Javascript

Concepto JavaScript

Según mozilla MDN (2020) , JavaScript, o simplemente JS, es un lenguaje de scripting multiplataforma y orientado a objetos. Es un lenguaje pequeño y liviano. Dentro de un ambiente de host, JavaScript puede conectarse a los objetos de su ambiente y proporcionar control programático sobre ellos.

Cómo continúa diciendo (...) JavaScript contiene una librería estándar de objetos, tales como Array, Date, y Math, y un conjunto central de elementos del lenguaje, tales como operadores, estructuras de control, y sentencias. (...) (mozilla MDN, 2020).

¿Por qué se llama Javascript?

Según Crockford (Crockford, 2020), el prefijo de Java sugiere que Javascript está de algún modo relacionado con Java, que es un subconjunto de instrucciones de Java o que es una versión menos potente. (...) "Parece que el nombre fue intencionalmente seleccionado para crear confusión y de esa confusión surge la mala interpretación" (...) (Crockford, 2020)

Javascript es un lenguaje **diferente** a Java.

Si bien javascript tiene una sintaxis similar a Java no significa que tenga algo que ver con Java. Es más, hasta Java tiene una sintaxis similar al lenguaje C y esto no quiere decir que Java sea C. El lenguaje C es otro lenguaje de programación.

Javascript fue creado por Brendan Eich en 1995 para la empresa (y navegador popular es su época) Netscape (...) (Ben Aston, Medium, 2015). Como Java era un lenguaje nuevo en esa era y con mucha aceptación, se optó por utilizar parte del nombre para ganar adeptos. (Brendan Eich, 2016).

Por último y para cerrar este apartado, vamos a decir que Javascript está estandarizado en Ecma International (asociación europea para la creación de estándares para la comunicación y la información) con el fin de ofrecer un lenguaje de programación estandarizado e internacional basado en Javascript y llamado ECMAScript para que todos los navegadores puedan realizar un intérprete que acepte la misma sintaxis de Javascript.

Aplicación

El núcleo de JavaScript puede extenderse para varios propósitos, complementandolo con objetos adicionales (Mozilla MDN), por ejemplo:

- Client-side JavaScript (JS del lado del cliente) extiende el núcleo del lenguaje proporcionando objetos para controlar un navegador y su modelo de objetos (o DOM,

por las iniciales de Document Object Model). Por ejemplo, las extensiones del lado del cliente permiten que una aplicación coloque elementos en un formulario HTML y responda a eventos del usuario, tales como clicks del ratón, ingreso de datos al formulario y navegación de páginas.

- Server-side JavaScript (JS del lado del servidor) extiende el núcleo del lenguaje proporcionando objetos relevantes a la ejecución de JavaScript en un servidor. Por ejemplo, las extensiones del lado del servidor permiten que una aplicación se comunique con una base de datos, proporcionar continuidad de la información de una invocación de la aplicación a otra, o efectuar manipulación de archivos en un servidor.

Sintaxis

A continuación vamos a ver la sintaxis del lenguaje JS (JavaScript) para familiarizarnos con la forma en que se escribe. Seguiremos la documentación de Mozilla sobre el lenguaje (Mozilla MDN, 2020).

Javascript es **case-sensitive**, o sea que distingue entre mayúsculas y minúsculas. Es decir, se puede declarar una variable como *Pajaro* que será distinta a *pajaro*.

Respecto a las instrucciones, en javascript son llamadas **sentencias** y son separadas por punto y coma (;).

Observación: Un punto y coma no es necesario al final si está escrita en una sóla línea. Es más, el intérprete según el estándar ECMAscript tiene reglas para auto insertar los punto y coma al ser ejecutadas las sentencias pero hay casos en donde no queda claro en donde se insertarán esos puntos y comas. Por lo tanto, se recomienda **añadir siempre los punto y coma**. (Mozilla MDN, 2020).

Comentarios

La sintaxis de comentarios es la misma que en otros lenguajes.

```
// comentario en una sola línea

/* este es un comentario

   multilínea

*/

/* no puedes, sin embargo, /* anidar comentarios */ SyntaxError */
```

Declaraciones

Hay tres tipos de declaraciones en Javascript.

var declara una variable, inicializándola opcionalmente a un valor. Si bien esta ha sido exclusivamente la forma de declarar variables durante mucho tiempo, sufre de unos efectos no deseados en el alcance (scope) de la declaración. En otras palabras, se aconseja no utilizarla ya que el nuevo estándar de javascript permite otras formas de declaración que corrige este problema.

El alcance o bloque de ámbito es el lugar en donde es visible esa variable. Por ejemplo, si se declara una variable dentro de una función, esta variable mantiene su valor y es visible sólo dentro de una función.

let declara una variable local en un bloque de ámbito (scope), inicializándola opcionalmente a un valor.

const declara una constante de sólo lectura en un bloque de ámbito.

Variables

Las variables se usan como nombres simbólicos para valores en tu aplicación. Los nombres de las variables, llamados identificadores, se rigen por ciertas reglas.

Un identificador en JavaScript tiene que empezar con una letra, un guión bajo (_) o un símbolo de dólar ($); los valores subsiguientes pueden ser números. Debido a que JavaScript diferencia entre mayúsculas y minúsculas, las letras incluyen tanto desde la "A" hasta la "Z" (mayúsculas) como de la "a" hasta la "z".

Algunos ejemplos de nombre permitidos son Numero_Visitas, temp99, _nombre, $nombre

Ámbito de variable

Cuando declaras una variable fuera de una función, se le denomina variable global, porque está disponible para cualquier otro código en el documento actual. Cuando declaras una variable dentro de una función, se le denomina variable local, porque está disponible sólo dentro de esa función donde fue creada.

Antes de ECMAScript 6 Javascript no tiene ámbito de sentencias de bloque; más bien, una variable declarada dentro de un bloque es local para la función (o ámbito global) en la que reside el bloque. Por ejemplo, el siguiente código registrará 5, porque el ámbito de x es la función (o contexto global) dentro del cual se declara x, no el bloque, que en este caso es la sentencia if.

```
if (true) {

  var x = 5;

}
```

```
console.log(x); // x vale 5
```

Lo anterior es lo que es potencialmente problemático para algunos casos. Por eso en el estándar ECMAScript 2015 se introdujo **let**

```
if (true) {
  let y = 5;
}
console.log(y);  // ReferenceError: y no está definida
```

Constantes

Una constante funciona como una variable pero no cambia su valor, es decir, representa un lugar de almacenamiento de tipos de datos en la memoria pero una vez asignado el valor inicial este no puede ser modificado .

Es de sólo lectura y se define con la palabra clave **const.**

La sintaxis de la definición del identificador es la misma que la de las variables.

```
const PI = 3.14;
```

Una constante no puede cambiar su valor mediante la asignación o volver a declararse mientras se ejecute el script.

Las reglas de ámbito para las constantes son las mismas que las de las variables let en un ámbito de bloque. Si la palabra clave const es omitida, el identificador se asume que representa una variable.

No puedes declarar una constante con el mismo nombre que una función o una variable en el mismo ámbito. Por ejemplo:

```
// ESTO CAUSARÁ UN ERROR
function f() {};
const f = 5;
// ESTO TAMBIÉN CAUSARÁ UN ERROR
function f() {
  const g = 5;
  var g;
```

```
  //sentencias
}
```

Estructura de datos y tipos

El último estándar ECMAScript define ocho tipos de datos:

Siete tipos de datos que son primitivos:

Boolean. true y false.

null. Una palabra clave especial que denota un valor nulo. Como JavaScript es case-sensitive, null no es lo mismo que Null, NULL, o cualquier otra variante.

undefined. Una propiedad de alto nivel cuyo valor no es definido.

Number. Un número entero o un número con coma flotante. Por ejemplo: 42 o 3.14159.

BigInt. Un número entero con precisión arbitraria. Por ejemplo: 9007199254740992n

String. Una secuencia de caracteres que representan un valor "Hola"

Symbol (nuevo en ECMAScript 6). Un tipo de dato cuyas casos son únicos e inmutables

y Object.

Aunque estos tipos de datos son una cantidad relativamente pequeña, permiten realizar funciones útiles con tus aplicaciones. Los otros elementos fundamentales en el lenguaje son los Objects y las funciones. Puedes pensar en objetos como contenedores con nombre para los valores, y las funciones como procedimientos que puede realizar tu aplicación.

Conversión de tipos de datos

JavaScript es un lenguaje de tipo **dinámico**. Esto significa que no tienes que especificar el tipo de dato de una variable cuando la declaras, y los tipos de datos son convertidos automáticamente de acuerdo a lo que se necesite en la ejecución del script. Así, por ejemplo, puedes definir una variable de la siguiente manera:

var respuesta = 42;

Y luego, puedes asignarle una cadena a esa misma variable, por ejemplo:

respuesta = "Gracias por el regalo...";

Debido a que es un lenguaje de tipos dinámicos, esta asignación no causa un mensaje de error.

Convertir string a números

En el caso que un valor representando un número está en memoria como string, hay métodos para la conversión.

parseInt() y parseFloat()

parseInt sólo retornará números enteros, por lo que su uso es disminuido por los decimales.

Literales

Los literales se utilizan para representar valores en JavaScript. Estos son valores fijos, no variables, que literalmente proporciona en su script.

Se usan para inicializar variables más convenientemente. O sea, en otros lenguajes de programación si queremos asignar una lista a una variable y llenarla, primero debemos instanciar el objeto lista y luego agregarle elementos. En Javascript se puede hacer directamente con una sola sentencia.

Literales Array

Un literal array es una lista de cero o más expresiones, cada uno representa un elemento array, entre corchetes (**[ ]**). Cuando se crea un array usando un literal array, se inicializa con los valores especificados como sus elementos, y su longitud se establece por el número de argumentos especificados.

El siguiente ejemplo crea el array cafes con tres elementos y una longitud de tres:

var cafes = ["Tostado Frances", "Colombiano", "Kona"];

Comas adicionales en literales array

No tienes que especificar todos los elementos en un literal array. Si pones dos comas en una fila, se crea el array con undefined para los elementos no especificados. Undefined es un tipo de datos que indica que la variable no ha sido inicializada. El siguiente ejemplo crea el array peces:

var peces = ["Leon", , "Angel"];

Este array tiene dos elementos con valores y un elemento vacío (peces[0] es "Leon", peces[1] es undefined, y peces[2] es "Angel").

Si incluyes una coma al final de la lista de los elementos, la coma es ignorada. En el siguiente ejemplo, la longitud del array es tres. No hay miLista[3]. Todas las demás comas en la lista

indican un nuevo elemento. (Nota: Las comas finales pueden crear errores en las versiones anteriores del navegador y es una buena práctica eliminarlos.)

var miLista = ['casa', , 'escuela',];

En el siguiente ejemplo, la longitud del array es cuatro, y miLista[0] y miLista[2] faltan.

var miLista = [, 'casa', , 'escuela'];

En la siguiente ejemplo, la longitud del array es cuatro, y miLista[1] y miLista[3] faltan. Solo la última coma es ignorada.

var miLista = ['casa', , 'escuela', ,];

Comprender el comportamiento de las comas adicionales es importante para comprender JavaScript como un lenguaje, sin embargo cuando escribimos nuestro propio código: declaramos explícitamente los elementos que faltan como undefined esto aumentará la claridad y el mantenimiento de su código.

Literales Booleanos

Los literales de tipo Booleanos tienen 2 valores posibles: true y false.

NO confundir los valores primitivos Booleanos true y false con los valores true y false del Objeto Booleano. El objeto Booleano es un contenedor alrededor del tipo de dato Primitivo Booleano.

Literales Enteros

Los Enteros pueden ser expresados en decimal (base 10), hexadecimal (base 16), octal (base 8) y binario (base 2).

El Literal entero decimal consiste en una secuencia de dígitos sin un prefijo 0 (cero).

El prefijo 0 (cero) en un literal entero indica que está en octal. Los enteros octales pueden incluir sólo los dígitos 0-7.

El tipo 0x (or 0X) indica hexadecimal. Los enteros Hexadecimal pueden incluir dígitos del (0-9) y letras a-f y A-F.

Algunos ejemplos de enteros Literales son:

0, 117 and -345 (decimal, base 10)

015, 0001 and -077 (octal, base 8)

0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" o base 16)

Literales de Punto Flotante

Un Literal de punto flotante, puede tener las siguientes partes:

Un entero decimal que puede tener signo (precedido por "+" o "-"),

Un Punto decimal ("."),

Una Fracción (otro número decimal),

Un Exponente.

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

Algunos ejemplos de literales de punto Flotante Son:

3.1415

-3.1E12

.1e12

2E-12

.333333333333333333

-.283185307179586

Literales Tipo Objeto

Un literal Objeto es una lista de cero o más parejas de nombres de propiedades asociadas con el respectivo valor, encerradas entre llaves (**{ }**). No debes usar un objetos literal al comienzo de una sentencia. Esto dará lugar a un error o no se comportará como se espera, porque las llaves { serán interpretadas como el inicio de un bloque.

El siguiente es un ejemplo de un objeto literal. El primer elemento del objeto carro define una propiedad, miCarro, y asigna a este el String , "Saturn"; el segundo elemento, a la propiedad getCarro, se le asigna inmediatamente el resultado de invocar a la función (TiposCarro("Honda")); el tercer elemento, la propiedad especial, usa una variable existente (Ventas).

```
var Ventas = "Toyota";

function TiposCarro(nombre) {

  if (nombre== "Honda") {
```

```
  return nombre;
 } else {
  return "Lo siento, nosotros no vendemos " + nombre + ".";
 }
}

var carro = { miCarro: "Saturn", getCarro: TiposCarro("Honda"), especial: Ventas};

console.log(carro.miCarro);  // Saturn

console.log(carro.getCarro); // Honda

console.log(carro.especial); // Toyota
```

Adicionalmente el nombre de una propiedad puede ser un literal entero o puede corresponder a otro objeto, como se muestra a continuación.

```
var auto = { algunosAutos: {a: "Saab", "b": "Jeep"}, 7: "Mazda" };

console.log(auto.algunosAutos.b); // Jeep

console.log(auto[7]); // Mazda
```

Los nombres de las propiedades de un objeto pueden ser cualquier string, incluyendo un string vacío. si el nombre de la propiedad no es un identificador JavaScript válido, este debe ser encerrado en barras cuadradas (corchetes). Los nombres de propiedad que no son identificadores válidos, no pueden ser accedidos con la propiedad punto (.), pero pueden ser accedidos y seteados con la notación de un arreglo ("[]").

```
var propiedadesDeNombreInusual = {
  "": "Un string vacio",
  "!": "Bang!"
}

console.log(propiedadesDeNombreInusual .""");   // SyntaxError: Unexpected string

console.log(propiedadesDeNombreInusual [""]);  // "Un string vacio"

console.log(propiedadesDeNombreInusual .!);    // SyntaxError: Unexpected token !

console.log(propiedadesDeNombreInusual ["!"]); // "Bang!"
```

Ten en cuenta:

```
var foo = {a: "alpha", 2: "two"};

console.log(foo.a);    // alpha

console.log(foo[2]);   // two

//console.log(foo.2);  // Error: missing ) after argument list

//console.log(foo[a]); // Error: a is not defined

console.log(foo["a"]); // alpha

console.log(foo["2"]); // two
```

Literales String

Un literal String corresponde a Cero o más caracteres, encerrados dentro de comillas dobles (") o sencilla (') . Un string debe estar delimitado por comillas del mismo tipo; esto quiere decir que, siempre ambas son dobles o sencillas en cada uno de los casos. Los siguientes son ejemplos de literales String:

```
"foo"

'bar'

"1234"

"Una linea \n otra linea"

"El gato de Jhon"
```

Puedes utilizar cualquiera de los métodos del objeto String en un literal de tipo string—JavaScript automáticamente convierte el literal string en un objeto String de manera temporal, llama al método, y finalmente destruye el objeto temporal de tipo String. También puedes usar la propiedad String.length con un literal string:

```
console.log("El gato de Jhon".length)

// Imprimira el numero de caracteres en el literal, incluyendo espacios en blanco.

// En este caso, 15.
```

Se recomienda que uses un literal string a menos que sea específicamente necesario el uso de un objeto de tipo String.

Uso de caracteres especiales en strings

Adicional a los caracteres normales, también puede incluir caracteres especiales en los strings, como se muestra en el siguiente ejemplo:

"una linea \n otra linea"

La siguiente tabla lista los caracteres especiales que se pueden usar en un string JavaScript.

Tabla de caracteres especiales JavaScript

Character Significado

\b Retroceso

\f Avance de Página

\n Nueva Línea

\r Retorno de Línea

\t Tabulador

\v Tabulador Vertical

\' Apóstrofe o comilla sencilla

\" Comilla doble

\\ Carácter Backslash

\XXX Los caracteres con la codificación Latin-1 especificada por tres dígitos octales XXX entre 0 y 377. Por ejemplo, \251 es la secuencia octal para el símbolo copyright.

\xXX Los caracteres con la codificación Latin-1 especificada por dos dígitos hexadecimales XX entre 00 y FF. Por ejemplo, \xA9 es la secuencia hexadecimal para el símbolo copyright.

\uXXXX Los caracteres Unicode especificados por la secuencia de cuatro dígitos Hexadecimales XXXX. Por ejemplo, \u00A9 es la secuencia Unicode para el símbolo copyright .

Caracteres de Escape

Para caracteres no listados en la tabla anterior, la precedencia del backslash o barra invertida (“\”) es ignorada, pero su uso está obsoleto y debe ser evitado.

Puedes insertar comillas dobles dentro de un string anteponiendo un carácter backslash. significa esto como un escape de las comillas. Por ejemplo:

var quote = "El lee \"La cremación de Sam McGee\" de R.W. Service.";

console.log(quote);

El resultado de esto sería:

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

Para incluir un literal backslash en un string, debes usar el caracter de escape backslash. Por ejemplo, para asignar la ruta c:\temp a un string, use lo siguiente:

var home = "c:\\temp";

También puedes insertar saltos de línea. El backslash y el salto de línea son removidos del valor del string.

var str = "this string \

is broken \

across multiple\

lines."

console.log(str); // this string is broken across multiplelines.

Aunque JavaScript no tiene sintaxis "heredoc" (string de múltiples líneas con saltos de línea con una sintaxis especial) puede acercarse insertando un backslash y un salto de línea al final de cada línea:

var poem =

"Roses are red,\n\

Violets are blue.\n\

I'm schizophrenic,\n\

And so am I."

Expresiones y operadores

Este capítulo describe expresiones y operadores de JavaScript, incluyendo los de asignación, comparación, aritméticos, lógicos, cadena, ternarios y otros.

Operadores

JavaScript tiene operadores binarios y unarios, y un operador ternario especial, el operador condicional. Un operador binario requiere dos operandos, uno antes del operador y otro después de este.

operando1 operador operando2

Por ejemplo, 3+4 o x*y.

Un operador unario requiere un único operando, ya sea antes o después del operador:

operador operando

o

operando operador

Por ejemplo, x++ o ++x

Operadores de asignación

Un operador de asignación asigna un valor al operando de la izquierda en función del valor del operando de la derecha. El operador básico de asignación es el de igual (=), que asigna el valor del operando de la derecha al operando de la izquierda. Por ejemplo, x = y, está asignando el valor de y a x.

