Javascript I

Javier Miguel

- CTO & Freelance Developer
- Email: jamg44@gmail.com
- Twitter: @javiermiguelg

Javascript

JS Historia

Eric Elliott
@_ericelliott

Java is to JavaScript as pain is to painting.
#JavaScript

Ver traducción

JS Historia

A principos de los 90 las aplicaciones web no disponían de una forma de comprobar los datos antes de mandarlos al servidor o hacer otras operaciones.

Como las conexiones eran lentas había que esperar mucho para poder ver los errores de validación.

Ver: https://auth0.com/blog/a-brief-history-of-javascript/

JS Historia

En esas fechas un programador de Netscape, **Brendan Eich**, ideó un lenguaje de programación que corría en el navegador.

Lo llamó **Mocha**. Luego se renombró a **LiveScript** en 1995 para incluirse en Netscape Navigator 2.0.

Luego Netscape se alió con Sun y le cambiaron el nombre de nuevo, esta vez a **Javascript**.

JS Historia

Debido al éxito, Microsoft creó **JScript** para su navegador Internet Explorer 3.

En 1997 Netscape decidió estandarizar Javascript y envió la especificación JavaScript 1.1 al **ECMA** (European Computer Manufacturers Association).

ECMA creó el comité TC39. El primer estándar que creó el comité TC39 se denominó ECMA-262, en el que se definió por primera vez el lenguaje **ECMAScript**.

La organización internacional para la estandarización (ISO) adoptó el estándar ECMA-262 a través de su comisión IEC, dando lugar al estándar **ISO/IEC-16262**.

Lenguaje Javascript

Tipos y variables

Tipos y variables

Javascript usa tipado dinámico.

El tipo de una variable es lo que representa su contenido.

```
var numero = 1;
console.log(typeof(numero)); // number
numero = "hola";
console.log(numero); // hola
console.log(typeof(numero)); // string
```

El interprete asigna el tipo en función de su contenido.

Tipos y variables

La palabra reservada **var** declara una variable.

```
var pastillaRoja = 1;
pastillaRoja // 1
```

- No admiten espacios
- No empiezan por números
- Mejor camelCase!!

Tipos y variables

Javascript tiene 5 tipos primitivos (que no son objetos y no tienen métodos):

- boolean (true, false)
- null (es un valor que indica que no hay valor)
- undefined (no definidas o que no han tenido nunca valor)
- number (enteros como *16* o decimales como *3.14*)
- string (con comillas 'simples' o "dobles")

Aparte de los tipos primitivos, todos los demás son objetos (funciones, arrays, etc).

null y undefined

Todas las variables que no han sido definidas (no existen) o han sido definidas sin asignarles un valor son del tipo **undefined**.

null por el contrario es un valor, que indica que la variable está vacía.

```
var dato = null; // de que tipo es dato?
```

dato tiene un valor de tipo **object** (aunque no es un objeto) con valor null.

Tipos y variables

También hay objetos como String, Number, etc, que tienen métodos.

Los tipos primitivos no tienen métodos.

Si intentamos usar un método sobre un tipo primitivo lo que hace el interprete es crear temporalmente su análogo en objeto para ejecutar el método sobre el.

```
"texto de prueba".split(' '); // ["texto", "de", "prueba"]
```

Tipos y variables

Ejemplos con **string** y **String**

```
var objetoString = new String("follow"); // object!, no string literal
typeof(objetoString); // "object"
console.log(objetoString);
String {0: "f", 1: "o", 2: "l", 3: "l", 4: "o", 5: "w", length: 6,
[[PrimitiveValue]]: "follow"}
```

```
var texto = "Follow the white rabbit"; // string literal
texto.substr(11,5); // "white"
typeof(texto); // "string"
```

Tipos y variables - string

No se puede escribir como multi-linea, debe estar en una sola línea (hasta ES5). Se concatena normalmente con + ("ho" + "la")

```
var textoLargo = 'linea1\n' +
     'linea2';
```

o haciendo un array y llamando a `array.join('\n')`.

```
var textoLargo = ['linea1',
    'linea2'
    ].join('\n');
```

Tipos y variables - string

Los caracteres especiales se escapan con \ como por ejemplo

```
"Linea1\nLinea2"
"\tTabulado"
"Comilla \"doble\""
```

