato1 = dato1 * dato2
dato1 **/=** dato2 es equivalente a dato1 = dato1 / dato2
dato1 **%=** dato2 es equivalente a dato1 = dato1 % dato2

Los operadores de bits se utilizan para números de 32 bits pero debemos despreocuparnos de ese tema ya que JavaScript los convierte internamente y devuelve el número decimal. Estos son los caracteres empleados: AND (**&**), OR (**|**), NOT (**~**), XOR (^), LEFT SHIFT (**<<**), RIGTH SHIFT (**>>**).

El operador **delete** elimina una propiedad de un objeto:

```
var datos = {nombre:"Fulano", id:"1234567", item:22};
delete datos. item ;
```

o bien:

```
delete datos["item"];
```

eliminará la propiedad y el valor del objeto dejando:

```
datos = {nombre:"Fulano", id:"1234567"};
```

El operador **in** devuelve true si cierta propiedad existe dentro de un objeto o array. En el ejemplo anterior:

```
"nombre" in datos; // devuelve true
"item" in datos; // devuelve false porque lo habíamos eliminado
```

El operador **void** se usa para devolver **undefined**.

```
void(0);
```

Un poco más sobre las funciones

Las funciones se definen utilizando la palabra **function** y pueden ser utilizadas de dos modos; lo más usual es utilizar este tipo de sintaxis:

```
function nombre(parametros) {
        // las instrucciones
}
```

y no se ejecutan hasta que son llamadas de algún modo.

También podemos definirlas como expresiones, guardándolas en una variable:

```
var ejemplo = function (parametros) { /* las instrucciones */};
```

de ese modo, la función se ejecutará cuando se utilice esa variable:

```
var demo = ejemplo(parametros);
```

Ese tipo de función se denomina anónima ya que no tiene nombre; el nombre es el de la variable donde se guarda.

Otra forma de crearlas es es utilizar un tipo especial de función interna de JavaScript llamada constructor aunque este método no es algo utilizado habitualmente:

```
var ejemplo = new Function(parametros, instrucciones);
```

Si bien es cierto que las funciones sólo se ejecutan cuando son “llamadas”, hay una sintaxis especial que permite que se ejecuten de manera automática sin ser llamadas y para eso, se escriben rodeándolas con paréntesis:

```
(function () {
        alert("hola");
})();
```

La propiedad **arguments.length** nos permite saber la cantidad de parámetros recibidos en la función y debemos tener presente que JavaScript no verifica los parámetros de las funciones ni permite que se especifique su tipo; si se espera un número y llega una cadena de texto algo funcionará mal.

Si la función ha sido declarada con un número de parámetros pero, se envían menos, los faltantes tendrán un valor de **undefined** por lo tanto, en situaciones específicas, deberíamos verificarlos:

```
function ejemplo(dato1, dato2) {
        if (dato1 === undefined) {
                dato1 = 0;
        }
}
```

Los parámetros también son un objeto y por lo tanto, podemos evaluar su contenido que se encuentra en un *array*:

```
function ejemplo(dato1, dato2) {
        alert(arguments[0]);
        alert(arguments[1]);
}

ejemplo("nombre","apellido");
```

nos mostrará esos dos datos, los textos *"nombre"* y *"apellido"* ya que los parámetros son valores.

Eventos en JavaScript

Todos estos ejecutan alguna clase de función JavaScript y se pueden aplicar a cualquier elemento visible aunque con limitaciones:

Eventos del teclado sobre un elemento:

onkeydown se ejecuta cuando comienza a oprimirse una tecla
onkeypress se ejecuta cuando se oprime una tecla
onkeyup se ejecuta cuando se suelta una tecla oprimida

Eventos del ratón sobre un elemento:

onclick se ejecuta cuando se hace *click* sobre el elemento
oncontextmenu se ejecuta cuando se usa el botón derecho
ondblclick se ejecuta cuando se hace doble *click* sobre el elemento
onmousedown se ejecuta cuando comienza a oprimirse el botón ratón
onmouseenter se ejecuta cuando el puntero del ratón entra al elemento
onmouseleave se ejecuta cuando el puntero del ratón sale del elemento
onmousemove se ejecuta cuando se mueve el ratón
onmouseout se ejecuta cuando el ratón sale del elemento
onmouseover se ejecuta cuando el ratón se mueve dentro del elemento
onmouseup se ejecuta cuando se suelta el botón del ratón
onmousewheel se ejecuta cuando se usa la rueda del ratón
onwheel se ejecuta cuando se mueve el puntero con la rueda del ratón

Eventos para dispositivos de tipo *touchscreen*:

ontouchcancel se ejecuta cuando se interrumpe
ontouchend se ejecuta cuando el dedo abandona el objeto
ontouchmove se ejecuta cuando el dedo se mueve
ontouchstart se ejecuta cuando el dedo toca la pantalla

Eventos de arrastrar y soltar (*drag&drop*):

ondrag se ejecuta cuando el elemento es arrastrado
ondragend se ejecuta cuando el elemento termina de ser arrastrado
ondragenter se ejecuta cuando el elemento arrastrado entra a una zona válida
ondragleave se ejecuta cuando el elemento es soltado en una zona válida
ondragover se ejecuta cuando el elemento es arrastrado a una zona válida
ondragstart se ejecuta cuando el elemento comienza a ser arrastrado
ondrop se ejecuta cuando el elemento es soltado

Eventos copiar y pegar:

oncopy se ejecuta cuando el elemento es copiado
oncut se ejecuta cuando el elemento es cortado
onpaste se ejecuta cuando se pega un contenido

Eventos asociados a la impresora:

onafterprint se ejecuta cuando comienza la impresión
onbeforeprint se ejecuta cuando se abre la ventana de impresión

Eventos asociados a las animaciones y transiciones CSS:

animationend se ejecuta cuando se ha completado la animación
animationiteration se ejecuta cuando la animación se repite
animationstart se ejecuta cuando comienza la animación
transitionend se ejecuta cuando la transición se ha completado

Eventos de los formularios y sus controles:

onblur se ejecuta cuando el elemento pierde el foco
onchange se ejecuta cuando el elemento cambia
onfocus se ejecuta cuando el elemento esta en foco
oninput se ejecuta cuando el elemento es activado
oninvalid se ejecuta cuando el elemento no es válido
onreset se ejecuta cuando se cancelamos
onsearch se ejecuta cuando escribe en un **<input>** de tipo *search*
onselect se ejecuta cuando el elemento es seleccionado
onsubmit se ejecuta cuando es enviado el formulario

Eventos asociados a los objetos multimedia:

onabort se ejecuta cuando se cancela la carga
oncanplay se ejecuta cuando se ha cargado suficiente como para ejecutarse
oncanplaythrough se ejecuta cuando puede reproducirse sin demoras
ondurationchange se ejecuta cuando se modifica la duración
onemptied se ejecuta cuando se desconecta y se produce un error
onended se ejecuta cuando se llega al final de la reproducción
onerror se ejecuta cuando se produce algún error de carga
onloadeddata se ejecuta cuando se ha cargado
onloadedmetadata se ejecuta cuando se han cargado los meta datos
onloadstart se ejecuta cuando comienza la carga
onpause se ejecuta cuando el usuario pausa la reproducción
onplay se ejecuta cuando el usuario reproduce el medio
onplaying se ejecuta cuando se esta reproduciendo
onprogress se ejecuta cuando se está en proceso de carga
onratechange se ejecuta cuando se cambia la velocidad
onseeked se ejecuta cuando termina el cambio de la posición
onseeking se ejecuta cuando se esta cambiando la posición
onstalled se ejecuta cuando no puede accederse a los datos
onsuspend se ejecuta cuando el navegador detiene la carga
ontimeupdate se ejecuta cuando se modifica la posición del tiempo
onvolumechange se ejecuta cuando se cambió el volumen
onwaiting se ejecuta cuando se espera que continúe la carga

Eventos generales:

onabort se ejecuta cuando se cancela la carga de algún recurso
onbeforeunload se ejecuta cuando el documento va a ser cerrado
onerror se ejecuta cuando se produce un error al cargar algún archivo externo
onhashchange se ejecuta cuando el *hash* de la URL es modificado
onload se ejecuta cuando se carga la página
onmessage se ejecuta cuando se recibe un mensaje a través de un evento
onoffline se ejecuta cuando se usa el navegador comienza a usarse sin conexión
ononline se ejecuta cuando se usa el navegador comienza a usarse con conexión
onopen se ejecuta cuando se abre un documento a través de un evento
onpageshow se ejecuta cuando se navega a una página
onpagehide se ejecuta cuando se navega fuera de una página
onpopstate se ejecuta cuando cambia la ventana del historial
onresize se ejecuta cuando se redimensiona el documento
onscroll se ejecuta cuando se mueve la barra de desplazamiento del elemento
onshow se ejecuta cuando a me abre un elemento **<menu>** como menú contextual
onstorage se ejecuta cuando se actualiza un dato guardado localmente
ontoggle se ejecuta cuando se abre o cierra una etiqueta **<details>**
onunload se ejecuta cuando se cierra la página

Los eventos tienen algunas propiedades que, en ciertos casos, pueden ser de utilidad.

Las propiedades **target** y **currentTarget** nos devuelve el objeto que lo ha generado; por ejemplo, si tenemos botones que ejecutan algo:

```
<button id="boton1" onclick="ejemplo(event)">click</button>
<button id="boton2" onclick="ejemplo(event)">click</button>
```

Podemos saber cuál es que se está ejecutando consultando su **id**:

```
function ejemplo(event) {
   alert(event.target.id);
}
```

La propiedad **type** devuelve el tipo de evento; en el caso anterior devolvería *"click"*.

Los eventos también tienen algunos métodos útiles como **preventDefault()** que cancela el evento asociado. Esto puede ser útil en ciertos casos como cuando se crean enlaces con la etiqueta **<a>** y no queremos que se ejecute el atributo **href**:

```
<a id="enlace" onclick="ejemplo(event);" href="http://algunsitio.com/">click</a>

function ejemplo(event){
      event.preventDefault();
      alert("el enlace no funcionará");
}
```

Ampliando ese método existe **stopPropagation()** que evita que el evento tampoco se ejecute en el elemento padre si es que este tuviera alguno.