También existen operadores compuestos de asignación que son la forma abreviada de las operaciones de la siguiente tabla:

Tabla Operadores de asignación

Nombre	Operador abreviado	Significado
Operadores de asignación	x = y	x = y
Asignación de adición	x += y	x = x + y
Asignación de sustracción	x -= y	x = x - y
Asignación de multiplicación	x *= y	x = x * y
Asignación de división	x /= y	x = x / y
Asignación de resto	x %= y	x = x % y
Asignación de exponenciación	x **= y	x = x ** y
Asignación de desplazamiento a la izquierda	x <<= y	x = x << y

Asignación de desplazamiento a la derecha x >>= yx = x >> y

Asignación de desplazamiento a la derecha sin signo x >>>= y x = x >>> y

Asignación AND binaria x &= y x = x & y

Asignación XOR binaria x ^= y x = x ^ y

Asignación OR binaria x |= y x = x | y

Destructuración

Para asignaciones más complejas, la sintaxis de asignación con destructuración es una expresión de Javascript que permite extraer datos de arreglos u objetos usando una sintaxis que se asemeja a la construcción de arreglos o objetos literales.

```
var foo = ['uno','dos','tres'];

// sin destructuración

var uno  = foo[0];

var dos  = foo[1];

var tres = foo[2];

// con destructuración

var [uno, dos, tres] = foo;
```

Operadores de comparación

Un operador de comparación compara sus operandos y devuelve un valor lógico en función de si la comparación es verdadera (true) o false (false). Los operadores pueden ser numéricos, de cadena de caracteres (Strings), lógicos o de objetos.

Las cadenas de caracteres son comparadas basándose en un orden lexicográfico estándar, usando valores Unicode (es una codificación que determina los caracteres que se pueden mostrar en la pantalla, incluyendo chino, etc).

En la mayoría de los casos, si los dos operandos no son del mismo tipo, JavaScript intenta convertirlos en el tipo apropiado para permitir la comparación, generalmente esta conversión se realiza de manera numérica. Las únicas excepciones que tiene esta conversión son los operadores === y !== que ejecutan comparaciones de igualdad o desigualdad de manera estricta (chequeando si ambos operandos son del mismo tipo). Estos operadores no intentan convertir los operandos a un tipo compatible antes de comprobar su igualdad.

La siguiente tabla describe los operadores de comparación en base al siguiente código de ejemplo:

var var1 = 3;

var var2 = 4;

Tabla Operadores de comparación

Operador Descripción Ejemplos devolviendo true

Igualdad (==) Devuelve true si ambos operadorandos son iguales. 3 == var1

"3" == var1

3 == "3"

Desigualdad (!=) Devuelve true si ambos operandos no son iguales. var1 != 4

var2 != "3"

Estrictamente iguales (===)

Devuelve true si los operandos son iguales y tienen el mismo tipo. Mira también Object.is y sameness in JS.

3 === var1

Estrictamente desiguales (!==)

Devuelve true si los operandos no son iguales y/o no son del mismo tipo.

var1 !== "3"

3 !== "3"

Mayor que (>)

Devuelve true si el operando de la izquierda es mayor que el operando de la derecha.

var2 > var1

"12" > 2

Mayor o igual que (>=) Devuelve true si el operando de la izquierda es mayor o igual que el operando de la derecha. var2 >= var1

var1 >= 3

Menor que (<) Devuelve true si el operando de la izquierda es menor que el operando de la derecha. var1 < var2

"2" < 12

Menor o igual que (<=) Devuelve true si el operando de la izquierda es menor o igual que el operando de la derecha. var1 <= var2

var2 <= 5

Nota: (=>) no es un operador, sino una notación para las Funciones Flecha.

Operadores aritméticos

Los operadores aritméticos toman los valores numéricos (tanto literales como variables) de sus operandos y devuelven un único resultado numérico. Los operadores aritméticos estándar son la suma (+), la resta (-), la multiplicación (*) y la división (/). Estos operadores funcionan como en la mayoría de los lenguajes de programación cuando son usados con números de coma flotante (en particular, tenga en cuenta que la división por cero produce Infinity). Por ejemplo:

1 / 2; // 0.5

1 / 2 == 1.0 / 2.0; // es true

Además de las operaciones de aritmética estándar (+, -, * y /), JavaScript brinda los siguientes operadores aritméticos descritos en la tabla:

Tabla Operadores aritméticos

Operador Descripción Ejemplo

Resto (%)

Operador binario correspondiente al módulo de una operación. Devuelve el resto de la división de dos operandos.

12 % 5 devuelve 2.

Incremento (++)

Operador unario. Incrementa en una unidad al operando. Si es usado antes del operando (++x) devuelve el valor del operando después de añadirle 1 y si se usa después del operando (x++) devuelve el valor de este antes de añadirle 1.

Si x es 3, entonces ++x establece x a 4 y devuelve 4, mientras que x++ devuelve 3 y, solo después de devolver el valor, establece x a 4.

Decremento (--)

Operador unario. Resta una unidad al operando. Dependiendo de la posición con respecto al operando tiene el mismo comportamiento que el operador de incremento.

Si x es 3, entonces --x establece x a 2 y devuelve 2, mientras que x-- devuelve 3 y, solo después de devolver el valor, establece x a 2.

Negación Unaria (-)

Operación unaria. Intenta convertir a número al operando y devuelve su forma negativa.

-"3" devuelve -3.

-true devuelve -1.

Unario positivo (+) Operación unaria. Intenta convertir a número al operando. +"3" devuelve 3.

+true devuelve 1.

Exponenciación (**) baseexponente Calcula la potencia de la base al valor del exponente. Es equivalente a 2 ** 3 devuelve 8.

10 ** -1 devuelve 0.1.

Operadores lógicos

Los operadores lógicos son comúnmente utilizados con valores booleanos; estos operadores devuelven un valor booleano. Sin embargo, los operadores && y || realmente devuelven el valor de uno de los operandos, así que si estos operadores son usados con valores no booleanos, podrían devolverán un valor no booleano. En la siguiente tabla se describen los operadores lógicos:

Tabla Operadores lógicos

Operador	Uso	Descripción
AND Lógico (&&)	expr1 && expr2	

Devuelve expr1 si puede ser convertido a false de lo contrario devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, && devuelve true si ambos operandos son true, en caso contrario devuelve false.

OR Lógico (||) expr1 || expr2

Devuelve expr1 si puede ser convertido a true de lo contrario devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, || devuelve true si alguno de los operandos es true, o false si ambos son false.

NOT Lógico (!) !expr

Devuelve false si su operando puede ser convertido a true, en caso contrario, devuelve true.

Ejemplos de expresiones que pueden ser convertidas a false son aquellas que pueden ser evaluadas como null, 0, NaN, undefined o una cadena vacía.

El siguiente código muestra ejemplos del operador && (AND Lógico).

```
var a1 = true  && true;     // t && t devuelve true

var a2 = true  && false;    // t && f devuelve false

var a3 = false && true;     // f && t devuelve false

var a4 = false && (3 == 4); // f && f devuelve false

var a5 = "Cat" && "Dog";    // t && t devuelve "Dog"

var a6 = false && "Cat";    // f && t devuelve false

var a7 = "Cat" && false;    // t && f devuelve false
```

El siguiente código muestra ejemplos del operador || (OR Lógico).

```
var o1 = true  || true;     // t || t devuelve true

var o2 = false || true;     // f || t devuelve true

var o3 = true  || false;    // t || f devuelve true

var o4 = false || (3 == 4); // f || f devuelve false

var o5 = "Cat" || "Dog";    // t || t devuelve "Cat"

var o6 = false || "Cat";    // f || t devuelve "Cat"

var o7 = "Cat" || false;    // t || f devuelve "Cat"
```

El siguiente código muestra ejemplos del operador ! (NOT Lógico).

```
var n1 = !true;  // !t devuelve false

var n2 = !false; // !f devuelve true

var n3 = !"Cat"; // !t devuelve false
```

Evaluación mínima o evaluación de circuito corto

Como las expresiones lógicas son evaluadas de izquierda a derecha, estas son evaluadas de manera mínima (también llamada de circuito corto) usando las siguientes reglas:

false && algo es mínimamente evaluada a false.

true || algo es mínimamente evaluada a true.

Las reglas de la lógica garantizan que las anteriores evaluaciones son siempre correctas. Nota que el operando algo no es evaluado, por lo que situarlo no surte ningún efecto.

Operadores de cadenas de caracteres

Además de los operadores de comparación, que pueden ser usados en cadenas de caracteres, el operador de concatenación (+) une dos valores de tipo String, devolviendo otro String correspondiente a la unión de los dos operandos.

Por ejemplo,

```
console.log("mi " + "string"); // lanza el String "mi string" en la consola.
```

La versión acortada de este operador de asignación (+=) puede ser usada también para concatenar cadenas de caracteres.

Por ejemplo,

```
var mistring = "alfa";

mistring += "beto"; // devuelve "alfabeto" y asigna este valor a "mistring".
```

Operador condicional (ternario)

El operador condicional es el único operador de JavaScript que necesita tres operandos. El operador asigna uno de dos valores basado en una condición. La sintaxis de este operador es:

condición ? valor1 : valor2

Si la condición es true, el operador tomará el valor1, de lo contrario tomará el valor2. Puedes usar el operador condicional en cualquier lugar que use un operador estándar.

Por ejemplo,

var estado = (edad >= 18) ? "adulto" : "menor";

Esta sentencia asigna el valor adulto a la variable estado si edad es mayor o igual a 18, de lo contrario le asigna el valor menor.

Operador coma

El operador coma (,) simplemente evalúa ambos operandos y retorna el valor del último. Este operador es ante todo utilizado dentro de un ciclo for, permitiendo que diferentes variables sean actualizadas en cada iteración del ciclo.

Por ejemplo, si a es un Array bi-dimensional con 10 elementos en cada lado, el siguiente código usa el operador coma para actualizar dos variables al mismo tiempo. El código imprime en la consola los valores correspondientes a la diagonal del Array:

```
for (var i = 0, j = 9; i <= j; i++, j--)

  console.log("a[" + i + "][" + j + "]= " + a[i][j]);
```

Operadores unarios

Una operación unaria es una operación que sólo necesita un operando.

delete

La función del operador delete es eliminar un objeto, una propiedad de un objeto, o un elemento en el indice específico de un Array. La sintaxis es la siguiente:

```
delete nombreObjeto;

delete nombreObjeto.propiedad;

delete nombreObjeto[indice];

delete propiedad; // solo admitido con una declaración "with"
```

Donde nombreObjeto es el nombre de un objeto, propiedad el nombre de la propiedad de un objeto, e índice un entero que representa la localización de un elemento en un Array.

La cuarta forma es admitida sólo dentro de una sentencia with, para eliminar una propiedad de un objeto.

Puedes usar el operador delete para eliminar aquellas variables que han sido declaradas implícitamente, pero no aquellas que han sido declaradas con var.

Si la operación delete finaliza con éxito, establece la propiedad o el elemento a undefined. El operador delete devuelve true si la operación ha sido posible y false en caso contrario.

```
x = 42;

var y = 43;

miObj = new Number();

miObj.h = 4;    // crea la propiedad "h"

delete x;       // devuelve true (se puede eliminar si se declaró implícitamente)

delete y;       // devuelve false (no se puede eliminar si se declaró con var)

delete Math.PI; // devuelve false (no se pueden eliminar propiedades predefinidas)

delete miObj.h; // devuelve true (se pueden eliminar propiedades definidas por el usuario)

delete miObj;   // devuelve true (se puede eliminar si se ha declarado implícitamente)
```

Eliminando elementos de un Array

Cuando se elimina un elemento de un Array, su tamaño no se verá afectado. Por ejemplo, si elimina a[3], a[4] seguirá existiendo pero el valor de a[3] será undefined ya que dejará de existir en el Array.

En el siguiente ejemplo, arboles[3] es eliminado con delete y como se puede constatar el mismo dejará de existir en el Array.

```
var arboles = new Array("secoya", "laurel", "cedro", "roble", "arce");

delete arboles[3];

if (3 in arboles) {

  // Esto no se ejecutará

}
```

Si desea que un elemento de un Array exista pero su valor sea undefined, debe asignarle el valor undefined en vez de usar el operador delete. En el siguiente ejemplo a arboles[3] se le asigna el valor undefined, pero el elemento seguirá existiendo

```
var arboles = new Array("secoya", "laurel", "cedro", "roble", "arce");

arboles[3] = undefined;

if (3 in arboles) {
```

```
  // Este trozo si se ejecutará
}
```

Para eliminar un lugar en un Array hay que usar slice().

typeof

El operador typeof es usado de las siguientes maneras:

typeof operando

typeof (operando)

El operador typeof devuelve una cadena de caracteres indicando el tipo del operando evaluado. En los ejemplos anteriores operando hace referencia a la cadena de caracteres, variable, palabra clave u objeto del que se intenta obtener su tipo. Los paréntesis son opcionales.

Supón que defines las siguientes variables:

```
var miFuncion = new Function("5 + 2");
var forma = "redonda";
var largo = 1;
var hoy = new Date();
```

El operador typeof devolverá los siguientes resultados en estas variables:

```
typeof miFuncion; // devuelve "function"
typeof forma;     // devuelve "string"
typeof largo;     // devuelve "number"
typeof hoy;       // devuelve "object"
typeof noExiste;  // devuelve "undefined"
```

Con las palabras clave true y null, el operador typeof devuelve los siguientes resultados:

```
typeof true; // devuelve "boolean"
typeof null; // devuelve "object"
```

Con los números o las cadenas de caracteres, el operador typeof devuelve los siguientes resultados:

```
typeof 62;          // devuelve "number"

typeof 'Hello world'; // devuelve "string"
```

En el caso de que se utilice como operando una propiedad, el operador typeof devolverá el tipo de dicha propiedad:

```
typeof document.lastModified; // devuelve "string"

typeof window.length;       // devuelve "number"

typeof Math.LN2;            // devuelve "number"
```

Con métodos y funciones, el operador typeof devolverá los siguientes resultados:

```
typeof blur;       // devuelve "function"

typeof eval;       // devuelve "function"

typeof parseInt;   // devuelve "function"

typeof shape.split; // devuelve "function"
```

Para objetos predefinidos, el objeto typeof devuelve los siguientes valores:

```
typeof Date;     // devuelve "function"

typeof Function; // devuelve "function"

typeof Math;     // devuelve "object"

typeof Option;   // devuelve "function"

typeof String;   // devuelve "function"
```

Operadores relacionales

Un operador relacional compara sus operandos y retorna un valor booleano basado en si la comparación es verdadera.

in

El operador in devuelve true si la propiedad especificada como primer operando se encuentra en el objeto especificado como segundo operando. La sintaxis es:

```
nombrePropiedadoNumero in nombreObjeto
```

Donde nombrePropiedadoNumero es una cadena o expresión numérica que representa un nombre de propiedad o índice de matriz y nombreObjeto es el nombre de un objeto.

Los siguientes ejemplos muestran algunos usos del operador in.

```
// Arrays

var arboles = new Array("secoya", "laurel", "cedro", "roble", "arce");

0 in arboles;       // devuelve true

3 in arboles;       // devuelve true

6 in arboles;       // devuelve false

"laurel" in arboles; // devuelve false (Se debe especificar el número de índice,

                     // no el valor contenido en ese índice)

"length" in arboles; // devuelve true (length es una propiedad del Array)

// Objetos predefinidos

"PI" in Math;         // devuelve true

var miCadena = new String("coral");

"length" in miCadena;  // devuelve true

// Objetos creados

var miCoche = {marca: "Honda", modelo: "Accord", fecha: 1998};

"marca" in miCoche;  // devuelve true

"modelo" in miCoche; // devuelve true
```

instanceof

El operador instanceof devuelve true si el objeto especificado como primer operando es del tipo de objeto especificado como segundo parámetro. La sintaxis es:

```
nombreObjeto instanceof tipoObjeto
```

Donde nombreObjeto es el nombre del objeto que se desea comparar y tipoObjeto es un tipo de objeto, como Date o Array.

Utilice instanceof cuando necesite confirmar el tipo de un objeto en tiempo de ejecución. Por ejemplo, al captar excepciones, puede derivarse a un código de manejo de excepciones diferente dependiendo del tipo de excepción lanzada.

Por ejemplo, el código siguiente utiliza instanceof para determinar si elDia es un objeto de tipo Date. Y debido a que elDia es un objeto Date, las sentencias en la sentencia if se ejecutarán.

```
var elDia = new Date(1995, 12, 17);

if (elDia instanceof Date) {

  // código a ejecutar

}
```

Precedencia de operadores

La precedencia de operadores determina el orden en que estos son aplicados cuando se evalúa una expresión. Esta precedencia puede ser alterada usando paréntesis.

La siguiente tabla describe la precedencia de operadores, de mayor a menor importancia.

Tabla Precedencia de operadores

Tipo de operador	operadores individuales
miembro	. []
llamar / crear instancia	() new
negación / incremento	! ~ - + ++ -- typeof void delete
multiplicación / división	* / %
adición / sustracción	+ -
desplazamiento binario	<< >> >>>
relación	< <= > >= in instanceof
igualdad	== != === !==
AND binario	&
XOR binario	^
OR binario	|
AND lógico	&&
OR lógico	||
condicional	?:

asignación = += -= *= /= %= <<= >>= >>>= &= ^= |=

coma ,

Expresiones

Una expresión es cualquier unidad válida de código que resuelve un valor.

Cada expresión sintáctica válida resuelve a algún valor, pero conceptualmente, hay dos tipos de expresiones: las que tienen efectos secundarios (por ejemplo: aquellas que asignan un valor a una variable) y las que de alguna manera son evaluadas y resuelven un valor.

La expresión x = 7 es un ejemplo del primer tipo. Esta expresión usa el operador = para asignar el valor siete a la variable x. La expresión en sí misma evalúa a siete.

El código 3 + 4 es un ejemplo del segundo tipo de expresiones. Esta expresión usa el operador + para sumar tres y cuatro sin asignar el valor (siete) a ninguna variable.

JavaScript cuenta con las siguientes categorías de expresiones:

Aritméticas: evalúan a un número, por ejemplo 3.14159. (Usos generales Operadores aritméticos.)

Cadenas de caracteres: evalúan a una cadena de caracteres, por ejemplo, "Juan" o "234". (Usos generales Operadores de cadenas de caracteres.)

Lógicas: evalúan a true o false. (A menudo involucran a los Operadores lógicos.)

Expresiones primarias: Palabras clave básicas y expresiones generales en JavaScript.

Expresiones al lado izquierdo: Los valores izquierdos son el destino de una asignación.