Se pueden iterar sus caracteres por índice (variable[2]).

```
console.log(textoLargo[0]); // L
```

Objetos y Arrays

Tipos y variables - Object

Su representación es como un diccionario que contiene otras variables. Tiene propiedades y métodos.

```
var objeto = {
    numero: 1,
    texto: "hola",
    esCero: function(v){ return v == 0;}
}

objeto.esCero(0); // true
objeto.texto; // "hola"
objeto["texto"]; //"hola"
```

Tipos y variables - Object

Iterando las propiedades de un objeto

```
for (var pname in objeto) {
    if (objeto.hasOwnProperty(pname)) {
        console.log("la propiedad " + pname + " existe con el
valor " + objeto[pname]);
    }
}
```

Tipos y variables - Array

Empiezan en 0. Pueden contener elementos de distinto tipo.

```
var array = [
    1, // number
    "hola", // string
    function esCero(v){ return v === 0;}, // función
    {valor:"hola"} // objeto
];

array[1]; // "hola"
array[2](0); //true
```

Tipos y variables - Array

Otras formas de crear un array, con su constructor.

```
var array1 = new Array(longitud);
var array2 = new Array(item0, item1, ...);
```

Tipos y variables - Array

Algunas operaciones con arrays.

```
array.push('al final'); // añade al final, devuelve length
array.unshift('al inicio'); // añade al inicio, devuelve length

array.pop(); // elimina el último y lo devuelve
array.shift(); // elimina el primero y lo devuelve

array.join(','); // devuelve string con elementos unidos por ','
```

Tipos y variables - Array

Los arrays se iteran de varias formas, una de las más normales es con un bucle for:

```
for (var i = 0; i < array.length; i++) {
    var elemento = array[i];
    console.log(elemento);
}

// 1
// hola
// [Function: esCero]
// { valor: 'hola' }
```

Tipos y variables - Array

Los arrays también tienen un método muy util para iterarlos:

```
rows.forEach(function(row) {
  console.log(row);
});
```

Más adelante veremos más en detalle esta sintaxis.

Tipos y variables - Array

Más información sobre arrays:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Hoisting

Tipos y variables - Hoisting

Las declaraciones de variables en JS son "hoisted".
Esto significa que el interprete va a mover al principio de su contexto (función) la declaración, manteniendo la inicialización donde estaba.

```
var pinto = 'my value';

function pinta() {
  console.log('pinto', pinto); // my value
  //var pinto = 'local value'; // esto hará hoisting de pinto y será undefined
};

pinta();
```

Si el código es complejo, declarar las variables al principio

ejemplos/hoisted.js

Tipos y variables - Hoisting

Que valores se escribirán en la consola?

```
var x = 100;
var y = function () {

    if (x === 20) {
        var x = 30;
    }
    return x;
};

console.log( x, y() );
```

Tipos y variables - Hoisting

Que valores se escribirán en la consola?

```
var x = 100;
var y = function () {
    var x; // -> HOISTING -> x ahora es undefined
    if (x === 20) {
        var x = 30;
    }
    return x; -> undefined
};

console.log( x, y() ); // x = 100 | y() = undefined
```

Operadores y control de flujo

Operadores

Operadores

Los operadores básicos son:

+ suma
- resta
* multiplicación
/ división
% resto o módulo
= asignación
==, !=, ===, !==, >, <, >=, <= comparación

Operadores - orden

El orden en que se evalúan es:

1. **. [ ] ( )** Acceso a campos, indización de matrices, llamadas a funciones y **agrupamiento de expresiones**
2. **++ -- - ~ ! delete new typeof void** Operadores unarios, tipos de datos devueltos, creación de objetos, valores no definidos
3. *** / %** Multiplicación, división, división módulo
4. **+ - +** Suma, resta, concatenación de cadenas
5. **<< >> >>>** Desplazamiento bit a bit
6. **< <= > >= instanceof** Menor que, menor o igual que, mayor que, mayor o igual que, instanceof
7. **== != === !==** Igualdad, desigualdad, igualdad estricta y desigualdad estricta
8. **&** AND bit a bit
9. **^** XOR bit a bit
10. **|** OR bit a bit
11. **&&** AND lógico
12. **||** OR lógico
13. **?:** Condicional
14. **= OP=** Asignación, asignación con operación (como += o &=)