Más sobre el objeto window

Algunas propiedades del objeto **window**:

window.closed indica se la ventana está cerrada o no
window.content devuelve una referencia al contenido de la ventana actual
window.fullScreen indica si la ventana esta abierta en pantalla completa o no
window.parent devuelve una referencia a la ventana padre
window.self devuelve una referencia a la ventana actual
window.top devuelve una referencia a la ventana principal

Quienes utilizan *localStorage* podrían utilizar estas propiedades: **window.localStorage** y **window.sessionStorage.**

Antiguamente, era común el uso de propiedades que permitían mostrar u ocultar las barras del navegador pero eso es algo que ha pasado de moda porque estos tienen medidas de seguridad que impiden que puedan ser modificadas y de ese modo, evitar que se ejecuten códigos indeseados: **window.locationbar, window.menubar, window.personalbar, window.scrollbars, window.sidebar, window.statusbar** y **window.toolbar.**

Algunas de las propiedades más utilizadas hacen referencia a la posición, dimensión y otras características de la ventana del navegador:

window.screenX es la distancia entre el borde izquierdo de la pantalla y el borde izquierdo del navegador
window.screenY es la distancia entre el borde superior de la pantalla y el borde superior del navegador

Cuando hacemos *scroll* en una página, dos propiedades nos indican la distancia expresada en pixeles: **window.scrollX** y **window.pageXOffset** devuelven la distancia horizontal y **window.scrollY** y **window.pageYOffset** devuelven la distancia vertical.

Además, **window.scrollMaxX** y **window.scrollMaxY** devuelven los valores máximos que podemos desplazarnos dentro de la ventana.

Así como hay propiedades, también hay métodos (funciones internas) que podemos ejecutar. Algunos de ellos son muy comunes y los utilizamos habitualmente aunque no lo sepamos:

window.alert() muestra una ventana de alerta
window.confirm() muestra una ventana donde debemos responder si o no
window.prompt() muestra una ventana donde podemos ingresar algún texto

Estos métodos permiten navegar en el historial:

window.back() vuelve a la página anterior
window.forward() avanza a la página siguiente
window.home() carga la página de inicio

Estos métodos los podemos utilizar para manejar nuevas ventanas o pestañas:

window.stop() detiene la carga de una ventana previamente abierta
window.focus() establece el foco en la nueva ventana
window.blur() quita el foco de la nueva ventana

Al igual que ciertas propiedades en desuso, los navegadores impiden que las ventanas se redimensionen con **window.resizeBy(), window.resizeTo()** y **window.sizeToContent()** o posicionen con **window.moveBy()** y **window.moveTo()**.

Sin embargo, el *scroll* dentro de la ventana si tiene métodos accesibles y estos métodos nos posiciona en algún lugar de la página: **window.scroll(), window.scrollBy(), window.scrollTo(), window.scrollByLines()** y **window.scrollByPages().**

Los *timers* también son métodos del objeto **window**:

window.setTimeout() ejecuta una función cada cierto tiempo
window.clearTimeout() cancela una función llamada con **setTimeout()**
window.setInterval() repite una función cada cierto tiempo
window.clearInterval() cancela una función llamada con **setInterval()**

Métodos varios:

window.getSelection() devuelve el objeto seleccionado
window.postMessage() se utiliza para comunicar dos páginas
window.find() busca un texto
window.print() abre la ventana de impresión

Métodos asociados a los eventos:

window.dispatchEvent() lanza un evento
window.releaseEvents() elimina un evento de cierto tipo
window.removeEventListener() elimina un evento

Los eventos disponibles son similares a los de otros objetos: **onabort, onblur, onchange, onclick, onclose, oncontextmenu, onerror, onfocus, onkeydown, onkeypress, onkeyup, onload, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onreset, onresize, onscroll, onselect, onsubmit, onunload**.

Aunque hay otros más específicos: **onselectionchange, onbeforeunload, onafterprint, onbeforeprint, onhashchange, onmessage, onoffline, ononline, onpagehide, onpageshow, onpaint, onpopstate, onstorage**.

Las ventanas de alerta

Las ventanas de alerta son la forma en que los navegadores no muestran algún tipo de información pero están a nuestra disposición y su uso es muy simple:

Este es un ejemplo:

```
<button onclick="alert('texto en una ventana')">click</button>
```

Para abrir una ventana de alerta no es necesario un botón, puede utilizarse cualquier otra etiqueta:

```
<span onclick="alert('texto en una ventana')">click</span>
```

o en un enlace aunque en ese caso hay que ver cómo evitar que se redireccione utilizando **return false** para impedir que se ejecute **href**:

```
<a href="URL_pagina" onclick="alert('texto en una ventana'); return false">click</a>
```

Por el contrario, **return true** hará que se ejecute. Supongamos que tenemos esto:

```
<a href="URL_pagina" onclick="alert('texto en una ventana'); return true;">click</a>
```

hará que la ventana de alerta se abra y, cuando la cerremos, nos redirigirá a la página indicada por **href**.

Las alertas tiene otro "modelo" donde aparecen dos botones, uno para "aceptar" y otro para "cancelar". La función se llama **confirm()**:

```
<button onclick="confirm('aceptar o denegar')">click</button>
```

Con un poco de JavaScript, las ventanas de alerta pueden tener algo de interacción con los usuarios, por ejemplo, puede mostrarse un mensaje y solicitar una confirmación antes de efectuar el cambio de página. Esta seria la rutina:

```
<script>
	function confirmar(nueva_pagina) {
		respuesta = confirm("cambiar de página")
		if (respuesta!=0) {
			// cualquier respuesta que no sea cero significa SI
			location = nueva_pagina;
		}
	}
</script>

<button onclick="confirmar('URL_pagina');">ejemplo</button>
```

La función **confirm()** no sólo muestra la ventana sino que devuelve un resultado que indica el botón seleccionado (0 es cancelar, 1 aceptar).

Una extensión de las ventanas de alerta, son las ventanas que permiten ingresar algún texto, la función es **prompt()** tiene la siguiente sintaxis:

```
prompt(texto_visible,respuestaXdefecto);
```

Este es un ejemplo:

```
<script>
	function preguntar(){
		nombre =prompt("ingrese el nombre","anónimo");
		alert("saludos "+nombre);
	}
</script>


<button onclick="preguntar()">click</button>
```

Primero, lanzamos la ventana de ingresos con **prompt()** y vamos a guardar la respuesta que nos den en una variable que en este caso llamamos *nombre*.

Al igual que en el caso anterior, la función devuelve un resultado y luego lanzamos la segunda ventana y el texto a mostrar es el resultado de combinar un saludo con el nombre ingresado o el texto por defecto (en este caso la palabra *anónimo*) si el usuario canceló o no ingresó nada.

Otros métodos de los arrays

El método **forEach()** lista los items contenidos en un *array* donde el parámetro a usar debe ser el nombre de una función:

```
var lista = [1, 2, 3, 4, 5];

function demo(item, index) {
   alert("indice = " + index + " -> item = " + item);
}

lista.forEach(demo);
```

El método **every()** verifica si todos los items cumplen una determinada condición que debe ser definida en una función que será el parámetro a usar; devolverá *true* o *false*:

```
var lista = [1, 2, 3, 4, 5];

function verificar(valor) {
   return valor != 0;
}

lista.every(verificar); // devolverá true
```

El método **some()** tiene la misma sintaxis y hace algo parecido pero chequea si cualquiera de los items cumplen la condición.

Con el método **filter()** se verifica una condición y devuelve un nuevo *array* con esos datos:

```
var lista = [1, 2, 3, 4, 5];

function verificar(valor) {
   return valor >= 2 && valor <= 4;
}

var nueva = lista.filter(verificar); // devolverá [2, 3, 4]
```

El método **findIndex()** devuelve el índice del primer elemento que coincide con cierta condición y sigue las mismas reglas de los anteriores.

```
var lista = [1, 2, 3, 4, 5];

function verificar(valor) {
   return valor == 3;
}

lista.findIndex(verificar); // devolverá 2
```

El método **find()** devuelve el valor de ese item.

Así que en el mismo ejemplo:

```
lista.find(verificar); // devolverá 3
```

El método **copyWithin()** copia items dentro del mismo *array*. El primer parámetro indica el índice (empezando en cero) donde se deben copiar; el segundo indica el índice desde donde comenzar a copiar; de modo opcional, el tercer parámetro indica el índice final de los items a a copiar (si no existe se copia hasta el final de *array*).

```
var lista = [1, 2, 3, 4, 5];
var c1 = lista.copyWithin(3,1); // devuelve  [1, 2, 3, 2, 3]
var c2 = lista.copyWithin(3,1,2); // devuelve [1, 2, 3, 2, 5]
```

Debe tenerse en cuenta que cuando se usa este método, los items copiados reemplazan a los items destino.

El método **fill()** llena un *array* (que tengan contenido) con un valor. El único parámetro obligatorio es el valor con el que se quiere llenar pero tiene otros dos opcionales que indican desde donde se comienza y hasta donde se llena:

```
var lista = [1, 2, 3, 4, 5];
var c1 = lista.fill("x"); // devolverá ["x", "x", "x", "x", "x"]
var c2 = lista.fill("x",2); // devolverá [1, 2, "x", "x", "x"]
var c3 = lista.fill("x",2,4); // devolverá [1, 2, "x", "x", 5]
```

El método **map()** crea un nuevo *array* ejecutando una función determinada con todos los items del *array*.

```
var lista = [1, 2, 3, 4, 5];

function multiplicar(item){
        return item * 10;
}

lista.map(multiplicar); // devolverá [10, 20, 30, 40, 50]
```

Las propiedades CSS en JavaScript

Propiedades generales:

bottom → bottom
boxSizing →
clear → clear
clip → clip
content → content
cursor → cursor
display → display
cssFloat → float → float
height → height
left → left
maxHeight → max-height
maxWidth → max-width
minHeight → min-height
minWidth → min-width
overflow → overflow
overflowX → overflow-x
overflowY → overflow-y
position → position
resize → resize
right → right
top → top
width → width
zIndex → z-index

Propiedades de los fondos:

background → background
backgroundAttachment → background-attachment
backgroundBlendMode → background-blend-mode
backgroundColor → background-color
backgroundImage → background-image
backgroundPosition → background-position
backgroundRepeat → background-repeat
backgroundClip → background-clip
backgroundOrigin → background-origin
backgroundSize → background-size

Propiedades de los bordes:

border → border
borderBottom → border-bottom
borderBottomColor → border-bottom-color
borderBottomLeftRadius → border-bottom-left-radius
borderBottomRightRadius → border-bottom-right-radius
borderBottomStyle → border-bottom-style

borderBottomWidth → border-bottom-width
borderColor → border-color
borderImage → border-image
borderImageOutset → border-image-outset
borderImageRepeat → border-image-repeat
borderImageSlice → border-image-slice
borderImageSource → border-image-source
borderImageWidth → border-image-width
borderLeft → border-left
borderLeftColor → border-left-color
borderLeftStyle → border-left-style
borderLeftWidth → border-left-width
borderRadius → border-radius
borderRight → border-right
borderRightColor → border-right-color
borderRightStyle → border-right-style
borderRightWidth → border-right-width
borderStyle → border-style
borderTop → border-top
borderTopColor → border-top-color
borderTopLeftRadius → border-top-left-radius
borderTopRightRadius → border-top-right-radius
borderTopStyle → border-top-style
borderTopWidth → border-top-width
borderWidth → border-width
outline → outline
outlineColor → outline-color
outlineStyle → outline-style
outlineWidth → outline-width