Expresiones primarias

Son palabras claves básicas y expresiones generales en JavaScript.

this

Use esta palabra reservada this para referirse al objeto actual. En general, this hace referencia al objeto llamador en un método. Se usa de la siguiente manera:

this["nombreDePropiedad"]

o

this.nombreDePropiedad

Ejemplo 1:

Supongamos una función llamada validate que valida la propiedad value de un objeto, dado un objeto y dos valores, lowval y hival, como extremos de un rango.

```
function validate(obj, lowval, hival){

  if ((obj.value < lowval) || (obj.value > hival))

    alert("¡Valor inválido!");

}
```

Puedes invocar a esta función validate para cada evento onChange de los elementos de un formulario, utilizando this para pasar el campo del formulario como elemento a validar. Por ejemplo:

```
<b>Ingrese un número entre 18 y 99:</b>

<input type="text" name="age" size=3 onChange="validate(this, 18, 99);">
```

Ejemplo 2:

Cuando es combinada con la propiedad form, this puede hacer referencia al formulario padre del elemento actual. En el siguiente ejemplo, el formulario miFormulario contiene un elemento input de tipo text y un elemento input de tipo button. Cuando el usuario hace clic en el botón, se le asigna el nombre del formulario al input de tipo text. El evento onClick del botón usa this.form para hacer referencia al formulario padre, miFormulario.

```
<form name="miFormulario">

   Nombre del formulario:

   <input type="text" name="text1" value="Beluga"/>

   <input type="button" name="button1" value="Mostrar nombre del formulario"

      onClick="this.form.text1.value = this.form.name;"/>

</form>
```

Operador de agrupación

El operador de agrupación ( ) controla la precedencia de la evaluación en las expresiones. Por ejemplo, puedes cambiar el orden de la multiplicación y la división antes de la suma y la resta para que se evalúe la suma primero:

```
var a = 1;

var b = 2;
```

```
var c = 3;

// precedencia por defecto

a + b * c     // 7

// se evalúa por defecto como

a + (b * c)   // 7

// ahora cambiamos la precedencia por defecto

// sumaremos primero antes de multiplicar

(a + b) * c   // 9

// lo cual es equivalente a

a * c + b * c // 9
```

Expresiones al lado izquierdo

Los valores izquierdos son el destino de una asignación

new

Utilice el operador new para crear una instancia de un tipo propio o de uno de los tipos de objetos predefinidos: Array, Boolean, Date, Function, Image, Number, Object, Option, RegExp o String. La semántica es la siguiente:

```
var nombreDeObjeto = new tipoDeObjeto([param1, param2, ..., paramN]);
```

super

La palabra clave super es usada para llamar funciones en el objeto padre. Es útil con clases para llamar al constructor padre, por ejemplo.

```
super([argumentos]); // llama al constructor padre.

super.funcionEnPadre([argumentos]);
```

Operador de propagación

El operador de propagación permite que una expresión sea expandida en situaciones donde se esperan múltiples argumentos (para llamadas a funciones) o múltiples elementos (para Arrays literales).

Ejemplo:

Hoy, si tienes un Array y deseas crear un nuevo Array que contenga también los elementos del primero, la sintaxis de un Array literal no es suficiente y debes recurrir a piezas de código que hagan uso de métodos como push, splice, concat, etc. Con la sintaxis de propagación esto se convierte en algo mucho más simple:

var partes = ["hombros", "rodillas"];

var letra = ["cabeza", ...partes, "y", "dedos"];

De manera similar el operador de propagación funciona también con llamadas a funciones:

function f(x, y, z) { }

var args = [0, 1, 2];

f(...args);

Objeto Number

El objeto integrado Number tiene propiedades para constantes numéricas, como valor máximo, not-a-number, e infinito. No se puede cambiar los valores de estas propiedades y se usan de la siguiente manera:

var biggestNum = Number.MAX_VALUE;

var smallestNum = Number.MIN_VALUE;

var infiniteNum = Number.POSITIVE_INFINITY;

var negInfiniteNum = Number.NEGATIVE_INFINITY;

var notANum = Number.NaN;

Siempre se refiere a una propiedad del objeto Number predefinido como se muestra arriba, y no como una propiedad de un objeto Number se crea a sí mismo.

La siguiente tabla resume las propiedades del objeto Number.

Propiedades de número

Property Description

Number.MAX_VALUE El número más grande representable

Number.MIN_VALUE El número más pequeño representable

Number.NaN "No es un número" valor especial

Number.NEGATIVE_INFINITY Valor infinito negativo Especial; regresó el desbordamiento

Number.POSITIVE_INFINITY Valor infinito positivo Especial; regresó el desbordamiento

Number.EPSILON Diferencia entre uno y el valor más pequeño mayor que uno que se puede representar como una Number.

Number.MIN_SAFE_INTEGER Número entero de seguridad mínima en JavaScript.

Number.MAX_SAFE_INTEGER Entero máximas seguras en JavaScript.

Métodos de Número

Método Descripción

Number.parseFloat() Analiza un argumento de cadena y devuelve un número de punto flotante.

Igual que lo global parseFloat() funcion.

Number.parseInt() Analiza un argumento de cadena y devuelve un número entero de la raíz o base especificada.

Igual que lo globalparseInt() funcion.

Number.isFinite() Determina si el valor pasado es un número finito.

Number.isInteger() Determina si el valor pasado es un entero.

Number.isNaN() Determina si el valor pasado es NaN. Versión más robusta de lo global originales isNaN().

Number.isSafeInteger() Determina si el valor proporcionado es un número que es un número entero seguro.

El prototipo Número proporciona métodos para recuperar información de objetos número en varios formatos. La siguiente tabla resume los métodos de Number.prototype.

Métodos de Number.prototype

Método Descripción

toExponential() Devuelve una cadena que representa el número en notación exponencial.

toFixed() Devuelve una cadena que representa el número en notación de coma fija.

toPrecision() Devuelve una cadena que representa el número con una precisión especificada en notación de coma fija.

Objeto Math

El incorporado Math objeto tiene propiedades y métodos para las constantes y funciones matemáticas. Por ejemplo, la propiedad PI del objeto Math tiene el valor de pi (3.141...), que se usaría en una aplicación como :

Math.PI

Del mismo modo, las funciones matemáticas estándar son métodos de Math. Estos incluyen trigonométricas, logarítmicas, exponenciales, y otras funciones. Por ejemplo, si desea utilizar la función seno trigonométrico, usted escribiría

Math.sin(1.56)

Tenga en cuenta que todos los métodos trigonométricos de Math toman argumentos en radianes.

La siguiente tabla resume los métodos del objeto Math.

Métodos de Math

Método Descripción

abs() Valor Absoluto

sin(), cos(), tan() Funciones trigonométricas estándar; argumento en radianes

asin(), acos(), atan(), atan2() Funciones trigonométricas inversas; devolver valores en radianes

sinh(), cosh(), tanh() Funciones trigonométricas hiperbólicas; devolver valores en radianes.

asinh(), acosh(), atanh() Funciones trigonométricas hiperbólicas inversas; devolver valores en radianes.

pow(), exp(), expm1(), log10(), log1p(), log2()

Funciones exponenciales y logarítmicas.

floor(), ceil() Devoluciones mayor / menor entero inferior / superior o igual al argumento

min(), max() Devuelve la lista de menor o mayor (respectivamente) de números separados por comas de los argumentos

random() Devuelve un número aleatorio entre 0 y 1.

round(), fround(), trunc(), Completan y funciones de truncamiento.

sqrt(), cbrt(), hypot() Raíz cuadrada, raíz cúbica, Raíz cuadrada de la suma de los argumentos cuadrados.

sign() El signo de un número, que indica si el número es positivo, negativo o cero.

clz32(),

imul() Número de líder cero bits en la representación binaria de 32 bits.

El resultado de la multiplicación de 32 bits C-como de los dos argumentos.

A diferencia de muchos otros objetos, nunca se crea un objeto personal de Math. Siempre se utiliza el objeto incorporado en Math.

Objeto Date

JavaScript no tiene un tipo de datos de fecha. Sin embargo, puede utilizar el Date objeto y sus métodos para trabajar con fechas y horas en sus aplicaciones. El objeto Date cuenta con un gran número de métodos para configurar, obtener y manipular fechas. No tiene ninguna propiedad.

JavaScript maneja fechas de manera similar a Java. Los dos idiomas tienen muchos de los mismos métodos de la fecha, y los dos idiomas almacenan fechas como el número de milisegundos desde el 1 de enero de 1970, 00:00:00.

El rango del objeto Date es -100000000 día a 100.000.000 día relativos a 01 de enero 1970 UTC.

Para crear un objeto Date:

var nombreObjetoDate = new Date([parameters]);

donde nombreObjetoDate es el nombre del objeto Date que se está creando; que puede ser un nuevo objeto o una propiedad de un objeto existente.

Llamando Date sin la palabra clave new simplemente convierte la fecha prevista en una representación de cadena.

Los parámetros de la sintaxis anterior pueden ser cualquiera de los siguientes:

Nada: crea la fecha y hora de hoy. por ejemplo, today = new Date();.

Una cadena que representa una fecha en la forma siguiente: "Mes día, año horas: minutos: segundos". Por ejemplo, var Xmas95 = new Date("December 25, 1995 13:30:00"). Si omite horas, minutos o segundos, el valor se establece en cero.

Un conjunto de valores enteros para año, mes y día. Por ejemplo, var Xmas95 = new Date(1995, 11, 25).

Un conjunto de valores enteros para año, mes, día, hora, minutos y segundos. Por ejemplo, var Xmas95 = new Date(1995, 11, 25, 9, 30, 0);.

Métodos del objeto Date

Los métodos Date de objetos para la manipulación de fechas y horas caen en estas amplias categorías:

"set" métodos, para el establecimiento de los valores de fecha y hora en Date objects.

"get" métodos, para obtener los valores de fecha y hora de Date objects.

"to" métodos, para devolver valores de cadena de Date objects.

parse and UTC métodos, para el análisis Date strings.

Con los métodos "get" y "set" puede obtener y establecer segundos, minutos, horas, días del mes, día de la semana, meses y años por separado. Hay un método getDay que devuelve el día de la semana, pero ningún método setDay correspondiente, ya que el día de la semana se ajusta automáticamente. Estos métodos utilizan números enteros para representar estos valores como sigue:

Seconds y minutes: 0 a59

Hours: 0 a 23

Day: 0 (Domingo) a 6 (sábado)

Date: 1 al 31 (día del mes)

Months: 0 (enero) a 11 (diciembre)

Year: año desde 1900

Por ejemplo, suponga que define la siguiente fecha:

var navidad95 = new Date("December 25, 1995");

Then navidad95.getMonth() returns 11, and navidad95.getFullYear() returns 1995.

Los métodos getTime y setTime son útiles para comparar las fechas. El método getTime devuelve el número de milisegundos desde el 1 de enero de 1970, 00:00:00 para un objeto Date.

Por ejemplo, el siguiente código muestra el número de días para que termine el año en curso:

```
var hoy = new Date();

var finDeAnio = new Date(1995, 11, 31, 23, 59, 59, 999); // Define día y mes

finDeAnio.setFullYear(hoy.getFullYear()); // Define el año de este año

var msPorDia = 24 * 60 * 60 * 1000; // Número de milisegundos por día

var diasFaltantes = (finDeAnio.getTime() - hoy.getTime()) / msPorDia;

var diasFaltantes = Math.round(daysLeft); //Regresa los días sobrantes en el año
```

En este ejemplo se crea un objeto Date llamado hoy que contiene la fecha de hoy. A continuación, se crea un objeto Date llamado finDeAnio y establece el año para el año en curso. Luego, utilizando el número de milisegundos por día, calcula el número de días entre hoy y finDeAnio, utilizando getTime y redondeando a un número entero de días.

El método parse es útil para la asignación de valores de cadenas de fecha a los objetos Date existentes. Por ejemplo, el código siguiente utiliza parse y setTime para asignar un valor de fecha al objeto IPOdate:

```
var IPOdate = new Date();

IPOdate.setTime(Date.parse("Aug 9, 1995"));
```

Ejemplo

En el siguiente ejemplo, la función JSClock() devuelve la hora en el formato de un reloj digital.

```
function JSClock() {

  var tiempo = new Date();

  var hora = tiempo.getHours();

  var minutos = tiempo.getMinutes();

  var segundos = tiempo.getSeconds();

  var temp = "" + ((hora > 12) ? hora - 12 : hora);

  if (hora == 0)
```

```
  temp = "12";
 temp += ((minutos < 10) ? ":0" : ":") + minutos;
 temp += ((segundos < 10) ? ":0" : ":") + segundos;
 temp += (hora >= 12) ? " P.M." : " A.M.";
 return temp;
}
```

La función JSClock crea primero un nuevo objeto Date llamado tiempo; ya que no se dan argumentos, el tiempo se crea con la fecha y la hora actual. Entonces las llamadas a los métodos getHours, getMinutes y getSeconds asignan el valor de la hora actual, los minutos y segundos a hora, minutos y segundos.

Las siguientes cuatro declaraciones construyen un valor de cadena en función del tiempo. La primera sentencia crea una variable temp, asignándole un valor utilizando una expresión condicional; si hora es mayor que 12, (hora - 12), de lo contrario simplemente horas, a menos que hora es 0, en cuyo caso se convierte en 12.

La siguiente sentencia añade el valor de minutos a temp. Si el valor de minutos es menor que 10, la expresión condicional añade una cadena con un precedente cero; de lo contrario se añade una cadena con dos puntos de demarcación. A continuación, una declaración anexa un valor segundos a temp de la misma manera.

Por último, una expresión condicional añade "P. M." a temp si hora es 12 o mayor; de lo contrario, se anexa "A. M." a temp.

Cadenas (String)

Las cadenas o String de JavaScript se utilizan para representar datos de texto. Es un conjunto de "elementos" de valores enteros de 16-bit no signados. Cada elemento en la cadena ocupa una posición en ella. El primer elemento se encuentra en el índice 0, el siguiente en el índice 1, y así subsecuentemente. La longitud de una cadena es el número de elementos en ella. Es posible crear cadenas utilizando literales de cadena u objetos de cadena.

Literales de cadena

Es posible crear cadenas simples utilizando tanto comillas simples como dobles:

'foo'

"bar"

Cadenas más avanzadas pueden ser creadas utilizando secuencias de escape:

Secuencias de escape hexadecimal

El número después de \x es interpretado como un número hexadecimal.

'\xA9' // "©"

Secuencias de escape Unicode

Las secuencias de escape Unicode requieren al menos cuatro dígitos hexadecimales después de \u.

'\u00A9' // "©"

Puntos de escape de código Unicode

Nuevo en ECMAScript 6. Con los puntos de escape de código Unicode, cualquier caracter puede ser escapado usando números hexadecimales así que es posible usar puntos de código hasta 0x10FFFF. Con los escapes Unicode simples a menudo es necesario escribir las mitades separadamente para conseguir lo mismo.

'\u{2F804}'

// lo mismo con escapes Unicode simples

'\uD87E\uDC04'

Objetos cadena

El objeto String es un envoltorio alrededor del tipo de datos de cadena original.

var s = new String("foo"); // Crea un objeto String

console.log(s); // Muestra: { '0': 'f', '1': 'o', '2': 'o'}

typeof s; // Devuelve 'object'

Es posible llamar cualquiera de los métodos del objeto String en un valor literal de cadena—JavaScript automáticamente convierte el literal de cadena a un objeto String temporal, invoca el método, y luego descarta el objeto String temporal. También es posible utilizar la propiedad String.length con un literal de cadena.

Se debería utilizar literales de cadena a menos que específicamente sea necesario usar un objeto String, porque los objetos String pueden tener un comportamiento contraintuitivo. Por ejemplo:

var s1 = "2 + 2"; // Crea un valor literal de cadena

var s2 = new String("2 + 2"); // Crea un nuevo objeto de cadena

eval(s1); // Devuelve el número 4

eval(s2); // Devuelve la cadena "2 + 2"

Un objeto String tiene una propiedad, lenght, que indica el número de caracteres en la cadena. Por ejemplo, el código siguiente asigna a x el valor 13, porque "Hello, World!" tiene 13 caracteres, cada uno representado por una unidad de código UTF-16. Se puede acceder a cada unidad de código utilizando un estilo de corchetes de arreglo. No es posible cambiar caracteres debido a que las cadenas son objetos tipo arreglo inmutables:

var mystring = "Hello, World!";

var x = mystring.length;

mystring[0] = 'L' // No produce ningún efecto

mystring[0]; // Esto devuelve "H"

Los caracteres cuyo valor Unicode escalar sea mayor que U+FFFF (tales como algunos caracteres Chinos/Japoneses/Coreanos/Vietnamitas infrecuentes y algunos emoji) son almacenados en UTF-16 con dos unidades de código sustitutas. Por ejemplo, una cadena que contenga el caracter único U+1F600 "cara con sonrisa burlona Emoji" tendrá una longitud de 2. El acceso a las unidades de código individual utilizando corchetes puede tener consecuencias indeseables como la formación de cadenas con unidades de código sustitutas no pareadas, en violación al estándar Unicode. Ver también String.fromCodePoint() o String.prototype.codePointAt().

Un objeto String tiene una variedad de métodos: por ejemplo aquellos que devuelven una modificación de la misma cadena, tales como substring y toUpperCase.

La siguiente tabla resume los métodos de los objetos String.

Métodos de String

Método Descripción

charAt, charCodeAt, codePointAt Devuelve el caracter o el código del caracter en la posición especificada en la cadena.

indexOf, lastIndexOf Devuelve la posición de la subcadena en la cadena o la última posición de una subcadena especificada respectivamente.

startsWith, endsWith, includes Devuelve si la cadena empieza, termina o contiene una cadena especificada, o no.

concat Combina el texto de dos cadenas y retorna una nueva cadena.

fromCharCode, fromCodePoint Construye una cadena desde la secuencia de valores Unicode especificada. Este es un método de la clase String, no una instancia de String.

split Divide un objeto String en un array de strings separados por substrings.

slice Extrae una sección de un string y devuelve un nuevo string.

substring, substr

Devuelve un substring del string, bien especificando el comienzo y el final, o bien el índice inicial y la longitud del substring.

match, replace, search Para trabajar con expresiones regulares

toLowerCase, toUpperCase

Devuelve el string en mayúsculas o en minúsculas

normalize

Devuelve es string Normalizado en Unicode.

repeat

Devuelve un string formado por los elementos del objetos repetidos el número de veces que le indiquemos.

trim Elimina los espacios en blanco del principio y del final del string.Cadenas de plantillas multilínea

Plantillas de cadena de texto son cadenas literales que permiten expresiones incrustadas. Se pueden utilizar cadenas multilínea y funciones de interpolación de cadenas.

Las plantillas de cadenas de texto están marcadas por el carácter (` `) (acento grave) en lugar de las comillas dobles o simples. Las plantillas de cadenas de texto pueden contener marcadores (placeholders). Son indicados por el signo de moneda y llaves rizadas (${expression}).

Multilíneas

Cualquier nueva línea de caracteres insertada en la fuente son parte de una plantilla de cadena de texto. Utilizando cadenas de texto normales, se puede usar la siguiente sintaxis para tener una cadena multilínea:

```
console.log("línea de texto 1\n\
línea de texto 2");
```

// "línea de texto 1

// línea de texto 2"

Para tener el mismo efecto con cadenas de texto multilínea, ahora se puede escribir:

console.log(`línea de texto 1

línea de texto 2`);

// "línea de texto 1

// línea de texto 2"

Expresiones incrustadas

Para incrustar expresiones dentro de cadenas normales, puedes utilizar la siguiente sintaxis:

var a = 5;

var b = 10;

console.log("Quince es " + (a + b) + " y\nno " + (2 * a + b) + ".");

// "Quince es 15 y

// no 20."

Ahora con las plantillas de cadenas de texto, es posible usar una manera más rápida (syntactic sugar) para hacer sustituciones mucho más leíbles:

var a = 5;

var b = 10;

console.log(`Quince es ${a + b} y\nno ${2 * a + b}.`);

// "Quince es 15 y

// no 20."

Objetos Matriz o Array

Una matriz o array es un conjunto ordenado de valores al que se refiere con un nombre y un índice. Por ejemplo: podría tener un array llamado emp que contenga a los nombre de los empleados indexados por su número de empleado. Así emp[1] sería el empleado número uno , emp[2] seria el empleado número dos, y así sucesivamente.

JavaScript no tiene un tipo de dato de matriz explícito. Sin embargo, podemos utilizar el objeto Array predefinido y sus métodos para trabajar con matrices en sus aplicaciones. El

objeto Array tiene métodos para manipular matrices de varias maneras, tales como unirlas, invertirlas, y ordenarlas. Tiene una propiedad para determinar la longitud de la matriz y otras propiedades para su uso con expresiones regulares.