Control de flujo

if else, switch, bucles, …

Control de flujo - if else

```
if (condicion) {
  statement_1A;
  statement_1B;
} else {
  statement_2;
}

if (condicion) statement_1;
else statement_2;

// esta segunda forma no se considera buena práctica
```

Control de flujo - if else

Ejemplos de condiciones:

```
var posicionX = 0;
if ((contador < 10 || contador > 20) && // fuera de 10..20 y
    (contador < 80 || contador > 90)) { // fuera de 80..90
  posicionX = contador;
}

if (!posicion > 0) muestraInicio();
else quitaInicio();
```

Control de flujo - if como expresión

Si una variable es verdadera evalúa hacia la primera expresión, sino a la segunda.

```
var nombre = unaVariable ? primeraExpresion : segunda;
```

```
var nombre = car.name ? car.name : '';
```

Control de flujo - switch

```
switch (expresion) {
  case 1:
    statements_1
    [break;]
  case "hola":
    statements_2
    [break;]
    ...
  default:
    statements_def
    [break;]
}
```

Control de flujo - switch

Cuidado con esto!

```
switch (1) {
    case 1 :
        console.log('es 1');
    case 2 :
        console.log('es 2');
    case 3 :
        console.log('es 3');
}
```

Resultado:

```
es 1
es 2
es 3
```

Control de flujo - switch

```
switch (1) {
    case 1 :
        console.log('es 1');
        break;
    case 2 :
        console.log('es 2');
        break;
    case 3 :
        console.log('es 3');
        break;
}
```

Control de flujo - bucles for

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

Control de flujo - bucles do

```
do {
   i += 1;
} while (i < 55);
```

Control de flujo - bucles while

```
while (n < 35) {
  n++;
}
```

Control de flujo - break

`break` rompe un bucle y sale fuera del bloque del bucle.

```
for (var i = 0; i < options.length; i++) {
  var elemento = options[i];
  if (elemento === textoBuscado) {
    encontrado = i;
    break;
  }
}

// después del break sigo por aquí
```

Control de flujo - continue

`continue` salta a la siguiente iteración.

```
while (n < 35) {
  n++;
  if (n < 10) {
    continue;
  }
  procesaRegistro(n);
}
```

Control de flujo - Errores

En Node.js podemos devolver errores de varias formas. Las más comunes son:

1. Lanzando una excepción, para que el código llamante la gestione con try/catch
2. Devolviéndolo en el callback de la llamada

Como regla general usaremos el primero (throw) en código síncrono, y el segundo (callback) en código asíncrono.

Control de flujo - Excepciones

Podemos usar throw para lanzar excepciones. Al lanzarla especificamos la expresión conteniendo el valor que se va a lanzar.

```
openMyFile();
try {
  writeMyFile(theData); //Puede lanzar una excepción
   throw "Error2";
   throw (new Error('Perdición!'));
} catch(e) {
  handleError(e); // Lo gestionamos
   // si lanzamos un Error e.name tendrá 'Error'
   // si lanzamos un Error e.message tendrá 'Perdición!'
} finally {
  closeMyFile(); // Siempre, falle o no
}
```

JSON

JSON

JavaScript Object Notation

- Es un formato para intercambio de datos, derivado de la notación literal de objetos de Javascript.
- Se usa habitualmente para serializar objetos o estructuras de datos.
- Se ha popularizado mucho principalmente como alternativa a XML, por ser más ligero que este.

JSON

Convirtiendo un objeto a JSON:

```
var empleado = {
  nombre: 'Thomas Anderson',
  profesion: 'Developer'
};

JSON.stringify(empleado);

// produce un string
'{"nombre":"Thomas Anderson","profesion":"Developer"}'
```

JSON

Convirtiendo un texto JSON en un objeto:

```
var textoJSON = '{"nombre":"Thomas
Anderson","profesion":"Developer"}';

JSON.parse(textoJSON);

// produce un objeto
Object {nombre: "Thomas Anderson", profesion: "Developer"}
```