Propiedades de las fuentes:

font → font
fontFamily → font-family
fontSize → font-size
fontSizeAdjust → font-size-adjust
fontStretch → font-stretch
fontStyle → font-style
fontVariant → font-variant
fontWeight → font-weight

Propiedades de los márgenes y *paddings*:

margin → margin
marginBottom → margin-bottom
marginLeft → margin-left
marginRight → margin-right
marginTop → margin-top
padding → padding
paddingBottom → padding-bottom

paddingLeft → padding-left
paddingRight → padding-right
paddingTop → padding-top

Propiedades de los textos:

direction → direction
letterSpacing → letter-spacing
lineBreak → line-break
lineHeight → line-height
quotes → quotes
tabSize → tab-size
textAlign → text-align
textDecoration → text-decoration
textDecorationColor → text-decoration-color
textDecorationLine → text-decoration-line
textDecorationStyle → text-decoration-style
textIndent → text-indent
textJustify → text-justify
textOrientation → text-orientation
textOverflow → text-overflow
textShadow → text-shadow
textTransform → text-transform
unicodeBidi → unicode-bidi
verticalAlign → vertical-align
whiteSpace → white-space
wordBreak → word-break
wordSpacing → word-spacing
wordWrap → word-wrap
writingMode → writing-mode

Propiedades de las tablas:

borderCollapse → border-collapse
borderSpacing → border-spacing
emptyCells → empty-cells
tableLayout → table-layout

Propiedades de las animaciones y transiciones:

animation → animation
animationDelay → animation-delay
animationDirection → animation-direction
animationDuration → animation-duration
animationFillMode → animation-fill-mode
animationIterationCount → animation-iteration-count
animationName → animation-name
animationPlayState → animation-play-state
animationTimingFunction → animation-timing-function
transition → transition

transitionDelay → transition-delay
transitionDuration → transition-duration
transitionProperty → transition-property
transitionTimingFunction → transition-timing-function

Propiedades gráficas, filtros y transformaciones:

backfaceVisibility → backface-visibility
boxDecorationBreak → box-decoration-break
boxShadow → box-shadow
color → color
filter → filter
opacity → opacity
perspective → perspective
perspectiveOrigin → perspective-origin
transform → transform
transformOrigin → transform-origin
transformStyle → transform-style
visibility → visibility

Propiedades de las listas:

counterIncrement → counter-increment
counterReset → counter-reset
listStyle → list-style
listStyleImage → list-style-image
listStylePosition → list-style-position
listStyleType → list-style-type

Propiedades de las columnas

columnCount → column-count
columnFill → column-fill
columnGap → column-gap
columnRule → column-rule
columnRuleColor → column-rule-color
columnRuleStyle → column-rule-style
columnRuleWidth → column-rule-width
columnSpan → column-span
columnWidth → column-width
columns → columns

Propiedad de las cajas flexibles:

alignContent → align-content
alignItems → align-items
alignSelf → align-self
flex → flex
flexBasis → flex-basis
flexDirection → flex-direction

flexFlow → flex-flow
flexGrow → flex-grow
flexShrink → flex-shrink
flexWrap → flex-wrap
justifyContent → justify-content
order → order

Cookies

Una *cookie* es un pequeño texto con información que envía un sitio web y que se guarda en el navegador del usuario para que ese mismo sitio, pueda leerlos en cualquier momento y de ese modo ahorrar tiempo al no tener que solicitarlos otra vez.

Las *cookies* son guardadas con un formato muy sencillo: un nombre y un valor que siempre es un *string*.

Con JavaScript podemos crearlas de este modo:

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

Por defecto, una *cookie* se borra cuando se cierra el navegador y para evitar eso, le agregamos una fecha de expiración:

```
document.cookie = "nombre=valor; expires=Wed, 17 Aug 2016 12:00:00 UTC";
```

Otra forma es establecer la duración expresada en segundos. Por ejemplo, de esta manera, durará un año: *;max-age=60*60*24*365*

Como por defecto, la *cookie* sólo puede ser leída por la misma página que la creó, si queremos que sea accesible para todas las páginas de nuestro sitio, lo indicamos así:

```
document.cookie = "nombre=valor; expires=Wed, 17 Aug 2016 12:00:00 UTC; path=/";
```

También podemos especificar ese *path* para limitarlo a ciertas páginas: *;path='path'*

Otra opción si nuestro sitio tiene subdominios y queremos que ellos también accedan a esos datos es usar: *;domain='misitio.com', '.example.com'*

Un dato extra que podemos agregar sólo queremos que la *cookie* sea leída si la conexión es segura (HTTPS), es incluir: *;secure*

El valor de una *cookie* siempre es un texto que no debe contener comas, puntos y comas ni espacios en blanco.

Leemos su contenido del mismo modo que lo hacemos con cualquier objeto o propiedad:

```
var lista = document.cookie;
```

Aunque así, sólo tendremos una larga cadena de texto con sus nombres y valores separados por un carácter punto y coma.

Para leer el contenido de una *cookie* específica, debemos crear una función propia ya que no hay ningún método interno que lo haga.

Una forma muy simple de hacer esto es usar una expresión regular:

```
function getCookieValue(nombre) {
	var re = new RegExp("(?:(?:^|.*;\s*)"+nombre+"\s*\=\s*([^;]*).*$)|^.*$");
	return document.cookie.replace(re, "$1");
}
alert(getCookieValue("nombre_cookie"));
```

Si queremos cambiar una *cookie* existente, debemos hacer lo mismo que hemos hecho para crearla ya que esto, simplemente la sobrescribirá.

Si queremos borrar una *cookie* existente, debemos cambiarla (aunque n o hace falta colocar el valor) y establecer la fecha de expiración a cualquier fecha pasada:

```
document.cookie = "nombre=; expires=Wed, 17 Aug 2000 12:00:00 UTC; path=/";
```

Si bien existen parámetros que permiten establecer cierta seguridad en las *cookies* que creamos, esta seguridad es muy relativa; desde un **<iframe>** podrían acceder a esos datos y es por eso que lo que se guarda en ellas suelen ser datos incomprensibles que sólo tienen significado para el desarrollador.

Dijimos que para leer el contenido es necesario crear alguna clase de función propia pero, como el manejo de *cookies* está bastante estandarizado, hay funciones clásicas al alcance de cualquiera.

```
/* crear una cookie */
function setCookie(nombre, valor, dias) {
	var d = new Date();
	d.setTime(d.getTime() + (dias*24*60*60*1000));
	var expires = "expires="+ d.toUTCString();
	document.cookie = nombre + "=" + valor + "; " + expires;
}

/* leer una cookie */
function getCookie(nombre) {
	var n = nombre + "=";
	var ca = document.cookie.split(';');
	for(var i=0; i<ca.length; i++) {
		var c = ca[i];
		while (c.charAt(0)==' ') {
			c = c.substring(1);
		}
		if (c.indexOf(name) == 0) {
			return c.substring(n.length,c.length);
		}
	}
	return "";
}
```

VER REFERENCIAS [Manejar el API Storage del navegador]

Detectar el navegador

El objeto **window.navigator** es quien contiene los datos del navegador utilizado por quien visita una página web.

Pero,detectar el navegador no es tan sencillo porque los datos son confusos. Como todo objeto, tiene propiedades

La propiedad **navigator.appCodeName** y **navigator.appName** contienen el nombre del navegador.

Sin embargo, si se usa Opera **navigator.appName** devuelve "Opera"; pero si se usa Internet Explorer, devuelve "Microsoft Internet Explorer" pero sólo en las versiones 10 o inferiores; en todos los demás (Firefox, Chrome, Safari IE11) devuelve "Netscape".

En cambio, **navigator.appCodeName** devuelve "Mozilla" en todos los casos.

La propiedad **navigator.appVersion** contiene la versión pero tampoco es un dato fiable ya que tanto Firefox, Chrome y Safari devolverán "5.0".

La propiedad **navigator.product** contiene el nombre del motor utilizado pero siempre devolverá "Gecko";

Entonces, si nada los identifica ¿cómo detectamos el navegador? Debemos usar la propiedad **navigator.userAgent** que devolverá un dato algo complejo de entender a simple vista:

En IE11 dirá:

"Mozilla/5.0 (Windows NT 6.1; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E; rv:11.0) like Gecko"

En Firefox dirá:

"Mozilla/5.0 (Windows NT 6.1; rv:47.0) Gecko/20100101 Firefox/47.0"

En Chrome dirá:

"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.82 Safari/537.36"

En Edge dirá:

Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.10136

Entonces podríamos usar algo así:

```
var userAgent=navigator.userAgent.toLowerCase();
var browser = {
        chrome : /chrome/.test(userAgent),
        safari : /webkit/.test(userAgent) && !/chrome/.test(userAgent),
        opera : /opera/.test(userAgent),
        edge : /edge/.test(userAgent),
        msie : /like/.test(userAgent) && !/edge/.test(userAgent),
        mozilla : /mozilla/.test(userAgent) && !/(compatible|webkit)/.test(userAgent)
};
```

Y una serie de condicionales nos mostrarían el navegador:

```
if(browser.chrome){
        alert("Chrome");
} else if(browser.msie){
        alert("EI11");
} else if(browser.edge){
        alert("Edge");
} else if(browser.mozilla){
        alert("Firefox");
} else if(browser.opera){
        alert("Opera");
} else {
        alert("desconocido");
}
```

Pero, ese dato va cambiando a gusto de las diferentes empresas así que conviene chequearlos de tanto en tanto.

Otras propiedades:

La propiedad **navigator.cookieEnabled** devuelve *true* si el navegador acepta el uso de *cookies*.

La propiedad **navigator.platform** devuelve el tipo de sistema operativo.

La propiedad **navigator.language** devuelve el idioma.

La propiedad **navigator.plugins** devuelve un *array* con los *plugins* soportados.

La propiedad **navigator.mimeTypes** devuelve un *array* con los tipos MIME soportados.

El objeto sólo tiene un método llamado **navigator.javaEnabled()** que devuelve *true* si el usuario tiene habilitado el uso de Java.

Manejar el API Storage del navegador

Cuando cargamos una página web, hay datos que se mantienen en la memoria y, una vez que la cerramos se pierden. Guardar información localmente (en el dispositivo del usuario) siempre ha sido un problema ya que para que esto fuera posible, solo se contaba con las *cookies* a los que podía accederse desde el mismo dominio que los había creado pero, como sólo se trataba de textos con una capacidad máxima muy pequeña, su uso estaba limitado y para colmo, las *cookies* gozaron siempre de muy mala reputación.

Los navegadores modernos han resuelto esto de manera simple, utilizando un objeto llamado **localStorage** con el que es posible usar JavaScript para guardar y leer datos desde el dispositivo sin generar problemas de seguridad.