Creando una matriz o array

Las siguientes declaraciones crean matrices equivalentes :

var arr = new Array(element0, element1, ..., elementN);

var arr = Array(element0, element1, ..., elementN);

var arr = [element0, element1, ..., elementN];

element0, element1, ..., elementN es una lista de valores para los elementos de la matriz. Cuando se especifican estos valores , la matriz se inicializa con ellos como elementos de la matriz. La propiedad de length de la matriz se establece con el número de argumentos.

Los corchetes en la sintaxis están llamando a "literal de la matriz" o "inicializador de matriz". Siendo este más corto que otras formas de creación de la matriz, y así se prefiere generalmente. Vea Array literals para más detalles. Vea literales de conjunto para obtener más detalles.

Para crear una matriz con longitud distinta de cero, pero sin ningún elemento, se puede utilizar cualquiera de las siguientes opciones:

var arr = new Array(arrayLength);

var arr = Array(arrayLength);

// Esto tiene exactamente el mismo efecto

var arr = [];

arr.length = arrayLength;

Nota: en el código de arriba, arrayLength debe ser Number. De otra forma, se creará un array con un único valor (el que fué provisto). Llamar arr.length devolverá arrayLength, pero el array realmente contendrá elementos vacíos (undefined). Correr un bucle for...in en el array no devolverá ninguno de sus elementos.

Adicionalmente a la nueva variable definida como se muestra arriba, los arrays también pueden ser asignados como una propiedad de un objeto nuevo o ya existente:

var obj = {};

// ...

obj.prop = [element0, element1, ..., elementN];

// OR

var obj = {prop: [element0, element1, ..., elementN]}

Si deseas inicializar un array con un solo elemento, y este debe ser un valor de tipo Number, deberás utilizar la sintaxis de corchetes, Cuando se pasa solo un valor Number al constructor o función Array(), este es interpretado como la longitud del array (arrayLength), no como un solo elemento.

var arr = [42];

var arr = Array(42); // Crea un array sin elementos,

// pero con una arr.length de 42

// El código de arriba es equivalente a lo siguiente

var arr = [];

arr.length = 42;

Llamar Array(N) resulta en un error RangeError, si N es un entero con una parte fraccionaria distinta de 0. El siguiente ejemplo ilustra este comportamiento.

var arr = Array(9.3); // RangeError: Longitud de array inválida

Si tu código necesita crear arrays con elementos únicos de tipos arbitrarios, es más seguro utilizar literales de array. O, crear un array vacío antes que agregar un único elemento a el.

LLenando un array

Puedes llenar un array al asignar valores a sus elementos. Por ejemplo,

var emp = [];

emp[0] = "Casey Jones";

emp[1] = "Phil Lesh";

emp[2] = "August West";

Nota: si en el código de arriba utilizas un valor no entero en el operador del array, se creará una propiedad en el objeto que representa el array, y no un elemento del array.

var arr = [];

```
arr[3.4] = "Oranges";

console.log(arr.length);              // 0

console.log(arr.hasOwnProperty(3.4));   // true
```

También puedes llenar un array al crearlo:

```
var myArray = new Array("Hello", myVar, 3.14159);

var myArray = ["Mango", "Apple", "Orange"]
```

Hacer referencia a elementos de un array

Puedes hacer referencia a los elementos contenidos en un array, usando el número ordinal del elemento. Por ejemplo, supongamos que defines el siguiente array:

```
var myArray = ["Wind", "Rain", "Fire"];
```

Ahora puedes hacer referencia el primer elemento del array conmyArray[0] y al segundo elemento de array con myArray[1]. Los índices de elemento comienzan siempre por cero.

Nota: the array operator (square brackets) is also used for accessing the array's properties (arrays are also objects in JavaScript). For example,

```
var arr = ["one", "two", "three"];

arr[2];  // three

arr["length"];  // 3
```

Iterando sobre arrays

Una operación común es iterar sobre los valores de un array, procesando cada uno de ellos de alguna manera. La forma más sencilla de hacerlo es la siguiente:

```
var colors = ['red', 'green', 'blue'];

for (var i = 0; i < colors.length; i++) {

  console.log(colors[i]);

}
```

Si sabes que ninguno de los elementos en tu array evalúan a falso en un contexto booleano o si sabes que tu array consiste sólo de nodos DOM, por ejemplo, puedes usar una forma más eficiente:

```
var divs = document.getElementsByTagName('div');

for (var i = 0, div; div = divs[i]; i++) {

  /* Process div in some way */

}
```

Esto permite evitar la sobrecarga de chequear la longitud del array, y asegura que la variable div sea reasignada al ítem actual una vez por vuelta de ciclo.

El método forEach() provee otra forma de iterar sobre el array:

```
var colors = ['red', 'green', 'blue'];

colors.forEach(function(color) {

  console.log(color);

});
```

La función pasada al forEach es ejecutada una vez por cada ítem en el array, con el ítem pasado como argumento a la función. Valores no asignados no son iterados en el ciclo forEach.

Observar que los elementos del array omitidos cuando el array está definido no son listados cuando se itera pero son listados cuando se ha asignado manualmente a los elementos:

```
var array = ['first', 'second', , 'fourth'];

// returns ['first', 'second', 'fourth'];

array.forEach(function(element) {

  console.log(element);

})

if(array[2] === undefined) { console.log('array[2] is undefined'); } // true

var array = ['first', 'second', undefined, 'fourth'];

// returns ['first', 'second', undefined, 'fourth'];

array.forEach(function(element) {

  console.log(element);

})
```

Métodos de Array

concat() junta dos arrays y retorna uno nuevo..

var myArray = new Array("1", "2", "3");

myArray = myArray.concat("a", "b", "c");

// myArray es ahora ["1", "2", "3", "a", "b", "c"]

join(deliminator = ',') une todos los elementos del array en una cadena.

var myArray = new Array("Wind", "Rain", "Fire");

var list = myArray.join(" - "); // list is "Wind - Rain - Fire"

push() agrega uno o más elementos al final de un arreglo y retorna la longitud resultante del arreglo.

var myArray = new Array("1", "2");

myArray.push("3"); // myArray is now ["1", "2", "3"]

pop() remueve el último elemento del arreglo y lo devuelve.

var myArray = new Array("1", "2", "3");

var last = myArray.pop();

// myArray se convierte en ["1", "2"], last = "3"

shift() quita el primer elemento del arreglo y lo devuelve.

var myArray = new Array ("1", "2", "3");

var first = myArray.shift();

// myArray is now ["2", "3"], first is "1"

unshift() agrega uno o más elementos al principio del arreglo y devuelve la longitud del arreglo.

```
var myArray = new Array ("1", "2", "3");

myArray.unshift("4", "5");

// myArray becomes ["4", "5", "1", "2", "3"]
```

slice(start_index, upto_index) extrae una sección del arreglo y devuelve un nuevo arreglo.

```
var myArray = new Array ("a", "b", "c", "d", "e");

myArray = myArray.slice(1, 4); // starts at index 1 and extracts all elements

                    // until index 3, returning [ "b", "c", "d"]
```

splice(index, count_to_remove, addElement1, addElement2, ...) quita elementos del arreglo y opcionalmente los reemplaza por otros..

```
var myArray = new Array ("1", "2", "3", "4", "5");

myArray.splice(1, 3, "a", "b", "c", "d");

// myArray is now ["1", "a", "b", "c", "d", "5"]

// This code started at index one (or where the "2" was),

// removed 3 elements there, and then inserted all consecutive

// elements in its place.
```

reverse() transpone los elementos de un arreglo: el primer elemento pasa a ser el último y viceversa.

```
var myArray = new Array ("1", "2", "3");

myArray.reverse();

// transpone el array, o sea, lo da vuelta a myArray = [ "3", "2", "1" ]
```

sort() ordena los elementos del arreglo.

```
var myArray = new Array("Wind", "Rain", "Fire");

myArray.sort();

// ordena el array  myArray = [ "Fire", "Rain", "Wind" ]
```

sort() puede también usar una función para determinar cómo los elementos del arreglo son comparados. Esta función compara dos valores y devuelve uno de tres valores:

Por ejemplo, la función siguiente ordenará por la última letra de una cadena:

```
var sortFn = function(a, b){

  if (a[a.length - 1] < b[b.length - 1]) return -1;

  if (a[a.length - 1] > b[b.length - 1]) return 1;

  if (a[a.length - 1] == b[b.length - 1]) return 0;

}

myArray.sort(sortFn);

// sorts the array so that myArray = ["Wind","Fire","Rain"]
```

Si a es menor que b entonces se devuelve -1 (o cualquier número negativo)

si a es mayor que b entonces se devuelve 1 (o cualquier número positivo)

si a y b son equivalentes o iguales, devuelve 0.

indexOf(searchElement[, fromIndex]) busca en el arreglo por el elemento searchElement y devuelve el índice del primero encontrado.

```
var a = ['a', 'b', 'a', 'b', 'a'];

console.log(a.indexOf('b')); // muestra 1

// Ahora de nuevo, empezando por el siguiente después del encontrado

console.log(a.indexOf('b', 2)); // muestra 3

console.log(a.indexOf('z')); // muestra  -1, porque 'z' no fué encontrado
```

lastIndexOf(searchElement[, fromIndex]) funciona como indexOf, pero empieza al final y busca de atrás para adelante.

```
var a = ['a', 'b', 'c', 'd', 'a', 'b'];

console.log(a.lastIndexOf('b')); // muestra 5

// Ahora de nuevo, empezando desde la última ocurrencia.

console.log(a.lastIndexOf('b', 4)); // muestra 1

console.log(a.lastIndexOf('z')); // muestra 1
```

forEach(callback[, thisObject]) ejecuta la función callback en cada ítem del arreglo.

```
var a = ['a', 'b', 'c'];

a.forEach(function(element) { console.log(element);} );

//muestra cada ítem
```

map(callback[, thisObject]) devuelve un nuevo arreglo con el resultado de aplicar la función callback a cada ítem.

```
var a1 = ['a', 'b', 'c'];

var a2 = a1.map(function(item) { return item.toUpperCase(); });

console.log(a2); // muestra A,B,C
```

filter(callback[, thisObject]) devuelve un nuevo arreglo conteniendo los items de los cuales al aplicar la función callback resultaron en verdadero true. callback es una funcion que retorna un booleano.

```
var a1 = ['a', 10, 'b', 20, 'c', 30];

var a2 = a1.filter(function(item) { return typeof item == 'number'; });

console.log(a2); // muestra 10,20,30
```

every(callback[, thisObject]) devuelve verdadero si la función callback devuelve verdadero por cada item del arreglo.

```
function isNumber(value){
  return typeof value == 'number';
}
var a1 = [1, 2, 3];
console.log(a1.every(isNumber)); // muestra true
var a2 = [1, '2', 3];
console.log(a2.every(isNumber)); // muestra false
```

some(callback[, thisObject]) devuelve verdadero si la función callback devuelve verdadero para al menos un elemento del arreglo..

```
function isNumber(value){
  return typeof value == 'number';
}
var a1 = [1, 2, 3];
console.log(a1.some(isNumber)); // muestra  true
var a2 = [1, '2', 3];
console.log(a2.some(isNumber)); // muestra true
var a3 = ['1', '2', '3'];
console.log(a3.some(isNumber)); // muestra  false
```

Los métodos de arriba que toman como parámetro una función callback son conocidas como métodos iterativos, porque iteran sobre todo el arreglo en alguna manera particular. Cada uno toma un segundo argumento opcional llamado thisObject.

reduce(callback[, initialValue]) aplica callback(firstValue, secondValue) para reducir la lista de items a un valor unitario..

```
var a = [10, 20, 30];
var total = a.reduce(function(first, second) { return first + second; }, 0);
```

```
console.log(total) // muestra 60
```

reduceRight(callback[, initalvalue]) funciona como reduce(), pero empieza en el último elemento.

Arreglos Multi-dimensionales

Los arreglos pueden anidarse, significa que un arreglo contiene otro arreglo como elemento.

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

for (i = 0; i < 4; i++) {

  a[i] = new Array(4);

  for (j = 0; j < 4; j++) {

    a[i][j] = "[" + i + "," + j + "]";

  }

}
```

El ejemplo anterior crea un arreglo con las siguientes filas:

Row 0: [0,0] [0,1] [0,2] [0,3]

Row 1: [1,0] [1,1] [1,2] [1,3]

Row 2: [2,0] [2,1] [2,2] [2,3]

Row 3: [3,0] [3,1] [3,2] [3,3]

Control de flujo y manejo de errores

Sentencia de bloque

La sentencia de bloque es el tipo de sentencia más básico y se utiliza para agrupar sentencias. El bloque se delimita entre un par de llaves:

```
{

  sentencia_1;

  sentencia_2;

  .
```

```
 .
 .
 sentencia_n;
}
```

Ejemplo

Los bloques de sentencias son comúnmente utilizados para sentencias de control de flujo (ej. if, for, while).

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

En este caso { x++; } es el bloque de sentencias.

Importante: Javascript no tiene ámbito a nivel bloque en versiones anteriores a ECMAScript 6. Las variables introducidas dentro de un bloque pertenecen a la función o script que lo contiene y el efecto de declararlas persiste más allá del bloque mismo. En otras palabras, los bloques no introducen un nuevo ámbito. Si bien los bloques "Standalone" son válidos no deberían ser utilizados en Javascript ya que no se comportan como los bloques de C o Java. Por ejemplo:

```
var x = 1;
{
 var x = 2;
}
console.log(x); // imprime 2
```

Este código imprime el número 2 dado que la sentencia var x dentro del bloque está en el mismo ámbito que la sentencia var x definida antes del bloque. En C o Java el equivalente de este código imprimiría 1.

A partir de ECMAScript 6, se introduce el ámbito a nivel bloque utilizando let para declarar las variables.

Sentencias condicionales

Una sentencia condicional es un conjunto de comandos que se ejecutan si una condición es verdadera. JavaScript soporta dos sentencias condicionales: if...else y switch

Sentencia if...else

Se utiliza la sentencia if para comprobar si la condición lógica es verdadera. Se utiliza la opción else para ejecutar un sentencia si la condición es falsa. A continuación se muestra un ejemplo de if...else:

```
if (condición) {

   sentencia_1;

} else {

   sentencia_2;

}
```

Aquí la condición puede ser cualquier expresión que se evalúa a true o false. Consultar Boolean para una explicación de cómo se evalúa true y false. Si la condición es verdadera, se ejecuta sentencia_1; de lo contrario, se ejecuta sentencia_2. La sentencia_1 y la sentencia_2 pueden ser cualquier sentencia, incluyendo otras sentencias anidadas en if.

También puedes componer sentencias más complejas usando else if para tener múltiples condiciones, como se muestra a continuación:

```
if (condición_1) {

  sentencia_1;

} else if (condición_2) {

  sentencia_2;

} else if (condición_n) {

  sentencia_n;

} else {

  ultima_sentencia;

}
```

En el caso de condiciones múltiples solamente la primera condición lógica que evalúa a verdadero va a ser ejecutada. Para ejecutar múltiples sentencias, agruparlas dentro de sentencias de bloque ({ ... }) . En general, usar siempre sentencias de bloque es una buena práctica, sobre todo cuando se anidan sentencias if:

```
if (condición) {
```

```
  ejecutar_sentencia_1_si_condición_es_verdadera;
  ejecutar_sentencia_2_si_condición_es_verdadera;
} else {
  ejecutar_sentencia_3_si_condición_es_falsa;
  ejecutar_sentencia_4_si_condición_es_falsa;
}
```

Es aconsejable no usar asignación simple dentro de una expresión condicional porque dicha asignación puede ser confundida con el comparador de igualdad cuando se lee de pasada el código. Por ejemplo, no usar el siguiente código:

```
if (x = y) {
  /* sentencias aquí */
}
```

Si necesitas usar una asignación dentro de una expresión de condición, una práctica común es poner paréntesis adicionales alrededor de la asignación. Por ejemplo:

```
if ((x = y)) {
  /* sentencias aquí */
}
```

Valores falsos

Los siguientes valores se evalúan como falso (también conocidos como valores Falsy):

false

undefined

null

0

NaN

la cadena vacía ("")

El resto de valores, incluídos todos los objetos, son evaluados como verdadero cuando son pasados a una sentencia condicional.

No confundir los valores primitivos booleanos true y false con los valores true y false del objeto Boolean. Por ejemplo:

```
var b = new Boolean(false);

if (b) // Esta condición se evalua a true

if (b == true) // Esta condición se evalua a false
```

Ejemplo

En el siguiente ejemplo, la función comprobarDatos devuelve true si el número de caracteres en un objeto Text es tres; en otro caso, muestra una alerta y devuelve false.

```
function comprobarDatos() {
  if (document.form1.threeChar.value.length == 3) {
    return true;
  } else {
    alert("Introduce exactamente tres caracteres. " +
    document.form1.threeChar.value + " no es válido.");
    return false;
  }
}
```

switch

Una sentencia switch permite a un programa evaluar una expresión e intentar igualar el valor de dicha expresión a una etiqueta de caso (case). Si se encuentra una coincidencia, el programa ejecuta la sentencia asociada. Una sentencia switch se describe como se muestra a continuación:

```
switch (expresión) {
  case etiqueta_1:
    sentencias_1
    [break;]
  case etiqueta_2:
    sentencias_2
```

```
  [break;]

  ...

 default:

  sentencias_por_defecto

  [break;]

}
```

El programa primero busca una cláusula case con una etiqueta que coincida con el valor de la expresión y, entonces, transfiere el control a esa cláusula, ejecutando las sentencias asociadas a ella. Si no se encuentran etiquetas coincidentes, el programa busca la cláusula opcional default y, si se encuentra, transfiere el control a esa cláusula, ejecutando las sentencias asociadas. Si no se encuentra la cláusula default, el programa continúa su ejecución por la siguiente sentencia al final del switch. Por convención, la cláusula por defecto es la última cláusula, aunque no es necesario que sea así.

La sentencia opcional break asociada con cada cláusula case asegura que el programa finaliza la sentencia switch una vez que la sentencia asociada a la etiqueta coincidente es ejecutada y continúa la ejecución por las sentencias siguientes a la sentencia switch. Si se omite la sentencia break, el programa continúa su ejecución por la siguiente sentencia que haya en la sentencia switch.

Ejemplo

En el siguiente ejemplo, si tipoFruta se evalúa como "Plátanos", el programa iguala el valor con el caso "Plátanos" y ejecuta las sentencias asociadas. Cuando se encuentra la sentencia break, el programa termina el switch y ejecuta las sentencias que le siguen. Si la sentencia break fuese omitida, la sentencia para el caso "Cerezas" también sería ejecutada.