Funciones

Funciones

- Se definen con la palabra clave **`function`**
- **Opcionalmente** tienen un nombre
- Pueden recibir un número variable de argumentos y retornar cosas con la palabra clave *`return`*

```
function accion(numero1, numero2) {
  console.log(numero1); // 1
  console.log(numero2); // undefined
  return "smith";
}
accion(1); // "smith"
```

Funciones

- La palabra clave return sale de la función en el punto en el que se encuentre devolviendo como resultado de la función su argumento.

```
function accion(numero1, numero2) {
  console.log(numero1); // 1
  return "smith";
  console.log(numero2); // esta linea no se ejecutará
}
accion(1, 2); // "smith"
```

Funciones

- Se invocan usando el nombre de la función y poniendo `()` detrás.

```
function accion(numero1, numero2) {
  console.log(numero1); // 1
  console.log(numero2); // undefined
  return "smith";
}
accion(1); // "smith"
```

Si no ponemos `()` no estarémos invocando a la función, si no solamente haciendo referencia a la función (p.e. para pasarla como parámetro)

Funciones

En Javascript **las funciones son objetos**, por tanto tienen propiedades y métodos.

Hay varias formas de crear funciones:

- Function declaration
- Function expression
- Function object

Funciones - declaration

- Requieren un nombre.
- Están posicionadas: **solo** a nivel de programa o directamente en el cuerpo de otra función.
- Se hoistean (mueven al inicio del contexto).
- Tienen acceso al contexto de quien las llama.
- Se declaran así:

```
function suma(numero1, numero2) {
  return numero1 + numero2;
}
```

ejemplos/functions.js

Funciones - expression

```
var sumar = function(numero1, numero2) {
  return numero1 + numero2;
}
```

- También tienen acceso al contexto del llamante.
- Como son expresiones, se pueden definir en cualquier sitio donde pueda ir un valor. Por ejemplo, podemos pasarlas como parámetro.
- No hacen 'hoisting', se pueden usar solo después de su definición.
- Pueden tener un nombre, pero solo es visible dentro de ella.

```
var suma = function nombre(numero1, numero2) {
  return numero1 + numero2;
}
```

ejemplos/functions.js

Funciones - objeto

- No tienen acceso al contexto de quien las crea, pero si al contexto global (sin modo estricto).
- El keyword *new* es opcional.

```
var muestraVars = new Function('alert(1); alert(2);');
```

ejemplos/function_constructor.js

Funciones - objeto

Como hemos dicho, no tienen acceso al contexto de quien las crea (foo), pero si al contexto global:

```
x = 10;
function foo() {
  var y = 30;
  var bar = new Function('alert(x); alert(y);');
  bar();
}

// x = 10
// y = undefined
```

ejemplos/function_constructor.js

Métodos

Cuando una función es una **propiedad de un objeto** se le llama método.

```
var calculadora = {
  suma : function(a,b){return a+b},
  resta : function(a,b){return a-b}
}

console.log( calculadora.suma(1,2) ); //3
```

Funciones - instancias

Cuando se usa **new** al invocar una función, se comporta como un constructor de objetos.

```
function Fruta(){
  var nombre, familia;
  this.getNombre = function(){return nombre;};
  this.setNombre = function(value){nombre=value};
  this.getFamilia = function(){return familia;};
  this.setFamilia = function(value){familia=value};
}

var limon = new Fruta();
limon.setNombre("Citrus limon");
limon.setFamilia("Rutaceae");
```

ejemplos/instancias.js

Funciones - instancias

También se pueden hacer instancias con **Object.create()**

```
var Fruta = {
   nombre: null,
   setNombre: function(value){this.nombre=value},
}

// Creamos un objeto cuyo prototype es el objeto Fruta
// y opcionalmente le podemos añadir algo más
var limon = Object.create(Fruta, {
   olor: {value: function() { console.log('huele a limon!')} }
});
```

ejemplos/instancias2.js

Funciones - instancias

Una forma habitual de poner valores por defecto a los argumentos del constructor

```
function Persona(nombre, edad){
  this.nombre = nombre || "Bebe";
  this.edad = edad || 0;
}

var persona = new Persona();
persona.nombre; // "Bebe"
```