El objeto **localStorage** tiene unos pocos métodos que son sencillos. Para guardar algo usamos **setItem()** que requiere dos parámetros, un nombre que lo identifique y el valor a guardar:

```
localStorage.setItem("ejemplo","un dato cualquiera");
```

También podemos guardar los datos con esta sintaxis:

```
localStorage["ejemplo"] ="un dato cualquiera";
```

Una vez guardado, podríamos cerrar el navegador, pagar el dispositivo, volver otro día a nuestra página y leer. Esa es otra de las ventajas de este sistema; las *cookies* deben ser leídas siempre cada vez que se carga la página, en cambio, estos datos son mucho más flexibles porque podemos leerlos cuando lo necesitemos.

Para leerlo, usamos el método **getItem()** con el nombre de la clave (*key*) que le habíamos dado:

```
alert(localStorage.getItem("ejemplo")); // nos mostrara el texto "un dato cualquiera"
```

Y si quisiéramos eliminarlo, usaríamos **removeItem()** con el nombre de la clave.

```
localStorage.removeItem("ejemplo"));
```

Como es un objeto, usando **length** podemos saber cuántas claves hemos guardado y usando **key()** podríamos leerlas una por una utilizando su índice:

```
for (var i=0;i<localStorage.length;i++) {
        alert(localStorage.key(i));
}
```

Si quisiéramos eliminar todas las claves usaríamos:

```
localStorage.clear();
```

Todo esto tiene una limitación, si lo que guardamos son *strings*, no hay problema pero, otro tipo de datos requieren que se los maneje con cierto cuidado.

```
localStorage.setItem("ejemplo", JSON.stringify(datos));
```

Algo bastante común es utilizar *arrays* y para guardarlas necesitamos convertirlas con **JSON.stringify()**:

```
var datos = [1,2,3,4,5];
```

y leerlas con el método **JSON.parse()**:

```
var losDATOS = JSON.parse(localStorage.getItem("ejemplo"));
```

Lo mismo ocurriría si lo que queremos guardar es un objeto:

```
var obj = {nombre:"Fulano",id:"1234567",n:10};
localStorage.setItem( 'otroEJEMPLO', JSON.stringify(obj) );
var elOBJETO = JSON.parse(localStorage.getItem("otroEJEMPLO"));
```

Si vamos a hacer un uso intensivo de **localStorage**, lo mejor es tener alguna función que haga que la tarea de codificar sea más simple.

En este caso, la llamaremos *localCACHE()* y tendrá tres parámetros:

accion indicara qué queremos hacer
key indicara la clave sobre la que queremos operar
data es opcional y serán los datos a guardar cuando sea necesario

La función devolverá false si hubo algún error y true o el dato leído en caso contrario.

```
function localCACHE(accion,key,data){
    var CACHE = window['localStorage'];
    try {
        switch(accion){
            case "save": // guarda los datos en la cache local
                CACHE.setItem(key,data);
                return true; // devuelve true si no hay error
                break;
            case "load": //carga un dato desde la cache local
                // devuelve el dato leído o null si no existe
                return CACHE.getItem(key);
                break;
            case "del": // elimina un dato de la cache local
                CACHE.removeItem(key);
                return true; // devuelve true si no hay error
                break;
            case "tojson": // guarda un dato de tipo array u object
                CACHE.setItem(key,JSON.stringify(data));
                return true; // devuelve true si no hay error
                break;
            case "parse": // carga un dato tipo JSON desde la cache local
```

```
                        var temp = CACHE.getItem(key);
                        /* en caso e problemas podemos usar esta alternativa:
                                var temp = CACHE.getItem(key).split(',');
                        */
                        return JSON.parse(temp); // y devuelve el array
                        break;
                }
        } catch(e){
                return false; // ante cualquier otro error devuelve false
        }
}
```

Ejemplos de uso:

```
// guardamos un dato en la clave llamada "ejemplo"
localCACHE("save", "ejemplo", dato);
// guardamos un array en la clave llamada "ejemplo"
localCACHE("tojson", "ejemplo", dato);

// leemos los datos guardados en la clave llamada "ejemplo"
var datos = localCACHE("load", "ejemplo");

// leemos el array guardada en la clave llamada "ejemplo"
var datos = localCACHE("parse", "ejemplo");

// verificamos si existe una clave:
if (localCACHE("load", "ejemplo")===null){
        // no existe
}
```

Los scripts ofuscados

Cuando un *script* está "ofuscado" no significa que esté enojado y haga cosas raras sólo para molestarnos, significa que está encriptado. De vez en cuando vemos esas cosas en códigos que nos dicen que utilicemos y no hay ningún problema, funcionan bien hasta que uno quiere hacer algún pequeño cambio en cuyo caso, simplemente resulta imposible.

¿Para qué se hace eso? Por dos razones: reducir su tamaño e impedir que se copie; ambas razones son ... digamos que discutibles por no decir absurdas.

Básicamente, minimizar un *script* es quitar espacios innecesarios, tabulaciones, comentarios y saltos de línea además de optimizar variables, funciones, etc, para que la cantidad de caracteres sea menor y de tal modo, se cargue más rápido.

Eso es razonable cuando se trata de librerías o *frameworks* sofisticadas de gran volumen, cosas que uno ni siquiera se atrevería a mirar pero, si son *scripts* "normales", es bastante ridículo ya que sólo hará que su edición sea más difícil y la ganancia es prácticamente nula.

La cantidad de líneas de código que tenga un *script* no tiene relación con su velocidad. Puede poseer cuatro líneas y ser lenta o cien y ser rápida; lo que importa, no es su extensión sino qué procesos ejecuta y cómo los ejecuta.

Ofuscar *scripts* para impedir que se copien o se modifiquen es una tontera. Como cualquier cosa que carga el navegador, una vez que está allí, es editable, copiable y usable; sólo hay que saber cómo. No es un tema de *copyrights* o piraterías, es un tema absolutamente técnico, exento de interpretaciones morales.

Los *scripts* son cargados por el navegador, ya sea que estén en la misma página o en archivos externos; para que funcionen, deben ser accesibles, si son accesibles, pueden ser leídos y por lo tanto, copiados o modificados; ofuscarlos, sólo agrega un paso extra a esa tarea.

Hay muchos compresores/encriptadores, Google tiene el suyo Closure Compiler, Yahoo tiene el YUI compressor, incluso hay algunos *online* como Packer que es de los más utilizados.

Por ejemplo, si quiero que esto se "ofusque":

```
function SINO(cual) {
	var elElemento=document.getElementById(cual);
	if(elElemento.style.display == 'block') {
		elElemento.style.display = 'none';
	} else {
		elElemento.style.display = 'block';
	}
}
```

En Packer, marcaría la opción *Base62 encode* y *Shrink variables* y me daría como resultado este jeroglífico:

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};if(!''.replace(/^/,String))
{while(c--)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];e=function()
{return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)
+'\\b','g'),k[c]);return p}('3 4(a){5 b=6.7(a);8(b.0.1==\'2\')
{b.0.1=\'9\'}c{b.0.1=\'2\'}}',13,13,'style|display|block|function|SINO|var|document|
getElementById|if|none|||else'.split('|'),0,{}))
```

Para quién esté interesado, la función global eval() evalua una expresión o ejecuta el código del argumento.

Es obvio que esto tiene su proceso inverso así que puedo usar esas mismas herramientas para “desofuscar”, no es nada del otro mundo.

Minimizar u ofuscar *scripts* tiene un problema extra, a veces, el resultado obtenido no funcionará porque son herramientas automáticas y es posible que determinadas cosas sean malinterpretadas.

Sólo hay una forma segura de evitar que alguien copie algo: NO PUBLICARLO en la web.

Métodos de los strings

Los métodos **toUpperCase()** y **toLowerCase()** devuelven el texto convertido a mayúsculas o minúsculas:

```
var texto = "texto de ejemplo y demo";
var m1 = text.toUpperCase();  // devolverá "TEXTO DE EJEMPLO Y DEMO"
var m2 = text.toLowerCase();  // devolverá "texto de ejemplo y demo"
```

El método **trim()** elimina todos los espacios del inicio y final del texto:

```
var texto = "    texto de ejemplo y demo   ";
var m3 = text.trim();  // devolverá "texto de ejemplo y demo"
```

Un punto a considerar cuando se usan los métodos de los *strings* es que en ningún caso cambian el texto original sino que siempre devuelven un nuevo texto.

Los métodos **startsWith()** y **endsWith()** verifican si el texto comienza o termina con cierto carácter o caracteres.

```
var texto = "texto de ejemplo y demo";
texto.startsWith()("d"); //devolverá false
texto.startsWith()("t"); //devolverá true
texto.startsWith()("T"); //devolverá false porque buscamos una mayúscula
texto.startsWith()("tambor"); //devolverá false
texto.endsWith()("demo"); //devolverá true
```

De modo opcional se puede agregar un segundo parámetro para indicar desde que posición comenzar a buscar.

Los métodos **indexOf()** y **lastIndexOf()** devuelven la primera o última posición (contando a contar a partir de cero) encontrada del valor dentro de una cadena de texto o -1 si no se encuentra:

```
var texto = "texto de ejemplo y demo";
var i1 = texto.indexOf("de"); //devolverá 6
var i2 = texto.lastIndexOf("de"); //devolverá 19 porque encontró “demo"
```

En ambos casos se puede usar un segundo parámetro indicando la posición a partir de la cual se comenzará a buscar que, por defecto es cero.

El método **search()** es similar pero a diferencia de **indexOf()** donde el valor a buscar es otro texto, en este, ese valor debe ser una expresión regular.

El método **includes()** verifica si cierto texto se encuentra contenido dentro de otro y devuelve true si es así o en caso contrario false.

```
var i3 = texto.includes("ejemplo"); //devolverá true
var i4 = texto.includes("script"); //devolverá false
```

El método **match()** permite buscar un texto dentro de otro y devuelve un array conteniendo cada uno de los resultados encontrados.

El parámetro utilizado debe ser siempre una expresión regular y devuelve null si no se encuentra nada.

```
var texto = "texto de ejemplo y demo";
var arr= texto.match(/de/g);
```

Nos devolverá un array así: *arr["de", "de"]*

VER REFERENCIAS [Expresiones regulares]

Los métodos **slice()** y **substring()** copian el texto que se encuentra entre las posiciones inicial y final indicadas en los parámetros y guardan ese texto en otra variable.

La diferencia entre ambos es que **slice()** admite que el primer parámetro sea un número negativo y en ese caso se comienza a contar desde el final del texto.

El método **substr()** hace lo mismo pero, el segundo parámetro indica la cantidad de caracteres a copiar.