```
switch (tipoFruta) {

 case "Naranjas":

  console.log("Naranjas cuestan 53$ (0,59€) el kilo.");

  break;

 case "Manzanas":

  console.log("Manzanas cuestan 29$ (0,32€) el kilo.");

  break;

 case "Plátanos":
```

```
    console.log("Plátanos cuestan 43$ (0,48€) el kilo.");
    break;
  case "Cerezas":
    console.log("Cerezas cuestan 268$ (3,00€) el kilo.");
    break;
  case "Mangos":
    console.log("Mangos cuestan 50$ (0,56€) el kilo.");
    break;
  case "Papayas":
    console.log("Mangos y papayas cuestan 249$ (2,79€) el kilo.");
    break;
  default:
    console.log("Disculpa, no tenemos el tipo de fruta " + tipoFruta + ".");
}

console.log("¿Te gustaría tomar algo?");
```

Sentencias de manejo de excepciones

Puedes lanzar excepciones usando la sentencia throw y manejarlas usando las sentencias try...catch.

throw statement

try...catch statement

Tipos de excepciones

Prácticamente cualquier objeto puede ser lanzado en JavaScript. Sin embargo, no todos los objetos lanzados son creados igual. Mientras que es bastante común para lanzar números o strings como errores, frecuentemente son más efectivos utilizar uno de los tipos de excepciones específicamente creados para este propósito:

Excepciones ECMAScript

DOMException and DOMError

Sentencia throw

Utiliza la sentencia throw para lanzar una excepción. Cuando lanzas un excepción, se especifica la expresión que contiene el valor para ser lanzado:

throw expresión;

Puedes lanzar cualquier expresión, no solo expresiones de un tipo especifico. En el siguente código lanzamos varias excepciones de varios tipos:

```
throw "Error2";   // Tipo string

throw 42;         // Tipo número

throw true;       // Tipo booleano

throw {toString: function() { return "¡Soy un objeto!"; } };
```

Note: Puedes especificar un objeto cuando lanzas una excepción. A continuación, puedes hacer referencia a las propiedades del objeto en un bloque catch.

```
// Crear un tipo de objeto UserException

function UserException (aviso){

  this.aviso=aviso;

  this.nombre="UserException";

}

// Convertimos la excepción a String cuando es utilizada como un String

// (E.j: Un error de consola)

UserException.prototype.toString = function () {

  return this.nombre + ': "' + this.aviso + '"';

}

// Crea una instancia del objeto y lo lanza

throw new UserException("Value too high");
```

try...catch

La sentencia try...catch marca un bloque de instrucciones a intentar que pueden causar alguna excepción, y declarar una o más respuestas en caso de que una excepción sea arrojada. Si una excepción es arrojada, la sentencia try...catch se encarga de atraparla.

La sentencia try...catch consiste en un bloque try, el cual contiene una o más instrucciones, y ninguno o varios bloques catch, conteniendo sentencias que especifican qué hacer si una excepción es arrojada en un bloque try. Se desea que las instrucciones dentro del bloque try se ejecuten con éxito, de caso contrario caerán en el bloque catch para ser controladas. Si alguna instrucción dentro del bloque try (o en una función llamada dentro del bloque try) arroja una excepción, el control pasa inmediatamente al bloque catch. Si ninguna excepción es arrojada en el bloque try, el bloque catch es ignorado. Por último se ejecuta el bloque finally luego de que los bloques try y catch hayan sido ejecutados pero antes de las instrucciones que se encuentren a continuación de la sentencia try...catch.

El siguiente ejemplo usa la sentencia try...catch. El ejemplo llama a una función que retorna el nombre de un mes desde un arreglo basado en un valor pasado como argumento a la función. Si el valor no corresponde con el número de un mes (entre 1 y 12), una excepción es arrojada con el valor "InvalidMonthNo" y las instrucciones en el bloque catch le asignan a la variable monthName el valor de unknown.

```
function getMonthName (mo) {
  mo = mo-1; // Ajusta el índice del arreglo para el arreglo de meses (1=Jan, 12=Dec)
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul",
                "Aug","Sep","Oct","Nov","Dec"];
  if (months[mo] != null) {
    return months[mo];
  } else {
    throw "InvalidMonthNo"; //Arroja la palabra "InvalidMonthNo" al ocurrir una excepción
  }
}
try { // instrucciones a probar
  monthName = getMonthName(myMonth); // La función puede arrojar una excepción
}
```

```
catch (e) {
  monthName = "unknown";
  logMyErrors(e); // Pasa el objeto de la excepción a un manejador de errores
}
```

El bloque catch

Un bloque catch es usado para manejar todas las excepciones que pueden ser generadas en el bloque try.

```
catch (catchID) {
  instrucciones
}
```

El bloque catch especifica un identificador (catchID en la sintaxis anterior) que tiene el valor especificado por la sentencia throw; puedes usar este identificador para obtener información acerca de la excepción que fue arrojada. JavaScript crea este identificador cuando ha entrado en el bloque catch; el identificador dura mientras dure el bloque catch; después de que el bloque catch termine su ejecución, el identificador ya no estará disponible.

Por ejemplo, el siguiente código arroja una excepción. Cuando la excepción ocurre, el control es transferido al bloque catch.

```
try {
  throw "myException" // genera una excepción
}
catch (e) {
  // instrucciones para manejar cualquier excepción generada
  logMyErrors(e) // Pasa el objeto de excepción a un manejador de errores
}
```

El bloque finally

El bloque finally contiene instrucciones para ejecutar luego de la ejecución del bloque try y el bloque catch pero antes de las instrucciones ubicadas luego de la sentencia try...catch. El bloque finally se ejecuta cuando se haya arrojado o no una excepción. Si una excepción es

arrojada, las instrucciones en el bloque finally se ejecutan incluso si no existe un bloque catch que maneje la excepción.

Se puede usar el bloque finally para hacer que tu script falle con gracia cuando una excepción ocurre; por ejemplo, puedes tener la necesidad de liberar un recurso que tu script tiene ocupado. El siguiente ejemplo abre un archivo y luego ejecuta instrucciones que usan el archivo (JavaScript del lado del servidor permite acceder a archivos). Si una excepción es arrojada mientras el archivo está abierto, el bloque finally cierra el archivo antes de que el script falle.

```
openMyFile();

try {

  writeMyFile(theData); // Esto puede arrojar un error

} catch(e) {

  handleError(e); // Si ocurre un error es manejado

} finally {

  closeMyFile(); // Siempre cierra el recurso

}
```

Si el bloque finally retorna un valor, este valor se convierte en el valor de retorno de toda la sentencia try-catch-finally, independientemente de cualquier sentencia return en el bloque try y el bloque catch:

```
function f() {

  try {

    console.log(0);

    throw "bogus";

  } catch(e) {

    console.log(1);

    return true; // Esta sentencia de retorno es suspendida

                 // hasta que el bloque finally esté completo

    console.log(2); // no alcanzable

  } finally {
```

```
    console.log(3);
    return false; // sobreescribe la sentencia de retorno anterior
    console.log(4); // no alcanzable
  }
  // "return false" es ejecutada ahora
  console.log(5); // no alcanzable
}
f(); // console 0, 1, 3; retorna false
```

Sobreescribiendo los valores retornados por el bloque finally también aplica a excepciones arrojadas o relanzadas dentro de un bloque catch:

```
function f() {
  try {
    throw "bogus";
  } catch(e) {
    console.log('caught inner "bogus"');
    throw e; // Esta sentencia throw es suspendida hasta que
             // el bloque finally se termine de ejecutar
  } finally {
    return false; // Sobreescribe la sentencia throw anterior
  }
  // "return false" es ejecutado ahora
}
try {
  f();
} catch(e) {
  // Esta nunca es encontrada porque la sentencia throw dentro
```

```
  // del bloque catch es sobrescrita por la sentencia return
  // en el bloque finally
  console.log('caught outer "bogus"');
}
// SALIDA
// atrapado dentro de "bogus"
```

Sentencias try...catch anidadas

Es posible anidar una o más sentencias try...catch. Si una sentencia try...catch interna no posee un bloque catch, la sentencia try...catch exterior verifica si el bloque exterior genera una coincidencia.

Bucles e iteración

sentencia for

Un bucle for se repite hasta que la condición especificada se evalúa como false. El bucle for en JavaScript es similar al de Java y C. Una sentencia for se muestra como sigue:

```
for ([expresionInicial]; [condicion]; [expresionIncremento])
  sentencia
```

Cuando un bucle for se ejecuta, ocurre lo siguiente:

La expresión de inicialización expresionInicial, si existe, se ejecuta. Esta expresión habitualmente inicializa uno o más contadores del bucle, pero la sintaxis permite una expresión con cualquier grado de complejidad. Esta expresión puede también declarar variables.

Se evalúa la expresión condición. Si el valor de condición es true, se ejecuta la sentencia del bucle. Si el valor de condición es false, el bucle for finaliza. Si la expresión condición es omitida, la condición es asumida como verdadera.

Se ejecuta la sentencia. Para ejecutar múltiples sentencias, use un bloque de sentencias ({ ... }) para agruparlas.

Se ejecuta la expresión expresionIncremento, si hay una, y el control vuelve al paso 2.

Ejemplo

La siguiente función contiene una sentencia for que cuenta el número de opciones seleccionadas en una lista (un elemento <select> que permite selección múltiple). La sentencia for declara la variable i y la inicializa a cero. Comprueba que i es menor que el número de opciones en el elemento <select>, ejecuta la sentencia siguiente if, e incrementa i en uno tras cada paso por el bucle.

```
<form name="selectForm">
  <p>
    <label for="musicTypes">Choose some music types, then click the button below:</label>
    <select id="musicTypes" name="musicTypes" multiple="multiple">
      <option selected="selected">R&B</option>
      <option>Jazz</option>
      <option>Blues</option>
      <option>New Age</option>
      <option>Classical</option>
      <option>Opera</option>
    </select>
  </p>
  <p><input id="btn" type="button" value="How many are selected?" /></p>
</form>
<script>
function howMany(selectObject) {
  var numberSelected = 0;
  for (var i = 0; i < selectObject.options.length; i++) {
    if (selectObject.options[i].selected) {
      numberSelected++;
    }
  }
  return numberSelected;
```

```
}


var btn = document.getElementById("btn");

btn.addEventListener("click", function(){

  alert('Number of options selected: ' + howMany(document.selectForm.musicTypes))

});

</script>
```

sentencia do...while

La sentencia do...while se repite hasta que la condición especificada que se evalúa sea false. Una sentencia do...while se mostrará como sigue:

```
do

  sentencia

while (condicion);
```

sentencia se ejecuta antes de que la condición sea evaluada. Para ejecutar múltiples sentencias, use un bloque de sentencias ({ ... }) para agruparlas. Si condicion es true, la sentencia se ejecuta de nuevo. Al final de cada ejecución, la condición es comprobada. Cuando la condición es falsa, la ejecución se detiene y el control pasa a la sentencia siguiente al do...while.

Ejemplo

En el siguiente ejemplo, el bucle do itera al menos una vez y vuelve a hacerlo mientras i sea menor que 5.

```
do {

  i += 1;

  console.log(i);

} while (i < 5);
```

sentencia while

Una sentencia while ejecuta sus sentencias mientras la condición sea evaluada como verdadera. Una sentencia while tiene el siguiente aspecto:

```
while (condicion)
  sentencia
```

Si la condición cambia a falsa, la sentencia dentro del bucle deja de ejecutarse y el control pasa a la sentencia inmediatamente después del bucle.

La condición se evalúa antes de que la sentencia contenida en el bucle sea ejecutada. Si la condición devuelve verdadero, la sentencia se ejecuta y la condición se comprueba de nuevo. Si la condición es evaluada como falsa, se detiene la ejecución y el control pasa a la sentencia siguiente al while.

Para ejecutar múltiples sentencias, use un bloque de sentencias ({ ... }) para agruparlas.

Ejemplo 1

El siguiente bucle while itera mientras n sea menor que tres:

```
n = 0;
x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

Con cada iteración, el bucle incrementa n y añade ese valor a x. Por consiguiente, x y n toman los siguientes valores:

Después del primer paso: n = 1 y x = 1

Después del segundo paso: n = 2 y x = 3

Después del tercer paso: n = 3 y x = 6

Tras completar el tercer paso, la condición n < 3 ya no es verdadera, por tanto el bucle termina.

Ejemplo 2

Evite los bucles infinitos. Asegúrese de que la condición en un bucle llegue finalmente a ser falsa; de otra forma, el bucle nunca terminará. Las sentencias en el siguiente bucle while se ejecutan sin fin, porque la condición nunca llega a ser falsa:

```
while (true) {
  console.log("Hello, world");
}
```

sentencia label

Un label proporciona una sentencia con un identificador que permite referirse a él desde cualquier lugar de su programa. Por ejemplo, usted puede usar un label para identificar un bucle, y usar las sentencias break o continue para indicar si el programa debe interrumpir un bucle o continuar su ejecución.

La sintaxis de la sentencia label es:

```
label :
   sentencia
```

El valor de label puede ser cualquier identificador JavaScript que no sea una palabra reservada. La sentencia que usted identifique con un label podrá ser cualquier sentencia.

Ejemplo

En este ejemplo, el label markLoop identifica a un bucle while.

```
markLoop:
while (theMark == true) {
   doSomething();
}
```

sentencia break

Use la sentencia break para salir de un bucle, switch, o en conjunto con una sentencia label.

Cuando use break sin un label, finaliza inmediatamente el código encerrado en while, do-while, for, o switch y transfiere el control a la siguiente sentencia.

Cuando usted use break con un label, termina la sentencia especificada por label.

La sintaxis de la sentencia break es la siguiente:

```
break;
```

break label;

La primera forma de la sintaxis finaliza con lo encerrado por el bucle o switch; la segunda finaliza lo especificado por la sentencia label.

Ejemplo 1

El siguiente ejemplo itera a través de los elementos en un array hasta que encuentra que un índice de un elemento cuyo valor es elValor:

```
for (i = 0; i < a.longitud; i++) {
  if (a[i] == elValor) {
    break;
  }
}
```

Ejemplo 2: Breaking a un label

```
var x = 0;
var z = 0
labelCancelLoops: while (true) {
  console.log("Outer loops: " + x);
  x += 1;
  z = 1;
  while (true) {
    console.log("Inner loops: " + z);
    z += 1;
    if (z === 10 && x === 10) {
      break labelCancelLoops;
    } else if (z === 10) {
      break;
    }
  }
```

}

sentencia continue

La sentencia continue puede usarse para reiniciar una sentencia while, do-while, for, o label.

Cuando use continue sin un label, este termina la iteración en curso del código encerrado en una sentencia while, do-while, o for y continúa la ejecución del bucle con la siguiente iteración. A diferencia de la sentencia break, continue no termina completamente la ejecución del bucle. En un bucle while, salta atrás hasta la condición. En un bucle for, salta a la expresionIncremento.

Cuando use continue con un label, esta se aplica al bucle identificado con el label indicado.

La sintaxis de la sentencia continue es la siguiente:

continue;

continue label;

Ejemplo 1

El siguiente ejemplo muestra un bucle while con una sentencia continue que se ejecuta cuando el valor de i es tres. Así, n toma los valores uno, tres, siete, y doce.

```
i = 0;

n = 0;

while (i < 5) {

 i++;

  if (i == 3) {

    continue;

  }

 n += i;

}
```

Ejemplo 2

Una sentencia etiquetada checkiandj contiene una sentencia etiquetada checkj. Si se encuentra continue, el programa termina la iteración en curso de checkj y empieza la siguiente iteración. Cada vez que continue es encontrado, checkj reitera hasta que su condición devuelve false. Y cuando devuelve false, el resto de la sentencia checkiandj es

completada, y checkiandj reitera hasta que su condición devuelve false. Cuando esto ocurre el programa continúa en la siguiente sentencia después de checkiandj.

Si continue tenía una etiqueta checkiandj, el programa continuaría al principio de la sentencia checkiandj.

```
checkiandj:
 while (i < 4) {
  console.log(i);
  i += 1;
  checkj:
   while (j > 4) {
    console.log(j);
    j -= 1;
    if ((j % 2) == 0) {
     continue checkj;
    }
    console.log(j + " is odd.");
   }
   console.log("i = " + i);
   console.log("j = " + j);
 }
```

sentencia for...in

La sentencia for...in itera una variable especificada sobre todas las propiedades enumerables de un objeto. Para cada propiedad distinta, JavaScript ejecuta las sentencias especificadas. Una sentencia for...in será como sigue:

```
for (variable in objeto) {
 sentencias
}
```

Ejemplo

La siguiente función toma como su argumento un objeto y el nombre del objeto. Entonces itera sobre todas las propiedades del objeto y devuelve una cadena que lista los nombres de las propiedades y sus nombres.

```
function volcar_propiedades(obj, obj_nombre) {
  var resultado = "";
  for (var i in obj) {
    resultado += obj_nombre + "." + i + " = " + obj[i] + "<br>";
  }
  resultado += "<hr>";
  return resultado;
}
```

Para un objeto coche con propiedades marca y modelo, resultado sería:

coche.marca = Ford

coche.modelo = Mustang

Arrays

Aunque puede ser tentador usar esto como una forma de iterar sobre elementos Array, la sentencia for...in devolverá el nombre de las propiedades que usted ha definido además de los índices numéricos. En consecuencia es mejor usar un bucle for tradicional con un índice numérico cuando esté iterando sobre arrays, ya que la sentencia for...in itera sobre las propiedades definidas por el usuario además de los elementos del array, si usted modifica el objeto Array, por ejemplo añadiendo propiedades personalizadas o métodos.

sentencia for...of

La sentencia for...of crea un bucle iterando sobre objetos iterables (incluyendo Array, Map, Set, argumentos, objetos etc), invocando una iteración personalizada conectando con sentencias para ser ejecutadas por el valor de cada propiedad distinta.

```
for (variable of objeto) {
  sentencia
```

}

El siguiente ejemplo muestra la diferencia entre un bucle for...of y un bucle for...in. Mientras for...in itera sobre nombres de propiedades, for...of itera sobre valores de propiedades:

```
let arr = [3, 5, 7];

arr.foo = "hello";

for (let i in arr) {

  console.log(i); // logs "0", "1", "2", "foo"

}

for (let i of arr) {

  console.log(i); // logs "3", "5", "7"

}
```

Definición de funciones

Declaraciones de función

La definición de una función (también llamada declaración de función o sentencia de función) consiste de la palabra clave (reservada) function, seguida por:

El nombre de la función (opcional).

Una lista de argumentos para la función, encerrados entre paréntesis y separados por comas (,).

Las sentencias JavaScript que definen la función, encerradas por llaves, { }.

Por ejemplo, el siguiente código define una función simple llamada square:

```
function square(number) {

 return number * number;

}
```

La función square toma un argumento, llamado number. La función consiste de una sentencia que expresa el retorno del argumento de la función (el cual es, number) multiplicado por sí mismo. La sentencia return especifica el valor retornado por la función.

```
return number * number;
```

Los parámetros primitivos (como puede ser un número) son pasados a las funciones por valor; el valor es pasado a la función, si la función cambia el valor del parámetro, este cambio no es reflejado globalmente o en otra llamada a la función.