Funciones - instancias

El objeto creado puede ser otro para cambiar la visibilidad de sus métodos.

```
function Fruta(){
   var nombre;
   this.getNombre = function(){return nombre;};
   this.setNombre = function(value){nombre=value};
   return {
      setNombre: this.setNombre,
      //getNombre: this.getNombre
   }
}

Fruta();

var limon = new Fruta();
limon.setNombre("Citrus limon");
console.log(limon.getNombre()); // TypeError: undefined is not a function
```

ejemplos/instancias3.js

'use strict';

Modo estricto

El modo Strict habilita más avisos y hace JavaScript un lenguaje un poco más coherente. El modo no estricto se suele llamar “sloppy mode”. Para habilitarlo se puede escribir al principio de un fichero:

```
'use strict';
```

O también se puede habilitar solo para una función:

```
function estoyEnStrictMode() {
    'use strict';
}
```

Modo estricto

Algunos ejemplos de los beneficios del modo estricto:

- **Las variables deben ser declaradas. En *sloppy mode*, una variable mal escrita se crearía global, en *strict* falla.**
- Reglas menos permisivas para los argumentos de funciones, por ejemplo no se pueden repetir.
- Los objetos de argumentos tienen menos propiedades (arguments.callee por ejemplo)
- **En funciones que no son métodos, `this` será undefined.**
- Asignar y borrar propiedades inmutables fallará con una excepción, en *sloppy mode* fallaba silenciosamente.
- No se pueden borrar identificadores si cualificar (delete variable; —> delete this.variable;)
- **eval() es más limpio. Las variables que se definen en el código evaluado no pasan al scope que lo rodea.**
- No más *`with`*

Callbacks

Callbacks

```
function suma( n1, n2, callBack) {
    var resultado = n1 + n2;
    callBack(resultado);
}

suma(1, 5, function(resultado){
    console.log(resultado);
} );

// 6
```

Callbacks

En node.js todos los usos de IO (entrada/salida) deberían ser asíncronos.

Si tras una llamada a una función asíncrona queremos hacer algo, como comprobar su resultado o si hubo errores, le pasaremos **un argumento más**, una expresión de tipo función (callback), para que la invoque cuando termine.

Callbacks

Un ejemplo es cuando usamos setTimeout, que recibe como parámetros:

- Una función con el código que queremos que ejecute tras la espera
- El número de milisegundos que tiene que estar en pausa hasta llamarla.

```
console.log('empiezo');

setTimeout(function() {
  console.log('he terminado');
}, 2000);
```

La función que pasamos a setTimeout **es lo que llamamos *callback***.

Ejercicio

Haciendo una función asíncrona

Ejercicio - función asíncrona

Hacer una función que reciba un texto y tras 2 segundos lo escriba en la consola. La llamaremos ***`esribeTras2Segundos`***

Llamarla dos veces (texto1 y texto2). Deben salir los textos con sus pausas correspondientes.

Al final escribir en la consola "Fin".