```
var s1 = texto.slice(9,16); // devolverá "ejemplo"
var s2 = texto.substring(9,16); // devolverá "ejemplo"
var s3 = texto.substr(9,7); // devolverá "ejemplo"
```

En todos los casos, si se omite el segundo parámetro, se copia hasta el final.

El método **replace()** busca un texto dentro de otro y lo reemplaza por otro, devolviendo una nueva cadena de texto. Requiere dos parámetros, el texto a buscar y el nuevo texto:

```
var  t = texto.replace("texto","código"); // devolverá "código de ejemplo"
```

Como sólo cambia el primer texto que coincide, si deseamos cambiar varios, deberemos usar expresiones regulares. Por ejemplo, así cambiaríamos todas las letras e minúsculas por mayúsculas:

```
var  p = texto.replace(/e/g,"E"); // devolverá "tExto dE EjEmplo y dEmo"
```

Debe recordarse que este método no modifica el texto original así que si queremos reemplazar y guardarlo en la misma variable deberíamos hacer algo así:

```
texto = texto.replace("texto","código");
```

Los métodos **charAt()** y **charCodeAt()** devuelven el carácter que se encuentra en determinada posición; el primero, devolverá “la letra” y el segundo, el código Unicode.

```
var texto = "texto de ejemplo y demo";
var c1 = texto.charAt(2); //devolverá "x"
var c2 = texto.charCodeAt(2); //devolverá 120
```

El método **fromCharCode()** convierte un código Unicode en un carácter y tiene una sintaxis distinta a los otros métodos, debemos usarlo de este modo:

```
var c3 = String.fromCharCode(120); //devolverá "x"
```

El método **concat()** une dos o mas cadenas de texto:

```
var t1 = "primero ";
var t2 = "segundo ";
var t3 = "último";
var p = t1.concat(t2,t3); //devolverá "primero segundo último"
```

El método **repeat()** devuelve un texto repetido una cierta cantidad de veces:

```
var texto = "hola ";
var r = texto.repeat(3); //devolverá "hola hola hola "
```

El método **split()** crea un *array*, utilizando como separador un carácter determinado. Por ejemplo:

```
var texto = "texto de ejemplo y demo";
var arr = texto.split(" ");
```

Nos devolverá un *array* así: *arr["texto","de","ejemplo","y","demo"]* donde cada palabra es un item.

El carácter separador puede ser cualquiera; por ejemplo, acá usamos comas:

```
var texto = "1,2,3,4,5";
var arr = texto.split(",");
```

Nos devolverá un *array* así: *arr["1","2","3","4","5"]* pero debe tenerse en cuenta que los datos seguirán siendo *strings*.

Un segundo parámetro opcional permite establecer la cantidad máxima de elementos de ese *array*. Por ejemplo si como separador usáramos una cadena vacía, tendríamos un *array* con 23 items así que lo limitaremos a sólo 3 de este modo:

```
var texto = "texto de ejemplo y demo";
var arr = texto.split("",3); //devolverá ["t", "e", "x"]
```

Nos devolverá un *array* así: *arr["t","e","x"]*

Number y Math

La mayoría de los métodos que se pueden aplicar a los datos numéricos permiten convertirlos. El más evidente es **toString()** que convierte un dato numérico en un dato de tipo *string*:

```
var num = 100;
var t = num.toString(); // devuelve el texto "100"
```

El método **toFixed()** también convierte un número en un texto pero admite un parámetro que indica la cantidad de dígitos después del punto decimal.

```
var num = 3.14159265359;
var t = num.toFixed(4); // devuelve el texto "3.1416"
```

El método **toPrecision()** también convierte un número en un texto donde el parámetro define la cantidad de caracteres (sin incluir el punto decimal):

```
var num = 3.14159265359;
var t = num.toFixed(4); // devuelve el texto "3.142"
```

El método **toExponential()** convierte el dato usando notación exponencial donde el parámetro define la cantidad de caracteres detrás del punto decimal:

```
var num = 3.14159265359;
var n = num.toExponential(4); // devuelve 3.1416e+0
```

Todos los métodos devuelven otro y no cambian el dato original.

Algunos métodos permiten verificar el tipo de dato numérico y devuelven *true* o *false*:

isNaN() verifica si el dato es un número
isInteger() verifica si el dato es un número es un entero
isFinite() verifica si el dato es un número finito (no es infinito)

Algunos métodos globales de JavaScript nos permiten convertir una variable cualquiera en un dato numérico

El método **parseInt()** devuelve el dato como número entero o **NaN** si no puede ser convertido:

```
var num = 3.14159265359;
var n = parseInt(num); // devuelve 3
```

El método **parseFloat()** hace lo mismo pero devuelve un número decimal.

El objeto **Math** es el que utilizamos para realizar determinadas operaciones matemáticas y, como todo objeto, posee propiedades y métodos.

Las propiedades en realidad son constantes, no varían:

Math.E devuelve el número de E: 2.718281828459045
Math.PI devuelve el número pi: 3.141592653589793
Math.SQRT2 devuelve la raíz cuadrada de 2: 1.4142135623730951
Math.SQRT1_2 devuelve la raíz cuadrada de 1/2: 0.7071067811865476
Math.LN2 devuelve el logaritmo natural de 2: 0.6931471805599453
Math.LN10 devuelve el logaritmo natural de 10: 2.302585092994046
Math.LOG2E devuelve el logaritmo en base 2 de E: 1.4426950408889634
Math.LOG10E devuelve el logaritmo en base 10 de E: 0.4342944819032518

Los métodos básicos requieren que el parámetro sea un número o una variable numérica:

Math.sqrt(n) devuelve la raíz cuadrada de n
Math.log(n) devuelve el logaritmo natural con base E de n
Math.exp(n) devuelve el valor de E elevado a n
Math.pow(n,x) devuelve el valor de n elevado a la x

Los métodos asociados a funciones trigonométricas utilizan como parámetro un valor expresado en radianes:

Math.sin(n) devuelve el seno de n
Math.cos(n) devuelve el coseno de n
Math.tan(n) devuelve a tangente de n
Math.asin(n) devuelve el arcoseno de n (la inversa del seno)
Math.acos(n) devuelve el arcocoseno de n (la inversa del coseno)
Math.atan(n) devuelven el arcotangente de n (la inversa de la tangente)

El método **abs()** devuelve el valor absoluto de un dato; es decir, devuelve un número positivo:

```
Math.abs(-10.5); // devolverá  10.5
```

Son varios los métodos que redondean un valor y lo hacen de distintas formas:

Math.round() redondea el valor al entero más cercano
Math.ceil() redondea el valor al entero más grande
Math.floor() redondea el valor al entero más chico

Algunos ejemplos:

```
var n1 = 3.3
Math.round(n1); // devuelve 3
Math.round(n1); // devuelve 3
Math.ceil(n1); // devuelve 4
Math.floor(n1); // devuelve 3

var n2 = 3.8
Math.round(n2); // devuelve 4
```

```
Math.round(n2); // devuelve 4
Math.ceil(n2); // devuelve 4
Math.floor(n2) // devuelve 3
```

El método **random()** devuelve un número aleatorio entre 0 (incluido) y 1 (no incluido):

```
Math.random(); // devuelve un número positivo menor que 1
```

Los métodos **min()** y **max()** devuelven el número menor o mayor dentro de una lista:

```
Math.min(1,2,3,4,5); // devuelve 1
Math.max(1,2,3,4,5); // devuelve 5
```

Expresiones regulares

Una expresión regular son un grupo de caracteres que conforman alguna clase patrón de búsqueda.

De alguna manera, hemos usado cosas similares aunque sea de manera sencilla, por ejemplo, cuando hacemos una búsqueda con **.exe* estamos creando un patrón (*pattern*), diciendo que buscamos cualquier archivo de extensión *.exe* sin importar como se llame.

En JavaScript las utilizamos en los métodos **search()**, **replace()** y **match()** para buscar cierto contenido o para reemplazarlo.

Un ejemplo sencillo:

```
var texto = "este es un texto de ejemplo para expresiones regulares";
var t1 = texto.search(/texto/); // devolverá 11
var t2 = texto.search(/TEXTO/); // devolverá -1
```

En el primer caso nos mostrará la posición donde comienza la palabra que buscamos y en el segundo caso devolverá -1 indicando que no se encuentra ya que JavaScript es sensible a mayúsculas y minúsculas.

Lo que hemos creado allí son *patterns* simples colocando el texto a buscar entre barras inclinadas pero también podemos usar modificadores que son letras o caracteres que indican ciertas cosas extras

```
/pattern/modificador
```

Uno de ellos es **i** (*ignoreCase*) que define que la búsqueda no tendrá en cuenta las mayúsculas y minúsculas. De ese modo:

```
var t3 = texto.search(/TEXTO/i); // devolverá 11
var t4 = texto.search(/tExTo/i); // devolverá 11
```

Cuando hacemos la búsqueda, sólo nos devolverá el primer resultado encontrado; si hay varios y queremos tener una lista de todos ellos, usamos **match()** y el modificador **g** (*global*) que podemos combinar con otros:

```
var texto = "este es un texto de ejemplo para expresiones regulares";
var t5 = texto.match(/es/ig); // devolverá 5 resultados
```

Podemos colocar caracteres entre corchetes y de ese modo, buscar cualquiera de esos caracteres. Esto es más fácil de entender con **replace()**; en el ejemplo, reemplazaremos todas las letras *e* por letras *X*:

```
var texto = "este es un texto de ejemplo";
var t1 = texto.replace(/[e]/g,"X"); // devolverá "XstX Xs un tXxto dX XjXmplo"
```

Y ahora reemplazaremos todas los vocales:

```
var t2 = texto.replace(/[aeiou]/g,"X"); // devolverá "XstX Xs Xn tXxtX dX XjXmplX"
```

A la inversa, agregando el carácter ^ reemplazaremos todos los caracteres que no sean la letra *e* y todas las consonantes:

```
var t3 = texto.replace(/[^e]/g,"X"); // devolverá "eXXeXeXXXXXXeXXXXXeXeXeXXXX"
var t4 = texto.replace(/[^aeiou]/g,"X"); // devolverá
"eXXeXeXXuXXXeXXoXXeXeXeXXXo"
```

En determinados casos podemos simplificar el *pattern*; por ejemplo, si buscamos dígitos en alguna clase de secuencia, podemos indicarlo con un guión:

[0-9] buscará cualquier dígito
[3-6] buscará los dígitos 3 4 5 y 6

Lo mismo ocurriría con caracteres:

[A-Z] buscará cualquier letra mayúscula
[a-z] buscará cualquier letra minúscula
[A-z] buscará cualquier letra mayúscula o minúscula

Usando el carácter | indicamos que busque uno u otro:

```
var t5 = texto.replace(/[(a|o)]/g,"X"); // devolverá "este es un textX de ejemplX"
```

Los llamados *metacaracteres* son caracteres especiales que colocamos en el *pattern* y que nos permiten ajustar las búsquedas. Para diferenciarlos de los caracteres literales, se les antepone una barra inclinada invertida.

\s buscará los caracteres espacio
\S buscará los caracteres que no sean espacios

\d buscará los caracteres que sean dígitos
\D buscará los caracteres que no sean dígitos

```
var texto = "ejemplo 1 con 3 palabras y 2 números";
var t1 = texto.replace(/\s/g,"-"); // devolverá "ejemplo-1-con-3-palabras-y-2-números"
var t2 = texto.replace(/\d/g,"X"); // devolverá "ejemplo X con X palabras y X números"
var t3 =  texto.replace(/\D/g,""); // devolverá "132"
```

\b buscará los caracteres al inicio o fin de una palabra
\B buscará los caracteres que no estén al inicio de una palabra

\w buscará los caracteres que sean letras o números
\W buscará los caracteres que no sean ni letras ni números

```
var texto = "palabra, perro, pato, escala, papa";
var t1 = texto.replace(/\bpa/ig,"*"); // devolverá "*labra, perro, *to, escala, *pa"
var t2 = texto.replace(/\Bpa/ig,"*"); // devolverá "palabra, perro, pato, escala, pa*"
```

```
texto.replace(/\w/ig,"*"); // devolverá "*******, *****, ****, ******, ****"
texto.replace(/\W/ig,"*"); // devolverá "palabra**perro**pato**escala**papa"
```

Otros *metacaracteres* buscan caracteres muy específicos:

\0 buscará caracteres nulos
\n buscará el carácter *newline*
\f buscará el carácter *form feed*
\r buscará el carácter *carriage return*
\t buscará el carácter *tab*

Las expresiones regulares (*RegExp*) también tiene algunos métodos predefinidos que podemos usar.

El método **test()** hace una búsqueda y devuelve true encuentra algún resultado o false en caso contrario.

```
var texto = "palabra, perro, pato, escala, papa";
/u/.test(texto); // devolverá false porque no hay ninguna letra u en el texto
```

Es más cómodo colocar el patrón de búsqueda en una variable y luego usarlo así:

```
var pattern = /rr/;
pattern.test(texto); // devolverá true
```

El método **exec()** busca y devuelve un *array* con el texto encontrado o **null**:

```
var pattern = /pa/;
pattern.test(texto); // devolverá  ["pa"]
```

Si bien devuelve un *array*, **exec()** sólo devolverá el primer elemento encontrado así que esto, dará el mismo resultado que lo anterior:

```
var pattern = /pa/g;
pattern.test(texto); // devolverá  ["pa"]
```

Fechas

En JavaScript, una fecha puede ser escrita como un *string*:

"Fri Aug 05 2016 02:43:36 GMT-0300"

o como un número que indica la cantidad de milisegundos transcurridos desde Enero 1, 1970, 00:00:00:

1470375028265

Para crear un objeto **Date** usamos **new Date()** donde los parámetros que podemos incluir son varios.

```
var fecha = new Date(); // si no agregamos parámetros será la fecha actual
```

Pero podemos crear un objeto con una fecha específica usando como parámetros un número (milisegundos) o un *string*:

```
var fecha1 = new Date(1234567);
var fecha2 = new Date("March 10, 2010 20:15:00");
```

Otra forma es agregar siete parámetros con números que indican año, mes, día, hora, minutos, segundos y milisegundos.

```
var fecha3 = new Date(2015,3,10,20,30,00,0); // será abril 15 de 2105 20:30:00
```

Aunque podemos omitir el tiempo (los últimos cuatro parámetros):

```
var fecha4 = new Date(2015,3,10); // será abril 15 de 2105 00:00:00
```

Si el mes es 3 y no 4 es porque se comienza a contar desde cero sí que enero es 0 y diciembre es 11.

El problema de trabajar con fechas es que podemos definirlas con hay diferentes formatos aunque el más utilizado es el formato internacional estándar (ISO 8601)

YYYY-MM-DD -> "2016-03-25"
YYYY-MM -> "2016-03"
YYYY -> "2016"
YYYY-MM-DDTHH:MM:SS -> "2016-03-25T20:30:00"

También se pueden escribir en formato corto:

"MM/DD/YYYY" -> "03-25-2016"
"YYYY/MM/DD" ->"2016/03/25"

o formato largo:

"MM DD YYYY" → "03 25 016"
"MM DD YYYY" → "Mar 25 2016"
"MM DD YYYY" → " March 25 2016"

o formato “full”:

"Wednesday March 9 2016"

Para colmo todo depende de la zona horaria que por defecto, es la especificada en el navegador a menos que se le indique lo contrario.

Una vez que hemos creado un objeto **Date**, podemos leer los distintos valores contenidos. Para eso existen distintos métodos; por ejemplo:

```
var fecha = new Date();
var dia = fecha.getDate();
```

Cada valor puede ser leído de modo individual:

getDate() devuelve un numero con el dia (1 a 31)
getDay() devuelve un número con el día de la semana (0 a 6)
getMonth() devuelve un número con el mes (0 a 11)
getFullYear() devuelve el número del año
getHours() devuelve un número con la hora (0 a 23)
getMinutes() devuelve un número con los minutos (0 a 59)
getSeconds() devuelve un número con los segundos (0 a 59)
getMilliseconds() devuelve un número con los milisegundos (0 a 999)

UTC (*Coordinated Universal Time*) es el estándar de las fechas. Las zonas horarias se expresan como valores positivos o negativos de ese estándar usando el meridiano de *Greenwich* como referencia.

Con el método **toUTCString()** convertimos una fecha local a formato UTC.

Hay métodos que, en lugar de devolvernos los valores locales que son los valores por defecto que dependen de la zona horaria, nos devuelven los valores UTC:

getUTCDate() devuelve un numero con el día (1 a 31)
getUTCDay() devuelve un número con el día de la semana (0 a 6)
getUTCMonth() devuelve un número con el mes (0 a 11)
getUTCFullYear() devuelve el número del año
getUTCHours() devuelve un número con la hora (0 a 23)
getUTCMinutes() devuelve un número con los minutos (0 a 59)
getUTCSeconds() devuelve un número con los segundos (0 a 59)
getUTCMilliseconds() devuelve un número con los milisegundos (0 a 999)

El método **getTimezoneOffset()** devuelve la diferencia entre la fecha UTC y la fecha local, expresada en minutos.

En JavaScript, los diás de la semana comienzan con el domingo y siempre se muestran en inglés: *"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"*.

Los meses también se muestran en inglés: *"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"*.

Así como podemos leer cada dato de una fecha, también podemos modificarla usando un número como parámetro:

setDate() y **setUTCDate()** establecen el dia (1 a 31)
setMonth() y **setUTCMonth()** establecen el mes (0 a 11)
setFullYear() y **setUTCFullYear()** establecen el año
setHours() y **setUTCHours()** establecen la hora (0 a 23)
setMinutes() y **setUTCMinutes()** establecen los minutos (0 a 59)
setSeconds() y **setUTCSeconds()** establecen los segundos (0 a 59)
setMilliseconds() y **setUTCMilliseconds()** establecen los milisegundos (0 a 999)

Los métodos **getTime()** y **setTime()** devuelven o establecen la cantidad de milisegundos transcurridos desde enero 1 de 1970 hasta la fecha indicada y **UTC()** lo hace en formato UTC.

El método **now()** hace lo mismo pero toma como referencia la fecha actual.

Una serie de métodos convierten la fecha a distintos formatos:

toString() la convierte en un formato legible → *"Fri Aug 05 2016 15:43:10 GMT-0300"*
toDateString() convierte la fecha -> *"Fri Aug 05 2016"*
toTimeString() convierte la hora -> *"15:43:18 GMT-0300"*

Como cada dispositivo puede utilizar un formato distinto definido por el usuario, se puede convertir la fecha para que se vean con esos formatos:

toLocaleString() → *"5/8/2016 15:43:02"*
toLocaleDateString() → *"5/8/2016"*
toLocaleTimeString() → *"15:42:53"*

Otros formas de conversión:

toUTCString() la convierte a un formato UTC → *"Fri, 05 Aug 2016 18:43:27 GMT"*
toISOString() la convierte a formato ISO → *"2016-08-05T18:42:17.589Z"*
toJSON() la convierte a un formato JSON → *"2016-08-05T18:42:28.126Z"*

El método **parse()** lo usamos para convertir un fecha expresada como *string* en el número de milisegundos transcurridos desde enero 1 de 1970:

```
var m = Date.parse("Aug 05, 2016");
var fecha = new Date(m);
```

Los caracteres “raros”

Todas las cadenas de texto deben estar escritas entre comillas, ya sean simples o dobles. Entonces, cuando queremos incluir comillas dentro del mismo texto, las cosas se complican.

Para que podamos usar comillas dentro de un texto debemos agregar una barra invertida que se conoce como carácter *escape*.

```
var texto = "esto es una \"cita\" textual";
alert(texto);

var texto = 'esto es una \'cita\' textual';
alert(texto);
```

Claro que de ese modo, no podríamos usar el carácter de la barra inclinada pero, se soluciona usando dos:

```
var texto = "esto \\está entre barras\\ inclinadas";
alert(texto);
```

Esa barra también se usa para incluir otros tipos de caracteres especiales:

\n es una nueva línea
\r es un retorno de carro
\t es una tabulación
\b es backspace
\f es form feed

Cualquier código puede ser escrito en varias líneas:

```
elDIV.innerHTML =
"este es un ejemplo de texto";
```

Pero si queremos “partir” una cadena de texto, debemos adosarle una barra inclinada al final para indicar que continua:

```
var texto = "este es un ejemplo \
de texto";
alert(texto);
```

A veces, cuando se usan *scripts*, debemos escribir otros tipos de caracteres especiales como acentos y estos no se ven bien o aparece un signo de interrogación u otros símbolos raros. Por lo general, esto se resuelve si la codificación de caracteres es correcta pero lo lógico no siempre funciona.

Si tenemos ese problema y queremos usar caracteres de ese tipo lo que debemos hacer es escribirlos con el carácter *escape* seguido de una letra y un número en formato hexadecimal.

En el caso de los acentos o caracteres comunes, sería \x más el código Unicode de dos dígitos:

\xe1 es la letra á
\xe9 es la letra é
\xed es la letra í
\xf3 es la letra ó
\xfa es la letra ú