Si pasa un objeto (p. ej. un valor no primitivo, como un Array o un objeto definido por el usuario) como parámetro, y la función cambia las propiedades del objeto, este cambio sí es visible desde afuera de la función, como se ve en el siguiente ejemplo:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}

var mycar = {make: 'Honda', model: 'Accord', year: 1998},

var x, y;

x = mycar.make;     // x toma el valor "Honda"

myFunc(mycar);

y = mycar.make;     // y toma el valor "Toyota"
                    // (la propiedad make fue cambiada por la función)
```

Nota: Tenga en cuenta que asignar un nuevo objeto al parámetro no tendrá ningún efecto fuera de la función, porque esto está cambiando el valor del parámetro en lugar de una de las propiedades del objeto:

```
function myFunc(theObject) {
  theObject = {make: 'Ford', model: 'Focus', year: 2006};
}

var mycar = {make: 'Honda', model: 'Accord', year: 1998},

var x, y;

x = mycar.make;     // x toma el valor "Honda"

myFunc(mycar);

y = mycar.make;     // y sigue con el valor "Honda"
```

Expresiones de función

Si bien la declaración de la función anterior es sintácticamente una sentencia, las funciones pueden también ser creadas por una expresión de función. Tal función puede ser anónima; no debe tener un nombre. Por ejemplo, la función square podría haber sido definida como:

```
var square = function(number) {return number * number};

var x = square(4) //x obtiene el valor 16
```

Sin embargo, se puede proporcionar un nombre a una expresión de función, y éste puede ser utilizado dentro de la función para referirse a sí misma, o en un depurador para identificar la función en el trazado de pila:

```
var factorial = function fac(n) {return n<2 ? 1 : n*fac(n-1)};

print(factorial(3));
```

Las expresiones de función son convenientes cuando se pasa una función como argumento a otra función. El siguiente ejemplo muestra una función map siendo definida y luego llamada con una expresión de función como primer parámetro:

```
function map(f,a) {

  var result = [], // Crea un nuevo Array

      i;

  for (i = 0; i != a.length; i++)

    result[i] = f(a[i]);

  return result;

}
```

El siguiente código:

```
var multiplicar= function(x) { return x * x * x;} //Expresión de función

map(multiplicar, [0, 1, 2, 5, 10]);
```

retorna [0, 1, 8, 125, 1000].

En JavaScript, una función puede ser definida en base a una condición. Por ejemplo, la siguiente definición de función myFunc es definida sólo si num es igual a 0:

```
var myFunc;
```

```
if (num == 0){
  myFunc = function(theObject) {
    theObject.make = "Toyota"
  }
}
```

Además de definir funciones como se describe aquí, se puede utilizar el constructor Function para crear funciones desde una cadena en tiempo de ejecución, muy al estilo de eval().

Un método, es una función que es propiedad de un objeto.

Llamando funciones

Definir una función no la ejecuta. Definir una función simplemente la nombra y especifica qué hacer cuando la función es llamada. Llamar la función es lo que realmente realiza las acciones especificadas con los parámetros indicados. Por ejemplo, si define la función square, podría llamarla como sigue:

```
square(5);
```

La sentencia anterior llama a la función con el argumento 5. La función ejecuta sus sentencias y retorna el valor 25.

Las funciones deben de estar dentro del ámbito cuando son llamadas, pero la declaración de la función puede ser izada (aparecer por debajo de la llamada en el código), como muestra el siguiente ejemplo:

```
console.log(square(5));
/* ... */
function square(n) { return n*n }
```

El ámbito de la función es la función en la que es declarada o el programa entero si ésta es declarada en el nivel superior.

Nota: Esto sólo funciona cuando se define la función utilizando la sintaxis anterior (p.ej. function funcName(){}). El siguiente código no funcionará. Esto quiere decir que el izado de funciones sólo funciona con una declaración de función y no con una expresión de función

```
console.log(square); // square es creado con un valor inicial indefinido
console.log(square(5)); //TypeError: square no es un función
square = function (n) {
```

```
  return n * n;
}
```

Los argumentos de una función no están limitados a cadenas y números. Pueden enviarse objetos enteros a una función. La función show_props() (definida en Trabajando con objetos) es un ejemplo de una función que toma un objeto como argumento.

Una función puede ser recursiva; es decir, que puede llamarse a sí misma. Por ejemplo, a continuación tenemos una función que calcula el factorial de forma recursiva:

```
function factorial(n){
  if ((n == 0) || (n == 1))
    return 1;
  else
    return (n * factorial(n - 1));
}
```

Entonces, podría calcular los factoriales desde uno hasta cinco de la siguiente manera:

```
var a, b, c, d, e;
a = factorial(1); // a obtiene el valor 1
b = factorial(2); // b obtiene el valor 2
c = factorial(3); // c obtiene el valor 6
d = factorial(4); // d obtiene el valor 24
e = factorial(5); // e obtiene el valor 120
```

Hay otras formas de llamar a las funciones. A menudo hay casos en donde una función necesita ser llamada de forma dinámica, o en donde el número de argumentos de la misma varía; o en la cual, el contexto de la llamada de la función necesita ser ajustada para un objeto específico determinado en el tiempo de ejecución. Resulta que las funciones en sí mismas son objetos, y estos objetos a su vez tienen métodos (ver el objeto Function). Uno de éstos, el método apply(), se puede utilizar para lograr este objetivo.

Ámbito de una Función

Las variables definidas dentro de una función no pueden ser accedidas desde ningún lugar fuera de la función, ya que la variable está definida sólo en el ámbito de la función. Sin embargo, una función puede acceder a todas las variables y funciones definidas dentro del

ámbito en el cual está definida. En otras palabras, una función definida en el ámbito global puede acceder a todas las variables definidas en el ámbito global. Una función definida dentro de otra función, también puede acceder a todas las variables definidas en su función padre y a cualquier otra variable a la que la función padre tenga acceso.

```
// Las siguientes variables están  definidas en el ámbito global

var num1 = 20,

   num2 = 3,

   nombre = "Messi";

// Esta función se define en el ámbito global

function multiplicar() {

 return num1 * num2;

}

multiplicar(); // Retorna 60

// Un ejemplo de función anidada

function obtenerPuntaje () {

 var num1 = 2,

    num2 = 3;


 function agregar() {

  return nombre + " puntaje " + (num1 + num2);

 }

 return agregar();

}

obtenerPuntaje(); // Retorna "Messi puntaje 5"
```

Parámetros de función

A partir de ECMAScript 6, hay dos nuevos tipos de parámetros: Parámetros por defecto y los parámetros REST.

Parámetro por defecto

En JavaScript, los parámetros de funciones están establecidos por defecto a undefined. Sin embargo, en ciertas situaciones puede ser útil establecerlos a un valor suministrado por defecto diferente. Es entonces cuando los parámetros por defecto pueden ayudar.

En el pasado, la estrategia general para establecer los parámetros por defecto era comprobar los valores de éstos en el cuerpo de la función y asignar un valor si estos eran undefined. Si en el siguiente ejemplo ningún valor es suministrado para b durante el llamado, su valor sería undefined cuando se evalúe a*b; y la llamada de multiply retornaría NaN. Sin embargo, esto se evita con la segunda línea en este ejemplo:

```
function multiply(a, b) {

  b = typeof b !== 'undefined' ?  b : 1;

  return a*b;

}

multiply(5); // 5
```

Con los parámetros por defecto, la comprobación en el cuerpo de la función ya no es necesaria. Ahora, puede simplemente poner 1 como valor por defecto para b en la cabeza de la función.

```
function multiply(a, b = 1) {

  return a*b;

}


multiply(5); // 5
```

Para más detalles, puede consultar parámetros por defecto en la referencia.

Parámetros rest

La sintaxis de parámetros rest (en inglés) nos permite representar un número indefinido de argumentos en forma de array. En el ejemplo, usamos los parámetros rest para recolectar los argumentos a partir del segundo y hasta el final. Entonces los multiplicamos por el primero. Este ejemplo está usando una función flecha, la cual es introducida en la siguiente sección.

```
function multiply(multiplier, ...theArgs) {

  return theArgs.map(x => multiplier * x);
```

```
}

var arr = multiply(2, 1, 2, 3);

console.log(arr); // [2, 4, 6]
```

Funciones flecha

Una expresión de función flecha (también conocida como función flecha gruesa o fat arrow function en inglés) tiene una sintaxis más corta comparada con las expresiones de función y no tiene su propio this, arguments, super o new.target. Las funciones flecha son siempre funciones anónimas. Véase también esta entrada en el blog hacks.mozilla.org : "ES6 In Depth: Arrow functions" (en inglés).

Dos factores influenciaron la introducción de las funciones flecha: funciones más cortas y el léxico this.

Funciones más cortas

En algunos patrones funcionales, las funciones más cortas son bienvenidas. Compare:

```
var a = [

  "Hydrogen",

  "Helium",

  "Lithium",

  "Beryllium"

];


var a2 = a.map(function(s){ return s.length });

var a3 = a.map( s => s.length );
```

Sin propio this

Hasta antes de las funciones flecha, cada nueva función definía su propio valor this (un nuevo objeto en el caso de un constructor, no definido en llamada a funciones en modo estricto, el objeto de contexto si la función es llamada como un "método de objeto", etc.). Esto probó ser molesto en un estilo de programación orientada a objetos.

```
function Person() {
```

```
  // El constructor Person() define `this` como el mismo.
  this.age = 0;
  setInterval(function growUp() {
    // En modo no estricto, la función growUp() define `this`
    // como el objeto global, el cual es diferente de el `this`
    // definido por el constructor Person().
    this.age++;
  }, 1000);
}
var p = new Person();
```

En ECMAScript 3/5, esto fue solucionado asignado el valor contenido por this a una variable sobre la que se podía cerrar (o clausurar).

```
function Person() {
  var self = this; // Algunos eligen `that` en lugar de `self`.
                   // Elija uno y sea consistente.
  self.age = 0;
  setInterval(function growUp() {
    // La retrollamada se refiere a la variable `self` de la cual
    // el valor es el objeto esperado.
    self.age++;
  }, 1000);
}
```

Alternativamente, una función ligada podría ser creada de modo que el propio valor de this sería pasado a la función growUp().

Las funciones flecha capturan el valor de this del contexto circundante, por lo que el siguiente código funciona como se espera.

```
function Person(){
```

```
  this.age = 0;

  setInterval(() => {

    this.age++; // |this| se refiere apropiadamente al objeto instancia de Person.

  }, 1000);

}

var p = new Person();
```

Funciones predefinidas

Función global isFinite

La función global isFinite determina si el valor pasado es un número finito, si se necesita el parámetro primero es convertido a número. La sintaxis de isFinite es:

```
isFinite(number);
```

donde number es el valor a evaluar.

Si el argumento es NaN, el infinito positivo Infinity o infinito negativo -Infinity, este método devuelve false, de lo contrario, devuelve true.

El siguiente Codigo de control determina si la entrada del cliente se trata de un número finito.

```
if(isFinite(ClientInput)){

   /* tomar pasos especificos */

}
```

Funcion isNaN

La función isNaN evalúa un argumento para determinar si es "NaN" (no es un número). La sintaxis de isNaN es:

```
isNaN(testValue);
```

donde testValue es el valor que desea evaluar.

Las funciones parseFloat y parseInt regresan "NaN" cuando evalúan un valor que no es un número. isNaN devuelve true si se aprueba "NaN", y false en caso contrario.

El código siguiente evalúa floatValue para determinar si es un número y luego llama a un procedimiento en consecuencia:

```
var floatValue = parseFloat(toFloat);

if (isNaN(floatValue)) {

  notFloat();

} else {

  isFloat();

}
```

Funciones parseInt and parseFloat

Las dos funciones "parse", parseInt y parseFloat, devuelven un valor numérico cuando se les da una cadena como argumento.

La sintaxis de parseFloat es:

```
parseFloat(str);
```

donde parseFloat analiza su argumento, la cadena str, e intenta devolver un número de coma flotante. Si se encuentra un carácter que no sea un signo (+ o -), un número (0-9), un punto decimal o un exponente, a continuación, devuelve el valor hasta ese punto e ignora ese carácter y todos los caracteres siguientes. Si el primer carácter no puede ser convertido a un número, devuelve "NaN" (no un número).

La sintaxis de parseInt es:

```
parseInt(str [, radix]);
```

parseInt analiza su primer argumento, la cadena str, e intenta devolver un entero de la base especificada, indicada por el segundo, el argumento opcional, radix. Por ejemplo, un radix de diez indica a convertir en un número decimal, ocho octal, hexadecimal dieciséis, y así sucesivamente. Para bases superiores a diez, las letras del alfabeto indican numerales mayores de nueve. Por ejemplo, para los números hexadecimales (base 16),de A a F se utilizan.

Si parseInt encuentra un carácter que no es un número en la base especificada, lo ignora y todos los carácteres sucesivos y devuelve el valor entero analizado hasta ese momento. Si el primer carácter no puede ser convertido a un número en la base especificada, devuelve "NaN". La función parseInt trunca la cadena a valores enteros.

Funciones Number y String

Las funciones Number y String le permiten convertir un objeto a un número o una cadena. La sintaxis de estas funciones es:

```
var objRef;

objRef = Number(objRef);

objRef = String(objRef);
```

donde objRef es una referencia de objeto. Number utiliza el método valueOf () del objeto; String utiliza el método toString () del objeto.

El ejemplo siguiente convierte el objeto Date a una cadena legible.

```
var D = new Date(430054663215),
   x;
x = String(D); // x equivale a "Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983"
```

El ejemplo siguiente convierte el objeto String a un objeto Number.

```
var str = "12",
   num;
num = Number(str);
```

Puede comprobarlo. Utilice el método DOM write () y el operador typeof JavaScript.

```
var str = "12",
   num;
document.write(typeof str);
document.write("<br/>");
num = Number(str);
document.write(typeof num);
```

Conceptos básicos del objeto

Un objeto es una colección de datos relacionados y / o funcionalidad (que generalmente consta de varias variables y funciones, que se denominan propiedades y métodos cuando están dentro de objetos).

Al igual que con muchas cosas en JavaScript, la creación de un objeto a menudo comienza con la definición e inicialización de una variable.

```
var persona = {

  nombre: ['Bob', 'Smith'],

  edad: 32,

  genero: 'masculino',

  intereses: ['música', 'esquí'],

  bio: function () {

    alert(this.nombre[0] + '' + this.nombre[1] + ' tiene ' + this.edad + ' años. Le gusta ' +
this.intereses[0] + ' y ' + this.intereses[1] + '.');

  },

  saludo: function() {

    alert('Hola, Soy '+ this.nombre[0] + '. ');

  }

};
```

Después de guardar y actualizar, intente ingresar algunos de los siguientes en su entrada de texto:

```
persona.nombre

persona.nombre[0]

persona.edad

persona.intereses[1]

persona.bio()

persona.saludo()
```

Un objeto se compone de varios miembros, cada uno de los cuales tiene un nombre (por ejemplo, nombre y edad) y un valor (por ejemplo, ['Bob', 'Smith'] y 32). Cada par nombre/valor debe estar separado por una coma, y el nombre y el valor en cada caso están separados por dos puntos. La sintaxis siempre sigue este patrón:

```
var nombreObjeto = {
```

miembro1Nombre: miembro1Valor,

miembro2Nombre: miembro2Valor,

miembro3Nombre: miembro3Valor

}

El valor de un miembro de un objeto puede ser prácticamente cualquier cosa: en nuestro objeto persona tenemos una cadena de texto, un número, dos matrices y dos funciones. Los primeros cuatro elementos son elementos de datos y se denominan propiedades del objeto. Los dos últimos elementos son funciones que le permiten al objeto hacer algo con esos datos, y se les denomina métodos del objeto.

Un objeto como este se conoce como un objeto literal — literalmente hemos escrito el contenido del objeto tal como lo fuimos creando. Esto está en contraste con los objetos instanciados de las clases, que veremos más adelante.

Es muy común crear un objeto utilizando un objeto literal cuando desea transferir una serie de elementos de datos relacionados y estructurados de alguna manera, por ejemplo, enviando una solicitud al servidor para ponerla en una base de datos. Enviar un solo objeto es mucho más eficiente que enviar varios elementos individualmente, y es más fácil de procesar que con una matriz, cuando desea identificar elementos individuales por nombre.

Notación de punto

Arriba, accediste a las propiedades y métodos del objeto usando notación de punto (dot notation). El nombre del objeto (persona) actúa como el espacio de nombre (namespace); al cual se debe ingresar primero para acceder a cualquier elemento encapsulado dentro del objeto. A continuación, escriba un punto y luego el elemento al que desea acceder: puede ser el nombre de una simple propiedad, un elemento de una propiedad de matriz o una llamada a uno de los métodos del objeto, por ejemplo:

persona.edad

persona.intereses[1]

persona.bio()

Espacios de nombres secundarios

Incluso es posible hacer que el valor de un miembro del objeto sea otro objeto. Por ejemplo, intente cambiar el miembro nombre de

nombre: ['Bob', 'Smith'],

a

nombre : {

pila: 'Bob',

apellido: 'Smith'

},

Aquí estamos creando efectivamente un espacio de nombre secundario (sub-namespace). Esto suena complejo, pero en realidad no es así: para acceder a estos elementos solo necesitas un paso adicional que es encadenar con otro punto al final. Prueba estos:

persona.nombre.pila

persona.nombre.apellido

Importante: en este punto, también deberá revisar su código y cambiar cualquier instancia de

nombre[0]

nombre[1]

a

nombre.pila

nombre.apellido

De lo contrario, sus métodos ya no funcionarán.

Notación de corchetes

Hay otra manera de acceder a las propiedades del objeto, usando la notación de corchetes. En lugar de usar estos:

persona.edad

persona.nombre.pila

Puedes usar

persona['edad']

persona['nombre']['pila']

Esto se ve muy similar a cómo se accede a los elementos en una matriz, y básicamente es lo mismo: en lugar de usar un número de índice para seleccionar un elemento, se está utilizando el nombre asociado con el valor de cada miembro. No es de extrañar que los

objetos a veces se denominan arreglos asociativos: mapean cadenas de texto a valores de la misma manera que las arreglos mapean números a valores.

Establecer miembros de objetos

Hasta ahora solo hemos buscado recuperar (u obtener) miembros del objeto: también puede establecer (actualizar) el valor de los miembros del objeto simplemente declarando el miembro que desea establecer (usando la notación de puntos o corchetes), de esta manera:

persona.edad = 45;

persona['nombre']['apellido'] = 'Cratchit';

Intenta ingresar estas líneas y luego vuelve a ver a los miembros para ver cómo han cambiado:

persona.edad

persona['nombre']['apellido']

Establecer miembros no solo es actualizar los valores de las propiedades y métodos existentes; también puedes crear miembros completamente nuevos. Prueba estos:

persona['ojos'] = 'avellana';

persona.despedida = function() { alert("¡Adiós a todos!"); }

Ahora puede probar a los nuevos miembros:

persona['ojos']

person.despedida()

Un aspecto útil de la notación de corchetes es que se puede usar para establecer dinámicamente no solo los valores de los miembros, sino también los nombres de los miembros. Digamos que queríamos que los usuarios puedan almacenar tipos de valores personalizados en sus datos personales, escribiendo el nombre y el valor del miembro en dos entradas de texto. Podríamos obtener esos valores de esta manera:

var nombrePerzonalizado = entradaNombre.value;

var valorPerzonalizado = entradaValor.value;

entonces podríamos agregar este nuevo miembro nombre y valor al objeto persona de esta manera:

persona[nombrePerzonalizado] = valorPerzonalizado;

Para probar esto, intente agregar las siguientes líneas en su código, justo debajo de la llave de cierre del objeto persona:

```
var nombrePerzonalizado = 'altura';

var valorPerzonalizado = '1.75m';

persona[nombrePerzonalizado] = valorPerzonalizado;
```

Ahora intente guardar y actualizar, e ingrese lo siguiente en su entrada de texto:

```
persona.altura
```

Agregar una propiedad a un objeto no es posible con la notación de puntos, que solo puede aceptar un nombre de miembro literal, no un valor variable que apunte a un nombre.

¿Qué es "this" (esté)?

Es posible que hayas notado algo un poco extraño en nuestros métodos. Mira esto, por ejemplo:

```
saludo: function() {

  alert('¡Hola!, Soy '+ this.nombre.pila + '.');