Llamada1 - 2 secs. - **texto1** - llamada2 - 2 secs. - **texto2** - **Fin**

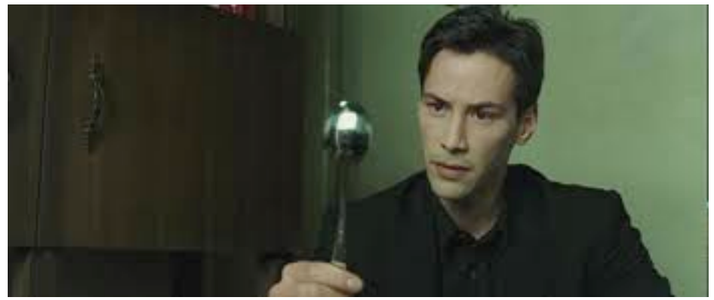
03a_funcion_asincrona

Ejercicio

Leer un fichero - 03b_procesando_un_fichero

Ejemplo

Haciendo un bucle asíncrono

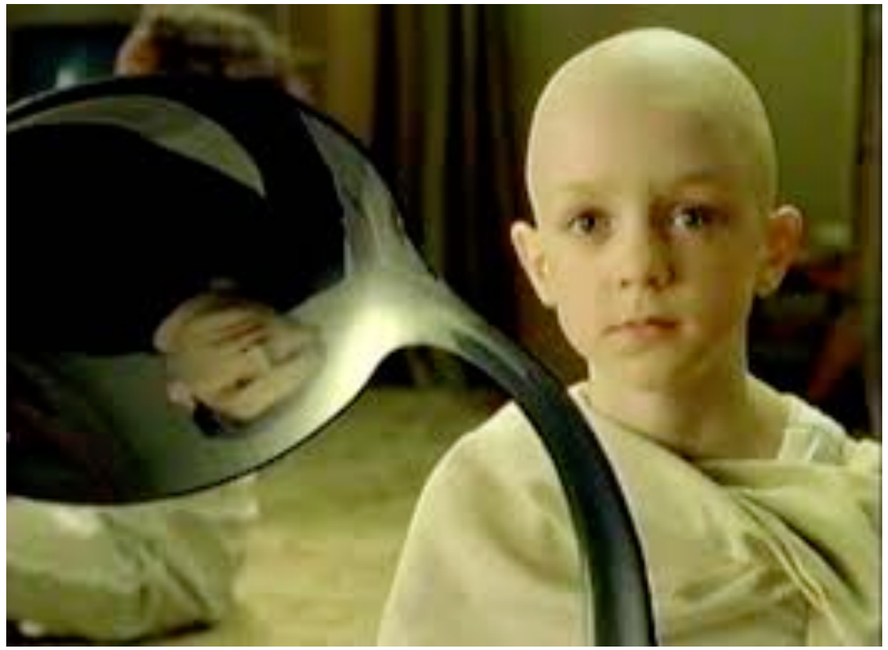


ejemplos/asincrono2_paralelo.js

"No intentes doblar la cuchara,
eso es imposible…
En vez de eso procura comprender la verdad

- ¿Que verdad?

Que no hay cuchara"



Ejemplo

Haciendo un bucle asíncrono en serie

ejemplos/asincrono3_serie.js

Truthy and Falsy

Como se produce

```
var variable = "value";
if(variable) {
    console.log("Soy truthy");
}

variable = 0;
if(variable) {
    console.log("..");
} else {
    console.log("Soy falsy");
}
```

Que son Truthy y Falsy

Los tipos de datos booleanos pueden contener true o false.

```
var a = true;
```

Además de esto, **en javascript todo tiene un valor booleano,** generalmente conocido como **truthy** o **falsy.**

…pero cuidado…

Reglas de truthy / falsy

Los siguientes valores siempre son **falsy**:

- `false`
- `0 (cero)`
- `"" (cadena vacía)`
- `null`
- `undefined`
- `NaN (valor especial de tipo Number que significa Not-a-Number!)`

Todos los demás valores son **truthy**, incluyendo **"0"** (cero entre comillas), **"false"** (false entre comillas), **empty functions**, **empty arrays**, y **empty objects**.

Comparando Falsys

Los valores `false`, `0` (cero), y `""` (cadena vacía) son equivalentes:

```
var c = (false == 0); // true
var d = (false == ""); // true
var e = (0 == ""); // true
```

Los valores `null` y `undefined` no son equivalentes con nada, excepto con ellos mismos:

```
var f = (null == false); // false
var g = (null == null); // true
var h = (undefined == undefined); // true
var i = (undefined == null); // true
```

Y por último, el valor `NaN` no es equivalente con nada, ni siquiera consigo mismo:

```
var j = (NaN == null); // false
var k = (NaN == NaN); // false
```