```
var texto = "\xe1 \xe9 \xed \xf3 \xfa";
alert(texto); // mostraría "áéíóú"
```

En esta página, hay una lista muy completa de todos los caracteres, tanto símbolos como distintos alfabetos.

Propiedades de las etiquetas HTML

Las etiquetas HTML son objetos de JavaScript que tiene y algunas de ellas tienen propiedades específicas que podemos consultar o modificar.

Todas ellas se pueden crear con **createElement()** y acceder con **getElementById()**.

la etiqueta <a>

Se puede acceder a los atributos **download, href, hreflang, rel, search, target** y **type**.

Podemos usar **text** para leer o escribir el texto contenido en el enlace.

Y varias propiedades permiten descomponer la dirección URL del enlace. Usemos una de ejemplo:

http://misitio.com:80/pagina.html#ejemplo

Y vamos de izquierda a derecha:

protocol es el protocolo → *http:*
hostname es el nombre del dominio → *misitio.com*
port es el número del port → *80*
pathname es el nombre de la pagina → */pagina.html*
hash es la parte que sigue al carácter # → *#ejemplo*
host es el nombre del domino más el port → *misitio.com:80*
origin es el protocolo más el domino más el port → *http://misitio.com:80*

Ademas, hay dos propiedades que permiten acceder al nombre de usuario y contraseña en direcciones que utilizan esa capacidad del HTML5; algo que deberíamos evitar: **username** y **password**.

la etiqueta <area>

Se puede acceder a los atributos **alt, coords, href, search, shape** y **target**

Además, también se puede descomponer la dirección URL de manera similar a como se hace con la etiqueta <a>, usando **hash, host, hostname, origin, password, pathname, port, protocol** y **username.**

la etiqueta <embed>

embed http://www.w3schools.com/jsref/dom_obj_embed.asp

la etiqueta <iframe>

Se puede acceder a los atributos **height, name, sandbox, src, srcdoc** y **width**.

Si la página y el contenido del **<iframe>** están en el mismo dominio, hay dos propiedades que permiten acceder a ese contenido

La propiedad **contentDocument** devuelve un objeto que es el documento del ***<iframe>*** y ***contentWindow*** devuelve un objeto que es la ventana del **<iframe>**; es decir los objetos **document** y **window**.

Suponiendo que tuviéramos un **<iframe>** con cierto id:

```
var elIFRAME = document.getElementById("ejemplo"); // la etiqueta <iframe>

// le cambiamos el color de fondo
elIFRAME.contentWindow.document.body.style.backgroundColor = "black";
// o con:
elIFRAME.contentDocument.body.style.backgroundColor = "blue";
```

Si no conocemos el tamaño de la página que se mostrará en el **<iframe>**, podemos leer su dimensión y colocar esos valores en los atributos **width** y **height**:

```
var ancho = elIFRAME.contentWindow.document.body.scrollWidth;
var alto = elIFRAME.contentWindow.document.body.scrollHeight;
elIFRAME.width = ancho;
elIFRAME.height = alto;
```

la etiqueta <img>

Se puede acceder a los atributos **alt, height, src, useMap** y **width**

La propiedad **isMap** nos indica si la imagen es "*clickeable*" por ejemplo, si es parte de algún enlace (*true*) o no (*false*).

La propiedad **complete** devuelve *true* si la imagen se ha terminado de cargar.
Las propiedades **naturalWidth** y **naturalHeight** devuelven el ancho y alto originales de la imagen sin importar si la estamos redimensionando con CSS.

Otras etiquetas

En **<blockquote>** se puede acceder al atributo **cite**.
En **<li>** se puede acceder al atributo **value**.
En **<ol>** se puede acceder a los atributos **start** y **type.**
En **<script>** se puede acceder a los atributos **async**, **charset, defer**, **src** y **type**.
En **<title>** se puede acceder al atributo **text**.

Hay etiquetas que son objetos más complejos y tienen propiedades y métodos exclusivos.

Tablas

En una etiqueta **<table>** podemos referenciar su contenido.

caption devolverá el elemento **<caption>**
rows devolverá un *array* con el contenido de los elementos **<tr>**
tBodies devolverá un *array* con el contenido de los elementos **<tbody>**
tFoot devolverá el elemento **<tfoot>**
tHead devolverá el elemento **<thead>**

Una serie de métodos permiten agregar etiquetas vacías o eliminarlas:

createCaption() agrega una etiqueta **<caption>**
createTFoot() agrega una etiqueta **<tfoot>**
createTHead() agrega una etiqueta **<thead>**

o eliminarlas con: **deleteCaption(), deleteTFoot()** y **deleteTHead()**.

Lo mismo podemos hacer con las filas o las columnas pero, en esos casos, debemos agregar un parámetro que debe ser un número que establece la posición (comenzando a contar desde cero):

insertRow(indice) inserta una etiqueta **<tr>**
deleteRow(indice) elimina una etiqueta **<tr>**

También podemos acceder a la lista de celdas de la etiqueta **<tr>** usando **cells** que devolverá un *array* a las etiquetas **<td>** y **<th>** internas.

La propiedad **rowIndex** devolverá la posición de esa fila dentro de la tabla (su índice).

Los métodos **insertCell() y deleteCell()** insertan o eliminan una etiqueta **<td>.**

Formularios

La etiqueta básica de los formularios es **<form>** y podemos acceder a varios de sus atributos: **acceptCharset, action, autocomplete, encoding, enctype, method, name, noValidate** y **target.**

Tiene dos métodos que equivalen a las acciones de hacer *click* en los botones *reset* y *submit*:

reset() resetea todos los valores del formulario
submit() envía el formulario

Todas las etiquetas que pueden incluirse en un formulario (**<button> <fieldset> <input> <label> <legend> <option> <select> <textarea>**) tienen una propiedad que permite referenciar el formulario donde están contenidas.

Por ejemplo:

```
document.getElementById("boton").form.id // devolverá el id del formulario
```

Y en cada una de ellas puede accederse a alguno de sus atributos.

En **<button>** se puede acceder a **autofocus, disabled, formAction, formEnctype, formmethod, formTarget, formNoValidate, name, type,** y **value**.

En **<fieldset>** se puede acceder a **disabled** y **name**.

En **<label>** se puede acceder a **for** usando **htmlFor**.

En **<select>** se puede acceder a **disabled, multiple, name** y **size.**

En **<option>** se puede acceder a **disabled, label, selected** y **value**.

En **<textarea>** se puede acceder a **autofocus, cols, disabled, maxLength, name, placeholder, readOnly, required, rows** y **wrap**.

Además, algunas de ellas poseen propiedades especiales.

En la etiqueta **<select>** podemos usar **options** para acceder a un *array* con la lista de etiquetas **<option>** que contiene.

La propiedad **selectedIndex** es el número de orden de la opción seleccionada y **value** es el valor de esa opción.

Los métodos **add()** y **remove()** permiten agregar o eliminar etiquetas **<option>** indicando su número índice.

De modo similar, en una etiqueta **<option>** la propiedad **index** nos devuelve su posición y **text** es el texto visible.

La propiedad **defaultSelected** devuelve *true* si la opción seleccionada es la opción por defecto y *false* si no lo es.

En **<textarea>** la propiedad **defaultValue** permite leer o establecer el valor por defecto y **value** el contenido.

El método **select()** selecciona todo el contenido del **<textarea>**.

La etiqueta **<input>** tiene distintos tipos pero en todos ellos podemos acceder a los atributos **autofocus, autocomplete, disabled, name, type** y si los admiten, los atriubutos **placeholder, readOnly** y **required**.

La propiedad **value** siempre nos permite leer o escribir su valor y **defaultValue** nos devuelve el valor por defecto.

Las etiquetas **<input>** de tipo *color, date, datetime, email, month, number, range, search, text, time, url* y *week* poseen una propiedad **list** que devuelve una referencia a los datos contenidos.

En las etiquetas **<input>** de tipo *date, datetime, month, number, range, time,* y *week* podemos acceder a los atributos **max**, **min** y **step** pero, además tienen dos métodos que permiten incrementar o decrementar esos valores: **stepUp()** y **stepDown()**.

En las etiquetas **<input>** de tipo *email, password, search, text* y *url* podemos acceder a los atributos **maxLength, pattern** y **size**.

En las etiquetas **<input>** de tipo *checkbox* y *radio*, la propiedad **checked** lee o establece el estado *checked* y **defaultChecked** devuelve el valor por defecto de ese atributo.

En la etiqueta **<input>** de tipo *password* el método **select()** selecciona todo su contenido.

En la etiqueta **<input>** de tipo *image* los atributos a los que podemos acceder son: **alt, formAction, formEnctype, formMethod, formNoValidate, formTarget, height, src** y **width**.

En la etiqueta **<input>** de tipo *submit* los atributos a los que podemos acceder son: **formAction, formEnctype, formMethod, formNoValidate** y **formTarget**.

En la etiqueta **<input>** de tipo *file*, podemos acceder a un *array* con la lista de archivos seleccionados usando **files.** Además, podemos acceder a los atributos **accept** y **multiple**.

VER REFERENCIAS [Validación de formularios]

VER REFERENCIAS [Propiedades y métodos de audio y video]

Validación de formularios

Validar un formulario significa verificar que al enviarlo, todos los datos existan y estén dentro de los parámetros establecidos.

La validación la hacemos con JavaScript para reducir el acceso al servidor que es quien seguramente procesará esos datos pero, esto no implica que allí no deban volver a verificarse.

En términos generales, podemos validar los datos agregando un atributo **onsubmit** en la etiqueta **<form>** o un atributo **onclick** en un botón. En ambos casos, llamaríamos a una función propia que evaluaría cada uno de los datos y actuaría en consecuencia.

La forma tradicional es usar el atributo **name** para referenciar los objetos:

```
<form name="formulario" action="#" onsubmit="return validar()" method="post">
	<input type="text" name="ingresos">
	<input type="submit" value="enviar">
</form>

document.forms["formulario"]; // es la etiqueta <form>
document.forms["formulario"]["ingresos"]; // es la etiqueta <input>
```

Sin embargo, vamos a colocar atributos **id** como en cualquier otra etiqueta y simplificar la explicación.