}
```

Probablemente se esté preguntando qué es "this". La palabra clave this se refiere al objeto actual en el que se está escribiendo el código, por lo que en este caso this es equivalente a la persona. Entonces, ¿por qué no escribir persona en su lugar? Como verá en el artículo JavaScript orientado a objetos para principiantes cuando comenzamos a crear constructores, etc., this es muy útil: siempre asegurará que se usen los valores correctos cuando cambie el contexto de un miembro (por ejemplo, dos instancias de objetos persona diferentes) puede tener diferentes nombres, pero querrá usar su propio nombre al decir su saludo).

Vamos a ilustrar lo que queremos decir con un par de objetos persona simplificados:

```
var persona1 = {

  nombre: 'Chris',

  saludo: function() {

    alert('¡Hola!, Soy '+ this.nombre + '.');

  }

}
```

```
var persona2 = {
  nombre: 'Brian',
  saludo: function() {
    alert('¡Hola!, Soy '+ this.nombre + '.');
  }
}
```

En este caso, persona1.saludo() mostrará "¡Hola!, Soy Chris"; persona2.saludo() por otro lado mostrará "¡Hola!, Soy Brian", aunque el código del método es exactamente el mismo en cada caso. Como dijimos antes, this es igual al objeto en el que está el código; esto no es muy útil cuando se escriben objetos literales a mano, pero realmente se vuelve útil cuando se generan objetos dinámicamente (por ejemplo, usando constructores).

Has estado usando objetos todo el tiempo

A medida que has estado repasando estos ejemplos, probablemente hayas pensando que la notación de puntos que has usando es muy familiar. ¡Eso es porque la has estado usando a lo largo del curso! Cada vez que hemos estado trabajando en un ejemplo que utiliza una API de navegador incorporada o un objeto JavaScript, hemos estado usando objetos, porque tales características se crean usando exactamente el mismo tipo de estructuras de objetos que hemos estado viendo aquí, aunque más complejos que nuestros propios ejemplos personalizados.

Entonces cuando usaste métodos de cadenas de texto como:

```
myCadena.split(',');
```

Estabas usando un método disponible en una instancia de la clase String. Cada vez que creas una cadena en tu código, esa cadena se crea automáticamente como una instancia de String, y por lo tanto tiene varios métodos/propiedades comunes disponibles en ella.

Cuando accedió al modelo de objetos del documento (document object model) usando líneas como esta:

```
var miDiv = document.createElement('div');

var miVideo = document.querySelector('video');
```

Estaba usando métodos disponibles en una instancia de la clase de Document. Para cada página web cargada, se crea una instancia de Document, llamada document, que representa la estructura, el contenido y otras características de la página entera, como su URL. De nuevo, esto significa que tiene varios métodos/propiedades comunes disponibles en él.

Lo mismo puede decirse de prácticamente cualquier otro Objeto/API incorporado que haya estado utilizando: Array, Math, etc.

Tenga en cuenta que los Objetos/API incorporados no siempre crean instancias de objetos automáticamente. Como ejemplo, la API de Notificaciones, que permite que los navegadores modernos activen las notificaciones del sistema, requiere que tu crees una instancia de un nuevo objeto para cada notificación que deseas disparar. Intente ingresar lo siguiente en su consola de JavaScript:

```
var miNotificacion = new Notification('¡Hola!');
```

Trabajando con objetos

Visión general sobre los Objetos

Los objetos en JavaScript, como en tantos otros lenguajes de programación, pueden ser comparados con objetos de la vida real. El concepto de Objetos en JavaScript puede entenderse con objetos tangibles de la vida real.

En JavaScript, un objeto es un entidad independiente con propiedades y tipos. Compáralo con una taza, por ejemplo. Una taza es un objeto con propiedades. Una taza tiene un color, un diseño, tiene peso y un material con la que fue hecha, etc. De la misma manera, los objetos de JavaScript pueden tener propiedades, las cuales definen sus características.

Objetos y propiedades

Un objeto de JavaScript tiene propiedades asociadas a él. Una propiedad de un objeto puede ser explicada como una variable adjunta al objeto. Las propiedades de un objeto son basicamente lo mismo que las variables comunes de JavaScript, excepto por el nexo con el objeto. Las propiedades de un objeto definen las características de un objeto; se puede acceder a ellas con una simple notación de puntos:

```
nombreObjeto.nombrePropiedad
```

Como todas las variables de JavaScript, tanto el nombre del objeto (que puede ser una variable normal) y el nombre de propiedad son sensible a mayúsculas y minúsculas. Puedes definir propiedades asignándoles un valor. Por ejemplo, vamos a crear un objeto llamado miAuto y le vamos a asignar propiedades denominadas marca, modelo, y año de la siguiente manera:

```
var miAuto = new Object();

miAuto.marca = "Ford";

miAuto.modelo = "Mustang";
```

```
miAuto.año = 1969;
```

Las propiedades no asignadas de un objeto son undefined (y no null).

```
miAuto.color; // undefined
```

Las propiedades de los objetos en JavaScript también pueden ser accedidas o establecidas mediante la notación de corchetes [ ](Para más detalle ver property accesors). Los objetos son llamados a veces arreglos asociativos, ya que cada propiedad está asociada con un valor de cadena que puede ser utilizada para acceder a ella. Así, por ejemplo, puedes acceder a las propiedades del objeto miAuto de la siguiente manera:

```
miAuto["marca"] = "Ford";

miAuto["modelo"] = "Mustang";

miAuto["año"] = 1969;
```

El nombre de la propiedad de un objeto puede ser cualquier cadena válida de JavaScript, o cualquier cosa que se pueda convertir en una cadena, incluyendo una cadena vacía. Sin embargo, cualquier nombre de propiedad que no sea un identificador válido de JavaScript (por ejemplo, el nombre de alguna propiedad que tenga un espacio o un guión, o comienza con un número) sólo puede ser accedido utilizando la notación de corchetes. Esta notación es muy útil también cuando los nombres de propiedades son determinados dinámicamente (cuando el nombre de la propiedad no se determina hasta su tiempo de ejecución). Ejemplos de ésto se muestran a continuación:

```
var miObjeto = new Object(),
    cadena = "miCadena",
    aleatorio = Math.random(),
    objeto = new Object();

miObjeto.type                = "Sintaxis con punto";
miObjeto["Fecha de creación"] = "Cadena con espacios y acento";
miObjeto[cadena]             = "String value";
miObjeto[aleatorio]          = "Número Aleatorio";
miObjeto[objeto]             = "Objeto";
```

```
miObjeto[""]            = "Incluso una cadena vacía";

console.log(miObjeto);
```

Por favor, advierte que todas las claves con notación en corchetes son convertidas a string a menos que éstas sean Symbols, ya que los nombres de las propiedades (claves) en Javascript pueden ser sólo strings o Symbols (en algún momento, los nombres privados también serán agregados a medida que la propuesta de los campos de la clase progrese, pero no las usarás con el formato []). Por ejemplo, en el código anterior, cuando la clave objeto es añadida a miObjeto, Javascript llamará al método obj.toString(), y usará el resultado de ésta llamada (string) como la nueva clave.

También puedes acceder a las propiedades mediante el uso de un valor de cadena que se almacena en una variable:

```
var nombrePropiedad = "marca";

miAuto[nombrePropiedad] = "Ford";

nombrePropiedad = "modelo";

miAuto[nombrePropiedad] = "Mustang";
```

Puedes utilizar la notación de corchetes con for ... in para iterar sobre todas las propiedades enumerables de un objeto. Para ilustrar cómo funciona esto, la siguiente función muestra las propiedades del objeto cuando pasas como argumentos de la función el objeto y el nombre del objeto:

```
function mostrarPropiedades(objeto, nombreObjeto) {
  var resultado = ``;
  for (var i in objeto) {
    //objeto.hasOwnProperty se usa para filtrar las propiedades del objeto
    if (objeto.hasOwnProperty(i)) {
       resultado += `${nombreObjeto}.${i} = ${objeto[i]}\n`;
    }
  }
  return resultado;
}
```

Por lo tanto, la llamada a la función mostrarPropiedades(miAuto, "miAuto") retornaría lo siguiente:

console.log(mostrarPropiedades(miAuto, "miAuto"));

miAuto.marca = Ford

miAuto.modelo = Mustang

miAuto.annio = 1969

Creando nuevos objetos

JavaScript tiene un número de objetos predefinidos. Además, puedes crear tus propios objetos. En JavaScript 1.2 y versiones posteriores, puedes crear un objeto usando un inicializador de objeto. Como alternativa, puedes crear primero una función constructora y luego crear una instancia de un objeto invocando esa función con el operador new.

El uso de inicializadores de objeto

Además de la creación de objetos utilizando una función constructora, puedes crear objetos utilizando un inicializador de objeto. El uso de los inicializadores de objeto se refiere a veces a como crear objetos con la notación literal. "Inicializador de objeto" es consistente con la terminología utilizada por C++.

La sintaxis para un objeto usando un inicializador de objeto es:

```
var objeto = { propiedad_1  : valor_1,   // propiedad_# puede ser un identificador...
        2:          valor_2,   // o un numero...
        // ...,
        "propiedad n": valor_n }; // o una cadena
```

donde objeto es el nombre del nuevo objeto, cada propiedad_i es un identificador (ya sea un nombre, un número o una cadena literal), y cada valor_i es una expresión cuyo valor es asignado a la propiedad_i. El objeto y la asignación es opcional; si no necesitas hacer referencia a este objeto desde otro lugar, no necesitas asignarlo a una variable. (Ten en cuenta que tal vez necesites envolver el objeto literal entre paréntesis si el objeto aparece donde se espera una declaración, a fin de no confundir el literal con una declaración de bloque.)

Los inicializadores de objetos son expresiones, y cada inicializador de objeto da como resultado un nuevo objeto donde la instrucción de creación sea ejecutada. Los inicializadores de objetos idénticos crean objetos distintos que no se compararán entre sí como iguales. Los

objetos se crean como si se hiciera una llamada new Objet(); esto es, los objetos hechos de expresiones literales de objetos son instancias de Object.

La siguiente declaración crea un objeto y lo asigna a la variable x si y sólo si la expresión cond es true.

if (cond) var x = {saludo: "Hola!"};

El siguiente ejemplo crea miHonda con tres propiedades. Observa que la propiedad motor es también un objeto con sus propias propiedades.

var miHonda = {color: "rojo", ruedas: 4, motor: {cilindros: 4, tamanio: 2.2}};

También puedes utilizar inicializadores de objetos para crear matrices. Consulta array literals.

En JavaScript 1.1 y versiones anteriores, no se puede utilizar inicializadores de objeto. Puedes crear objetos usando sólo sus funciones constructoras o utilizando una función suministrada por algún otro objeto para ese propósito.

Usando una función constructora

Como alternativa, puedes crear un objeto con estos dos pasos:

Definir el tipo de objeto escribiendo una función constructora. Existe una fuerte convención, con buena razón, para utilizar en mayúscula la letra inicial.

Crear una instancia del objeto con el operador new.

Para definir un tipo de objeto, crea una función para el objeto que especifique su nombre, propiedades y métodos. Por ejemplo, supongamos que deseas crear un tipo de objeto para los coches. Quieres llamar Auto a este tipo de objeto, y deseas que tenga las siguientes propiedades: marca, modelo y el año. Para ello, podrías escribir la siguiente función:

```
function Auto(marca, modelo, annio) {
  this.marca = marca;
  this.modelo = modelo;
  this.annio = annio;
}
```

Observa el uso de this para asignar valores a las propiedades del objeto en función de los valores pasados a la función.

Ahora puedes crear un objeto llamado miAuto de la siguiente manera:

var miAuto = new Auto("Eagle", "Talon TSi", 1993);

Esta declaración crea miAuto y le asigna los valores especificados a sus propiedades. Entonces el valor de miAuto.marca es la cadena "Eagle", para miAuto.annio es el número entero 1993, y así sucesivamente.

Puedes crear cualquier número de objetos Auto con las llamadas a new. Por ejemplo,

var kenscar = new Auto("Nissan", "300ZX", 1992);

var vpgscar = new Auto("Mazda", "Miata", 1990);

Un objeto puede tener una propiedad que es en sí mismo otro objeto. Por ejemplo, supongamos que defines un objeto llamado persona de la siguiente manera:

```
function Persona(nombre, edad, sexo) {
  this.nombre = nombre;
  this.edad = edad;
  this.sexo = sexo;
}
```

y luego instancias dos nuevos objetos Persona de la siguiente manera:

var fer = new Persona("Fernando Duclouk", 38, "M");

var alvaro = new Persona("Alvaro Caram", 36, "M");

Entonces, puedes volver a escribir la definición de Auto para incluir una propiedad propietario que tomará el objeto persona, de la siguiente manera:

```
function Auto(marca, modelo, annio, propietario) {
  this.marca = marca;
  this.modelo = modelo;
  this.annio = annio;
  this.propietario = propietario;
}
```

Para crear instancias de los nuevos objetos, utiliza lo siguiente:

var auto1 = new Auto("Eagle", "Talon TSi", 1993, fer);

var auto2 = new Auto("Nissan", "300ZX", 1992, alvaro);

Nota que en lugar de pasar un valor de cadena o entero cuando se crean los nuevos objetos, las declaraciones anteriores pasan al objetos fer y alvaro como argumentos para propietario. Si luego quieres averigüar el nombre del propietario del auto2, puedes acceder a la propiedad de la siguiente manera:

auto2.propietario.nombre

Ten en cuenta que siempre se puede añadir una propiedad a un objeto previamente definido. Por ejemplo, la declaración

auto1.color = "negro";

agrega la propiedad color a auto1, y le asigna el valor "negro". Sin embargo, esto no afecta a ningún otro objeto. Para agregar la nueva propiedad a todos los objetos del mismo tipo, tienes que la propiedad a la definición del tipo de objeto Auto.

Usando el método Object.create

Los objetos también se pueden crear mediante el método Object.create. Este método puede ser muy útil, ya que te permite elegir el prototipo del objeto que deseas crear, sin tener que definir una función constructora.

```
// Propiedades y método de encapsulación para Animal

var Animal = {

 tipo: 'Invertebrados', // valor por defecto de la propiedad

 mostrarTipo: function() {  // Método que mostrará el tipo de Animal

  console.log(this.tipo);

 }

};

// Crear una nuevo objeto de tipo Animal llamado animal1

var animal1 = Object.create(Animal);

animal1.mostrarTipo(); // Salida: Invertebrados

// Crear una nuevo objeto de tipo Animal llamado y asignar 'Pescados' a la propiedad tipo

var fish = Object.create(Animal);

fish.tipo = 'Pescados';

fish.mostrarTipo(); // Salida: Pescados
```

Para obtener información más detallada sobre el método y la forma de usarlo, consulte Object.create().

Herencia

Todos los objetos en JavaScript heredan al menos otro objeto. El objeto del cuál está siendo heredado se conoce como el prototipo, y las propiedades heredadas se pueden encontrar en el objeto prototype del constructor.

Propiedades del objeto indexado

En JavaScript 1.0, puede hacer referencia a una propiedad de un objeto, ya sea por su nombre de la propiedad o por su índice ordinal. En JavaScript 1.1 y posteriores, sin embargo, si inicialmente definimos una propiedad por su nombre, debe referirse siempre a ella por su nombre, y si inicialmente definimos una propiedad por un índice, siempre debe referirse a ella por su índice.

Esta restricción se aplica cuando creas un objeto y sus propiedades con una función constructora (como hicimos antes con el tipo de objeto Auto) y cuando defines propiedades individuales de forma explícita (por ejemplo, miAuto.color = "rojo"). Si inicialmente defines una propiedad de objeto con un índice, como miAuto[5] = "25 mpg", puedes hacer referencia posteriormente a la propiedad sólo como miAuto[5].

La excepción a esta regla son los objetos HTML, como por ejemplo los objetos contenidos en formularios. Siempre se puede hacer referencia a objetos en los formularios, ya sea por su número ordinal (basado en el lugar donde aparecen en el documento) o su nombre (si está definida). Por ejemplo, si la segunda etiqueta <FORM> en un documento tiene un atributo NAME con valor "myForm", puedes hacer referencia al formulario como document.forms [1] o document.forms ["myForm"] o document.myForm.

Definición de las propiedades de un tipo de objeto

Puedes agregar una propiedad a un tipo de objeto definido previamente mediante el uso de la propiedad prototype. Esto define una propiedad que es compartida por todos los objetos del tipo especificado, en lugar de por una sola instancia del objeto. El siguiente código agrega una propiedad color a todos los objetos del tipo Auto, y luego asigna un valor a la propiedad color del objeto auto1.

Auto.prototype.color = null;

auto1.color = "negro";

Para más información, ver la propiedad prototype del objeto Function en la Referencia de JavaScript.

Definiendo los métodos

Un método es una función asociada a un objeto, o, simplemente, un método es una propiedad de un objeto que es una función. Los métodos se definen normalmente como una función, con excepción de que tienen que ser asignados como la propiedad de un objeto. Un ejemplo puede ser:

```
nombreDelObjeto.nombreDelMetodo = nombreDeLaFuncion;

var miObjeto = {

    miMetodo: function(parametros) {

      // ...hacer algo

    }

    //O ESTO TAMBIÉN FUNCIONA

    miOtroMetodo(parametros){

      // ... hacer otra cosa

   }

};
```

donde nombreDelObjeto es un objeto existente, nombreDelMetodo es el nombre que se le va a asignar al método, y nombreDeLaFuncion es el nombre de la función.

Entonces puedes llamar al método en el contexto del objeto de la siguiente manera:

```
object.nombreDelMetodo(parametros);
```

Puedes definir métodos para un tipo de objeto incluyendo una definición del método en la función constructora del objeto. Podrías definir una función que formatée y muestre las propiedades de los objetos del tipo Auto previamente definidos; por ejemplo:

```
function mostrarAutos() {

 var resultado = `Un bonito ${this.marca} ${this.modelo} ${this.annio}`;

 imprimir_con_estilo(resultado);

}
```

donde imprimir_con_estilo es una función para mostrar una línea horizontal y una cadena. Observa el uso de this para referirse al objeto al que pertenece el método.

Puedes hacer de esta función un método de Auto agragando la declaración

this.mostrarAutos = mostrarAutos;

a la definición del objeto. Por lo tanto, la definición completa de Auto ahora se vería así:

```
function Auto(marca, modelo, annio, propietario) {
  this.marca = marca;
  this.modelo = modelo;
  this.annio = annio;
  this.propietario = propietario;
  this.mostrarAutos = mostrarAutos;
}
```

Entonces puedes llamar al método mostrarAutos para cada uno de los objetos de la siguiente manera:

```
auto1.mostrarAuto();
auto2.mostrarAuto();
```

Esto produce el resultado que se muestra en la siguiente figura.

Un bonito Eagle Talon TSi 1993

Un Bonito Nissan 300ZX 1992

Usando this para las referencias a objetos

JavaScript tiene una palabra clave especial, this, que puedes usar dentro de un método para referirte al objeto actual. Por ejemplo, supongamos que tenemos una función llamada validar que valida el valor de la propiedad de un objeto, teniendo en cuenta al objeto y los valores altos y bajos:

```
function validar(objeto, valorbajo, valoralto) {
  if ((objeto.value < valorbajo) || (objeto.value > valoralto))
    alert("Valores no válidos!");
}
```

Entonces, puedes llamar a validar en el controlador de eventos onchange de cada elemento del formulario, usando this para pasarle el elemento, como en el siguiente ejemplo:

<input type="text" name="edad" size="3"

onChange="validar(this, 18, 99)">

En general, this se refiere al objeto de llamada en un método.

Cuando lo combinamos con la propiedad form, this puede referirse al objeto actual del formulario principal. En el siguiente ejemplo, el formulario miForm contiene un objeto de texto y un botón. Cuando el usuario hace clic en el botón, el nombre del formulario se asigna al valor del texto en el formulario. El manejador de eventos del botón onclick utiliza this.form para referirse al formulario principal, myForm.

<form name="miForm">

<label>Nombre del formulario:<input type="text" name="text1" value="Beluga"></label>

<input name="button1" type="button" value="Mostrar Nombre del Formulario"

onclick="this.form.text1.value = this.form.name">

</form>

Definiendo getters y setters

Un getter es un método que obtiene el valor de una propiedad específica. Un setter es un método que establece el valor de una propiedad específica. Puede definir getters y setters en cualquier objeto central predefinido u objeto definido por el usuario que admite la adición de nuevas propiedades. La sintaxis para definir getter y setters utiliza la sintaxis literal de un objeto.

JavaScript 1.8.1 Nota:

Partir de JavaScript 1.8.1, los setters ya no son llamados a la hora de establecer las propiedades en los objetos y matrices inicializadores.

La siguiente sesión de JS shell ilustra como getters y setters podrían trabajar para un objeto o definido por el usuario. El JS shell es una aplicación que permite a los desarrolladores probar código JavaScript en modo por lotes o interactiva. En Firefox puede obtener un JS shell pulsando Ctrl + Shift + K.

En chrome ya lo hemos visto. La consola está en herramientas de desarrollador.

js> var o = {a: 7, get b() {return this.a + 1;}, set c(x) {this.a = x / 2}};

[object Object]

```
js> o.a;

7

js> o.b;

8

js> o.c = 50;

js> o.a;

25
```

Las propiedades del objeto o son:

o.a — un número

o.b — un getter que devuelve o.a más 1

o.c — un setter que establece el valor de o.a a la mitad del valor que este seteado o.c

Ten en cuenta que los nombres de las funciones getters y setters definidos en un literal de objeto utilizando "[gs] et property()" (a diferencia de __define[GS]etter__) no son los nombres de los propios getters, aunque la sintaxis de [gs]et propertyName(){} puede inducir a pensar lo contrario. Para nombrar una función en un getter o setter utilizando la sintaxis "[gs]et property()", define una función nombrada explícitamente mediante programación usando Object.defineProperty (o El retorno heredado de Object.prototype.__defineGetter__).

Esta sesión de shell de JavaScript ilustra como getters y setters pueden extender el prototipo de Date para agregar una propiedad year a todas las instancias de la clase predefinida Date. Utiliza los métodos existentes getFullYear y setFullYear de la clase Date para dar soporte a las propiedades getter y setter de year.

Estas declaraciones definen un getter y setter para la propiedad year:

```
js> var d = Date.prototype;

js> Object.defineProperty(d, "year", {

   get: function() {return this.getFullYear() },

   set: function(y) { this.setFullYear(y) }

});
```

Estas declaraciones utilizan el getter y setter de un objeto Date:

```
js> var now = new Date;
```

```
js> print(now.year);

2000

js> now.year = 2001;

987617605170

js> print(now);

Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001
```

Resumen

En principio, getters y setters pueden ser

definidos utilizando los inicializadores de objeto, o

agregados después a cualquier objeto en cualquier momento utilizando un método getter o setter añadido.

Cuando se define getters y setters utilizando los inicializadores de objeto todo lo que tienes que hacer es anteponer un método getter con get y un método setter con set. Por supuesto, el método getter no debe esperar un parámetro, mientras que el método setter espera exactamente un parámetro (el nuevo valor a definir). Por ejemplo:

```
var o = {
  a: 7,
  get b() { return this.a + 1; },
  set c(x) { this.a = x / 2; }
};
```

Los getters y setters también pueden añadirse a un objeto en cualquier momento después de su creación utilizando el método Object.defineProperties. El primer parámetro de este método es el objeto sobre el que se quiere definir el getter o setter. El segundo parámetro es un objeto cuyo nombre de propiedad son los nombres getter o setter, y cuyos valores de propiedad son objetos para la definición de las funciones getter o setter. Aquí hay un ejemplo que define el mismo getter y setter utilizado en el ejemplo anterior:

```
var o = { a:0 }

Object.defineProperties(o, {
    "b": { get: function () { return this.a + 1; } },
```

```
  "c": [ set: function (x) { this.a = x / 2; } ]

});

o.c = 10        // Ejecuta el setter, que asigna 10/2 (5) a la propiedad 'a'

console.log(o.b)  // Ejecuta el getter, que produce a + 1 o 6
```

¿Cuál de las dos formas elegir? Sepende de tu estilo de programación y de la tarea que te ocupa. Si ya utilizas el inicializador de objeto al definir un prototipo probablemente escojas la primer forma la mayoría de las veces. Esta forma es más compacta y natural. Sin embargo, si necesitas agregar getters y setters más tarde — porque no escribiste el objeto prototipo o particular — entonces la segunda forma es la única forma posible. La segunda forma, probablemente representa mejor la naturaleza dinámica de JavaScript — pero puede hacer que el código sea difícil de leer y entender.

Antes de Firefox 3.0, getter y setter no eran compatibles con los elementos del DOM. Las versiones anteriores de Firefox fallan silenciosamente. Si se necesitan excepciones para aquellos que, al cambiar el prototipo de HTMLElement (HTMLElement.prototype.__define [SG] etter__) y lanzar una excepción es una solución.

Con Firefox 3.0 , definiendo getter o setter en una propiedad ya definida se producirá una excepción. La propiedad debe ser eliminada previamente, lo cual no es el caso para las versiones anteriores de Firefox.

Eliminando propiedades

Puedes eliminar una propiedad no heredada mediante el operador delete. El siguiente código muestra cómo eliminar una propiedad.

```
//Crea un nuevo objeto, miobjeto, con dos propiedades, a y b.

var miobjeto = new Objecto;

miobjeto.a = 5;

miobjeto.b = 12;


//Elimina la propiedad, dejando miobjeto con sólo la propiedad b.

delete myobj.a;

console.log ("a" in myobj) // yields "false"
```

También puedes utilizar delete para eliminar una variable global si la palabra clave var no fue utiliza para declarar la variable:

g = 17;

delete g;

Comparando Objetos

Como sabemos los objetos son de tipo referencia en JavaScript. Dos objetos con las mismas propiedades métodos nunca son iguales. Sólo comparando la misma referencia al objeto consigo mismo dará como resultado true.

// variable de referencia del objeto fruta

var fruta = {nombre: "manzana"};

// variable de referencia del objeto fructificar

var fructificar = {nombre: "manzana"};

fruta == fructificar // retorna false

fruta === fructificar // retorna false

// variable de referencia del objeto fruta

var fruta = {nombre: "manzana"};

// variable de referencia del objeto fructificar

var fructificar = fruta; // asignamos la referencia del objeto fruta a la variable de referencia del objeto fructificar

// aquí fruta y fructificar apuntan al mismo objeto llamado fruta

fruta == fructificar // retorna true

// aquí fruta y fructificar apuntan al mismo objeto llamado fruta

fruta === fructificar // retorna true

Nota: El operador "===" se utiliza para comprobar el valor así como el tipo, ejemplo:

1 === "1" // retorna false

1 == "1" // retorna true

Formularios

Según w3school, el elemento <form> define un formulario para recolectar información del usuario.

Un formulario, a su vez, tiene elementos de formulario como inputs, checkboxes, radio buttons, etc.

Elemento <input>

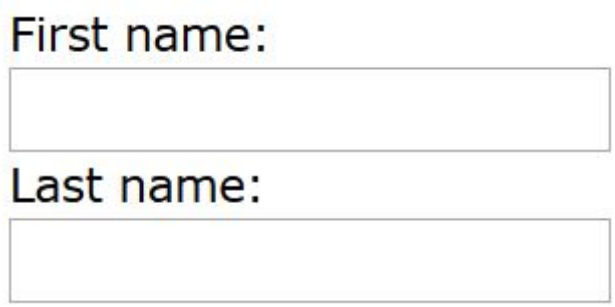
El elemento <input> es el elemento de formulario más importante.

Puede ser mostrado de diferentes formas, dependiendo del atributo type.

Text input

<input type="text"> define un campo de entrada de texto.

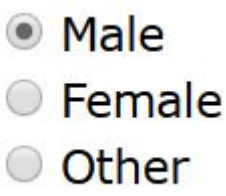
```
<form>
 First name:<br>
 <input type="text" name="firstname"><br>
 Last name:<br>
 <input type="text" name="lastname">
</form>
```



Radio button input

<input type="radio"> define un radio button.

```
<form>
 <input type="radio" name="gender" value="male" checked> Male<br>
 <input type="radio" name="gender" value="female"> Female<br>
 <input type="radio" name="gender" value="other"> Other
</form>
```



Tipos de inputs

Estos pueden ser:

<input type="button">

<input type="checkbox">

<input type="color">

<input type="date">

<input type="datetime-local">

<input type="email">

<input type="file">

<input type="hidden">

<input type="image">

<input type="month">

<input type="number">

<input type="number">

<input type="password">

<input type="radio">

<input type="range">

<input type="reset">

<input type="search">

<input type="submit">

<input type="tel">

<input type="text">

<input type="time">

<input type="url">

<input type="week">

Botón Submit

<input type="submit"> declara un botón para enviar los datos del formulario al servidor.

```
<form action="/action_page.php" method="post">
 First name:<br>
 <input type="text" name="firstname" value="Mickey"><br>
 Last name:<br>
 <input type="text" name="lastname" value="Mouse"><br><br>
 <input type="submit" value="Submit">
</form>
```

Agrupando datos del form

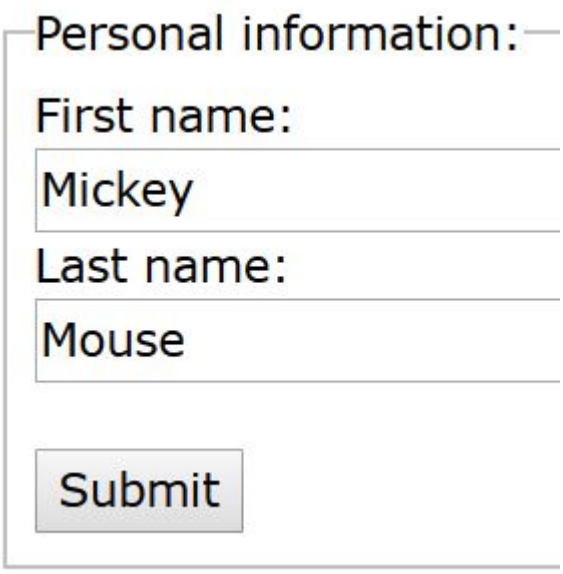
La etiqueta <fieldset> sirve para agrupar elementos relacionados.

El tag <legend> define un epígrafe o título para el elemento <fieldset>

```
<form action="/action_page.php">
 <fieldset>
  <legend>Personal information:</legend>
  First name:<br>
  <input type="text" name="firstname" value="Mickey"><br>
  Last name:<br>
  <input type="text" name="lastname" value="Mouse"><br><br>
  <input type="submit" value="Submit">
```

</fieldset>

</form>



Elemento <select>

El elemento <select> define un combo box.

<!DOCTYPE html>

<html>

<body>

<h2>The select Element</h2>

The select element defines a drop-down list:

<form action="/action_page.php">

<select name="cars">

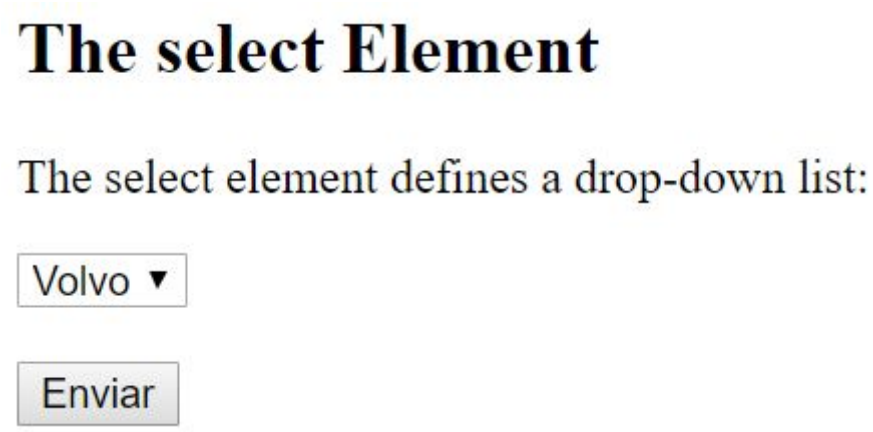
<option value="volvo">Volvo</option>

<option value="saab">Saab</option>

<option value="fiat">Fiat</option>

```
  <option value="audi">Audi</option>
 </select>
 <br><br>
 <input type="submit">
</form>

</body>
</html>
```



Elemento <textarea>

Define un input box de múltiples líneas.

rows especifica la cantidad de líneas visibles y cols el ancho.

```
<textarea name="message" rows="10" cols="30">
The cat was playing in the garden.
</textarea>
```

Elemento <button> y eventos

El elemento <button> define un botón clickeable.

```
<button type="button" onclick="alert('Hello World!')">Click Me!</button>
```

Observar que en la sección de onclick se pone una llamada a función del archivo javascript o una sentencia directamente.

Este atributo representa a un evento que recibe el control, en este caso el botón.

El evento onclick responde ejecutando la sentencia JS como respuesta a presionar el botón.

Con esto concluimos aquí la parte faltante de HTML que nos permitirá crear formularios.

Eventos comunes de HTML

Como vimos en la sección anterior, los eventos responden a acciones sobre la página web.

Los más comunes son los siguientes.

onchange

Responde cuando un elemento a cambiado

onclick

responde a un click del usuario

onmouseover

responde al movimiento del mouse sobre algún elemento.

onmouseout

se dispara cuando se sale con el mouse de un elemento

onkeydown

el usuario presiona una tecla.

onload

el navegador ha terminado de cargar la página.

Validación

A continuación se muestra el código de ejemplo de la validación de un formulario.

El código del formulario anterior es el siguiente:

```
<html>
  <head>
    <title>Revendedores</title>
    <link rel="stylesheet" href="estilo.css"/>
    <script src="app.js"></script>
  </head>
  <body>
    <h1>Nuevo Revendedor</h1>
    <form>
      <table>
        <tr>
          <td>Nombre</td>
          <td><input type="text" id="txtNombre"/></td>
```

```
            </tr>
            <tr>
              <td>Apellido</td>
              <td><input type="text" id="txtApellido"/></td>
            </tr>
            <tr>
              <td>Hijos?</td>
              <td>
                <input type="radio" name="hijos" value="Si" />Si
                <input type="radio" name="hijos" value="No" />No
              </td>
            </tr>
            <tr>
              <td colspan="2"><input type="button" value="Agregar" onclick="agregar();"/></td>
            </tr>
          </table>
        </form>
    </body>
</html>
```

El archivo de validación es el siguiente:

```
function validar() {
    var txtNombre = document.getElementById('txtNombre');
    var txtApellido = document.getElementById('txtApellido');

    if (txtNombre.value === "") {
        alert('Nombre incorrecto');
```

```
        return false;
    }
    if (txtApellido.value === "") {
        alert('Apellido incorrecto');
        return false;
    }

    var radios = document.getElementsByName('hijos');
    var algunoCheckeado = false;
    for (var i=0; i<radios.length; i++) {
        if (radios[i].checked) {
            //alert(radios[i].value);
            algunoCheckeado = true;
            break;
        }
    }
    if (!algunoCheckeado) {
        alert('Tiene hijos?');
        return false;
    }

    return true;
}

function agregar() {
    if (validar()) {
```

```
        //...
        alert('Revendedor agregado');
    } else {
        alert('Corrija los valores');
    }
}
```

La instrucción más importante es la siguiente:

```
document.getElementById('txtNombre');
```

El método aplicado sobre el objeto document, getElementById() retorna el elemento que tiene como id el valor del parámetro.

Document es el objeto que representa a la página web para ser consultado desde javascript. Este documento se encuentra en la estructura DOM (Document Object Model) que es una estructura en forma de árbol con la cual se muestra la página web y es posible manipularla desde HTML.

getElementById() devuelve null si no se encuentra un elemento con ese ID. Por eso es muy importante chequear la sintaxis de los identificadores (id's).

Como última observación diremos que los elementos retornados por getElementById() poseen varios atributos dependiendo del control. Con lo cual, en el ejemplo, podemos extraer el contenido del input para compararlo con la cadena vacía.

Más sobre formularios y validaciones

Validaciones

Según la página https://www.w3schools.com/js/js_validation_api.asp

JavaScript Form Validation

Si un campo (fname) es vacío, esta función dispara una alerta con un mensaje, y retorna falso para prevenir que el formulario sea enviado al servidor.

```
function validateForm() {
  var x = document.forms["myForm"]["fname"].value;
  if (x == "") {
```

```
alert("Name must be filled out");
   return false;
 }
}
```

La función puede ser invocada al enviar el formulario al servidor.

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

Validando números

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Can Validate Input</h2>
<p>Please input a number between 1 and 10:</p>
<input id="numb">
<button type="button" onclick="myFunction()">Submit</button>
<p id="demo"></p>
<script>
function myFunction() {
  var x, text;
  // Get the value of the input field with id="numb"
  x = document.getElementById("numb").value;
  // If x is Not a Number or less than one or greater than 10
  if (isNaN(x) || x < 1 || x > 10) {
```

```
  text = "Input not valid";
 } else {
  text = "Input OK";
 }
 document.getElementById("demo").innerHTML = text;
}
</script>
</body>
</html>
```

Validación automática en HTML

La validación puede hacerse directamente por el navegador.

```
<!DOCTYPE html>
<html>
<body>
<form action="/action_page.php" method="post">
 <input type="text" name="fname" required>
 <input type="submit" value="Submit">
</form>
<p>If you click submit, without filling out the text field,
your browser will display an error message.</p>
</body>
</html>
```

JavaScript Validation API

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

```
<body>

<p>Enter a number and click OK:</p>

<input id="id1" type="number" min="100" max="300" required>

<button onclick="myFunction()">OK</button>

<p>If the number is less than 100 or greater than 300, an error message will be displayed.</p>

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

<script>

function myFunction() {

  var inpObj = document.getElementById("id1");

  if (!inpObj.checkValidity()) {

    document.getElementById("demo").innerHTML = inpObj.validationMessage;

  } else {

    document.getElementById("demo").innerHTML = "Input OK";

  }

}

</script>

</body>

</html>
```

Bibliografía

Mozilla MDN web docs, Colaboradores Mozilla (10/3/2020) Javascript Guide. Recuperado de: https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Introducci%C3%B3n Bajo la licencia: https://creativecommons.org/licenses/by-sa/2.5/ para uso comercial y de libre distribución.

JavaScript: The World's Most Misunderstood Programming Language (10/3/2020) Douglas Crockford. Recuperado de: https://crockford.com/javascript/javascript.html

Douglas Crockford (2008) *JavaScript: The Good Parts,* O'Reilly, Yahoo! Press

Medium, A brief history of JavaScript (1/4/2015) Ben Aston. Consultado el 10/3/2020. Recuperado de: https://medium.com/@benastontweet/lesson-1a-the-history-of-javascript-8c1ce3bffb17

Fin JS (2016-06-17), Brendan Eich - CEO of Brave https://www.youtube.com/watch?v=XOmhtfTrRxc&t=2m5s, Consultado el 10/3/2020

w3schools. Consultado el 10/3/2020. Recuperado de https://www.w3schools.com/js/js_validation_api.asp