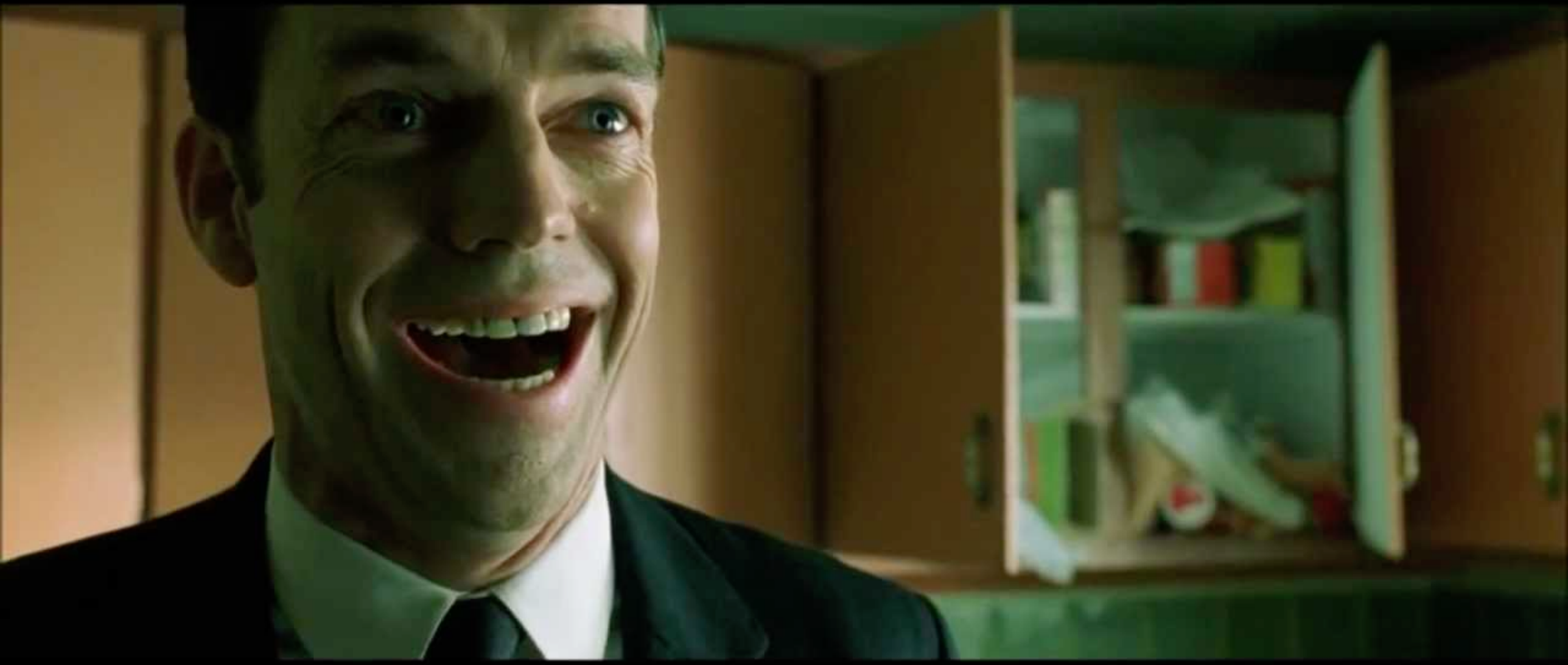
La cosa se complica

```
typeof(NaN) = "number"
/*Tenemos la función isNaN() para solventarlo*/


if ( [] ) { /*se ejecuta*/ }
/*Array es instancia de Object, y existe*/
if ( [] == true ) { /*no se ejecuta*/ }
/*comparamos valores!, [].toString -> "" -> falsy*/



if ( "0" == 0 ) // true (se convierten a números)
if ( "0" ) {console.log('si')} // "si" (se evalúa el string)
```



Como se evita todo esto?

Usamos el **igual estricto** (**===**) y el **distinto estricto** (**!==**) en los casos en que truthy o falsy nos pueden llevar a errores.

Estos operadores nos aseguran que los objetos son comparados **primero por su tipo y luego por su valor**.

```
var melio = (false == 0); // true
var seguro = (false === 0); // false
```

Ejercicio - leer un fichero

Cargar un fichero de texto en una variable.

Para ello usaremos los siguientes ingredientes:

- fs.readFile (https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback)
- path.join (https://nodejs.org/api/path.html#path_path_join_path1_path2)

Ejercicio - parte II

Versión de un módulo - 04_version_modulo

Ejercicio - versión de un módulo

Hacer una función llamada **versionModulo** que consiga la versión de un módulo.

Más ingredientes:

- JSON.parse (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse)
- Domain (https://nodejs.org/api/domain.html#domain_domain_create)

Ejercicio - versión de un módulo

Recibirá un nombre de módulo y un callback. Debe devolver un posible error y la versión del módulo en el callback.

La probaremos con un código como este:

```
versionModulo('chance', function(err, str) {
   if (err) {
      console.error('Hubo un error: ', err);
      return;
   }
   console.log('La version del módulo es:', str);
});
```

Ejercicio - versión de un módulo

Un paso más…

Podemos devolver también el nombre y email del primer desarrollador que lo mantiene?