```
<form id="formulario" action="#" onsubmit="return validar()" method="post">
	<input type="text" id="ingresos">
	<input type="submit" value="enviar">
</form>
```

Para algo tan simple, ni siquiera haría falta la etiqueta **<form>**, sobre todo porque en estos días, el envío y recepción de datos se suele hacer con Ajax pero esa es otra historia.

La función simplemente verifica que haya un texto y si no lo hay devuelve false y de ese modo, no se enviaría:

```
function validar() {
	var elINPUT = document.getElementById("ingresos");
	var datos = elINPUT.value;
	if (datos == null || datos == "") {
	alert("faltan datos");
		return false;
	}
}
```

Lo mismo podríamos hacer con cualquier otro dato pero el HTML5 nos provee de un atributo que facilita esas verificaciones, dejándolas en mano del navegador.

Para evitar que se envíen datos vacíos, basta que usemos el atributo **required**:

```
<form id="formulario" action="#" method="post">
	<input type="text" id="ingresos" required>
	<input type="submit" value="enviar">
</form>
```

Es el propio navegador el que nos mostrará un mensaje de error indicando que el campo debe tener contenido.

En los diferentes tipos de etiquetas **<input>** hay otros atributos que también indican los datos obligatorios o los valores aceptados tales como **max**, **min** y **pattern**.

Como decíamos, actualmente, el atributo **action** está en desuso y los "formularios" se han simplificado. Muchas veces, ni siquiera necesitamos la etiqueta **<form>** y nos manejamos sólo con **<input>**, **<select>** o **<textarea>** y un botón de envío:

```
<div>
	<input type="text" id="ingresos" required>
	<button onclick="verificar()">enviar</button>
</div>
```

En JavaScript este tipo de validación puede hacerse con **checkValidity()**:

```
function validar() {
	var elINPUT = document.getElementById("ingresos");
	if (elINPUT.checkValidity() == false) {
		alert(elINPUT.validationMessage);
		return;
	}
	alert("enviaremos -> " + elINPUT.value);

}
```

En el ejemplo, al querer "enviar" el dato haciendo *click*, verificamos que la etiqueta **<input>** tenga algún dato porque la hemos definida como obligatoria con required y para eso simplemente evaluamos el valor de su propiedad **checkValidity()** que será *false* si no hay un dato o *true* si lo hay.

Si no escribimos nada, la alerta nos mostrará el texto *"Complete este campo."*

Si usáramos una etiqueta **<input>** de tipo **number**:

```
<input type="number" min="10" max="20" id="ingresos" required>
```

y escribiéramos un valor mayor que 20, la alerta nos mostrará el texto *"Seleccione un valor que no sea más de 20."*

Cada uno de esos textos es el que mostramos con la propiedad **validationMessage** y son definidos por el navegador de modo automático.

La propiedad **willValidate** devuelve *true* si la etiqueta **<input>** debe ser validad y *false* si no debe serlo.

La propiedad **validity** es un dato de tipo objeto que contiene referencias a los distintos errores que pueden evaluarse y cada uno de ellos tendrá un valor de *false* o *true*:

badInput el navegador no puede evaluar el dato
customError es *true* cuando se establece un tipo me error personalizado
patternMismatch el dato no coincide con los límites establecidos por el atributo **pattern**
rangeOverflow el valor es mayor que el indicado en el atributo **max**
rangeUnderflow el valor es menor que el indicado en el atributo **min**
stepMismatch el valor no coincide con el indicado en el atributo **step**
tooLong el dato excede la longitud establecida en el atributo **maxlength**
typeMismatch el tipo de dato no se coincide con el atributo **type**
valid será *true* sólo si el dato es válido
valueMissing falta el dato cuando se usa el atributo **required**

Entonces, en el primer ejemplo: *elINPUT.validity.valueMissing* será *true* y en el segundo ejemplo, *elINPUT.validity.rangeOverflow* será *true*. En ambos casos, todos los demás será *false*.

Eventualmente, con **setCustomValidity()** podemos definir el mensaje que queremos mostrar.

Propiedades y métodos de audio y video

Las etiquetas **<audio>** y **<video>** se pueden crear con JavaScript exactamente igual que cualquier otra.

```
var nuevoAUDIO = document.createElement("audio");
var nuevoVIDEO = document.createElement("video");
```

Y podemos acceder a ellas usando **getElementById()**.

Estas son etiquetas que tienen varias formas de sintaxis; lo mas sencillo es:

```
<audio controls src="archivo.ogg" type="audio/ogg"></audio>
<video controls src="archivo.mp4" type="audio/mp4"></audio>
```

Pero también pueden incluir otras como **<source>** y **<tracks>**:

```
<audio controls>
	<source src="archivo.ogg" type="audio/ogg">
	<source src="archivo.mp3" type="audio/mpeg">
</audio>

<video controls>
	<source src="movie.ogg" type="video/ogg">
	<source src="movie.mp4" type="video/mp4">
	<track src="archivo_subtitulos" kind="subtitles" srclang="es" label="Spanish">
</video>
```

De todos modos, ambas etiquetas comparten una serie de propiedades que son la forma de acceder a los atributos: **autoplay, controls, loop, muted** y **preload**. Además, en el caso de la etiqueta **<video>** tenemos **height, poster** y **width**.

La propiedad **src** devuelve la URL del archivo multimedia si la etiqueta tiene ese atributo; la propiedad **currentSrc** hace lo mismo independientemente de la etiqueta donde esté colocada.

Las propiedades **buffered**, **played** y **seekable** devuelven un objeto que permite conocer o establecer rangos de tiempo (*time ranges*).

Las propiedades **defaultPlaybackRate** y **playbackRate** permiten acceder a la velocidad de reproducción y modificarla. Un valor de 1 es el valor normal, se reduce a la mitad con 0.5 y se duplica con 2. Valores negativos hacen lo mismo pero reproducen de modo inverso.

La propiedad **networkState** devuelve un número que indica el estado en que el servidor está respondiendo: 0 (*NETWORK_EMPTY*) indica que aún no se ha iniciado; 1 (*NETWORK_IDLE*) indica que se ha inicializado pero aún no ha comenzado a descargarse; 2 (*NETWORK_LOADING*) indica que el navegador está descargándolo y 3 (*NETWORK_NO_SOURCE*) indica que el archivo no existe.

Con la propiedad **readyState** devuelve un número que indica el estado en que se encuentra la carga del archivo: 0 (*HAVE_NOTHING*) no hay información; 1 (*HAVE_METADATA*) se han cargado los metadatos del archivo; 2 (*HAVE_CURRENT_DATA*) hay datos cargados pero no no son suficientes para reproducirlos; 3 (*HAVE_FUTURE_DATA*) hay suficientes datos cargados para reproducirlo parcialmente; 4 (*HAVE_ENOUGH_DATA*) hay suficientes datos para reproducirlo.

Si existen etiquetas **<track>** accedemos a ellas con **textTracks** que devuelve un objeto con los distintos parámetros.

La propiedad **duration** devuelve la duración del archivo (en segundos).

La propiedad **currentTime** permite conocer o modificar la posición desde donde se reproduce el archivo (expresada en segundos). La propiedad **seeking** devuelve true si se modifica esa posición.

La propiedad **volume** permite conocer o modificar el valor volumen.

La propiedad **defaultMuted** establece si el archivo debe comenzar sin audio.

La propiedad **ended** devuelve true si la reproducción del archivo ha finalizado.

La propiedad **paused** permite pausar la reproducción.

El método **canPlayType()** verifica si el navegador puede reproducir el tipo de archivo especificado. Devuelve un *string* vacío si no es soportado; caso contrario, devuelve *probably* o *maybe*.

El método **load()** permite recargar el archivo cuando se han hecho cambios en alguna propiedad.

Para manejar la reproducción sólo hay dos métodos; **play()** reproduce y **pause()** lo detiene.

Utilizando algunos eventos asociados podríamos manejar un objeto multimedia de manera personal. Supongamos que nuestra página sólo tiene una etiqueta:

```
<div id="datos">cargando</div>
```

Y agregamos un *script* que cargue un archivo de audio y lo reproduzca sin mostrar el reproductor:

```
var elAUDIO = new Audio("URL_archivo"); // definimos el objeto
```

Usando el evento **onloadeddata** verificamos si se ha cargado y podemos reproducirlo.

```
elAUDIO.onloadeddata = function() {
        elAUDIO.play(); // comienza a reproducirse
        elAUDIO.ontimeupdate = function() {reproducir()};
};
```

Allí usamos el evento **ontimeupdate** que se ejecuta cada vez que cambia la posición del tiempo y vamos mostrando ese tiempo a la vez que chequeamos que no se haya llegado al final.

```
function reproducir(){
        if (elAUDIO.ended){
                alert("terminado")
            } else {
                document.getElementById("datos").innerHTML = elAUDIO.currentTime;
            }
        }
}
```

Librerías: ¿si? ¿no? ¡Socorro!

Las librerías de JavaScript son un gran invento pero, no son otra cosa que herramientas, tan útiles o inútiles como cualquier otra. Ninguna de ellas es una solución universal, uno las puede agregar fácilmente pero debemos estar conscientes de lo que eso significa.

Primero que nada, hay que saber que no son elementos neutros. Una vez que decidimos usar una, la carga del sitio se incrementará, a veces de manera sustancial, ya que son archivos de cierto volumen, muchos de los cuales requieren de otros agregados es decir, de otros archivos que también se deben cargar.

Obviamente, mientras seamos conscientes de las cosas, cualquier decisión es aceptable pero, las librerías tienen sentido sólo si vamos a usarlas de verdad; es decir, si vamos a sacarles provecho porque en eso reside su utilidad. Si sólo las agregamos para usar un efecto eventual, su utilidad se reduce; no es que esté mal pero entendamos que, de alguna manera, es un desperdicio.

Definir la librería a usar también implica una limitación básica; de ahí en más, estaremos “atados” a esa ella; si vamos a agregar alguna otra cosa, deberemos verificar que sea compatible; si bien es cierto que en ciertos casos es posible utilizar varias simultáneamente, esto no es conveniente de ninguna manera ya que sólo estaremos duplicando funciones, incrementando la carga sin beneficio alguno y abriendo la puerta a conflictos que a veces no tienen solución. No vale la pena, lo que puede hacerse con una, seguramente puede hacerse con otra.

Y ese es problema ... si bien, lo que puede hacerse con una también puede hacerse con otra, no siempre es sencillo encontrar la respuesta porque a menos que sepamos cómo funcionan y nosotros mismos creemos nuestros propios *scripts*, encontrar las respuestas en la web puede ser una tarea imposible.

En la web hay una tendencia a canonizar herramientas (navegadores, sistemas operativos, software, redes sociales) y las librerías no están exentas de esa plaga bastante ridícula. "El martillo es lo mejor y quien no lo usa es un tonto" ... cambien martillo por una marca cualquiera y verán de lo que hablo.

El problema de estas dicotomías es que quien pondera el martillo probablemente, sólo conoce el martillo y por lo tanto, cree que es una herramienta universal, única e irreemplazable, cosa que, cualquiera con dos dedos de frente sabe que es falso. Nada es mejor o peor en abstracto. Estas cosas carecen de moral. Simplemente, nos gustan o no nos gustan, nos resultan útiles o no y punto.

Es más ... resultan útiles hoy ... ¿pero mañana? Entonces, ¿si usaba una librería y ahora quiero cambiarla?

Ahhhh buena pregunta ...

Historial

El objeto **window.history** es el que podemos consultar para manejar el historial del navegador pero de modo limitado.

La propiedad **length** nos devuelve la cantidad de páginas guardadas (con un máximo de 50).

El objeto dispone de tres métodos:

history.back() carga la dirección anterior
history.forward() carga la dirección siguiente

history.go() carga la dirección indicada; por ejemplo:

history.go(-1) equivale a **history.back()**
history.go(1) equivale a **history.forward()**