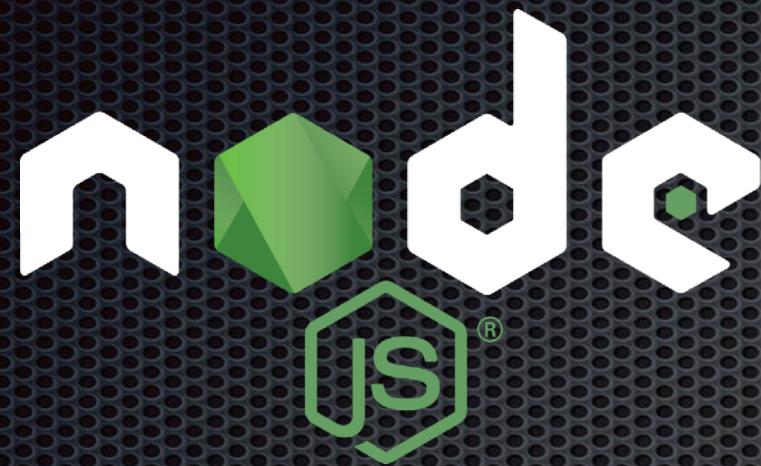


BIENVENIDO



Express.js

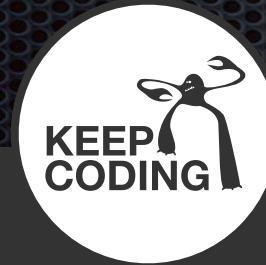


Javier Miguel

@JavierMiguelG

jamg44@gmail.com

CTO & Freelance Developer



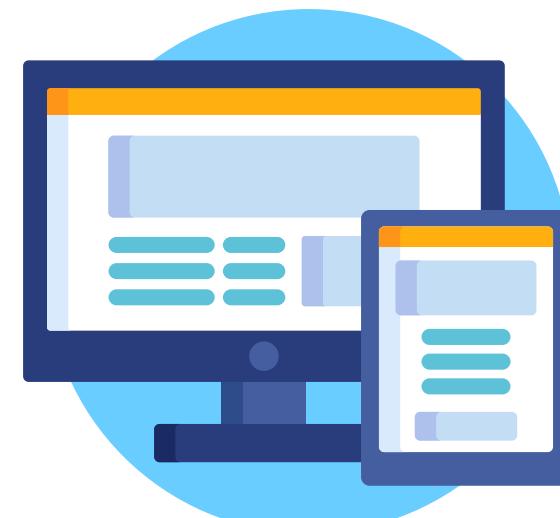


Introducción a la web



Cómo funciona la web

NAVEGADOR



PETICIÓN (Request)

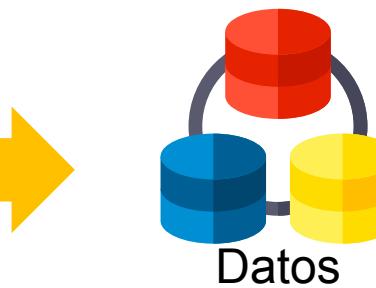
INTERNET

RESPUESTA (Response)

SERVIDOR WEB



Programas



Datos

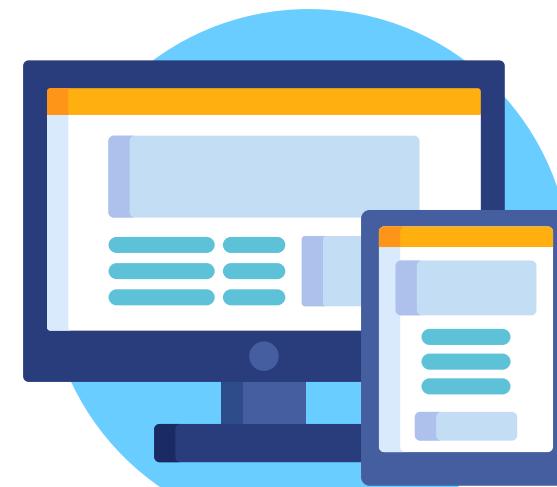


Recursos



■ Esto es FRONT-END

NAVEGADOR



PETICIÓN (Request)

INTERNET

RESPUESTA (Response)

SERVIDOR WEB



Programas



Datos



Recursos



■ Esto es BACK-END

NAVEGADOR



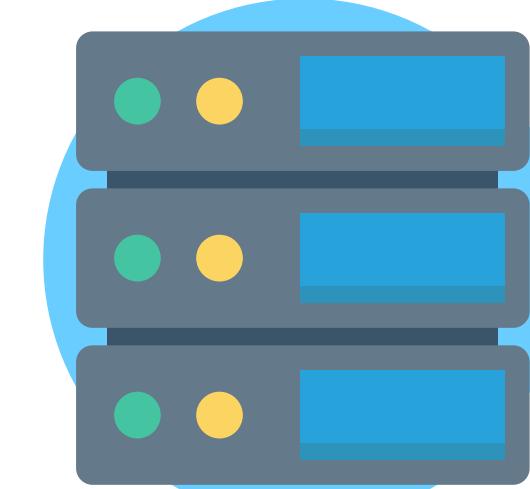
PETICIÓN (Request)

INTERNET

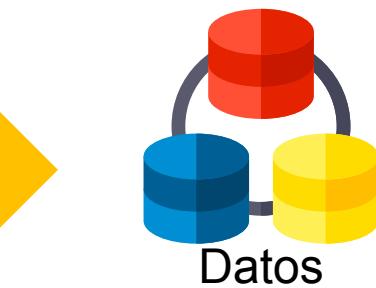
RESPUESTA (Response)



SERVIDOR WEB



Programas



Datos



Recursos



■ Esto es BACK-END

NAVEGADOR



PETICIÓN (Request)

INTERNET

RESPUESTA (Response)

SERVIDOR WEB



Programas



Datos

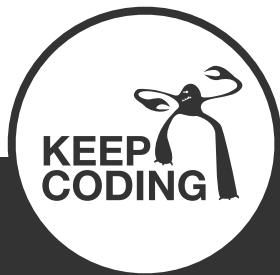


Recursos

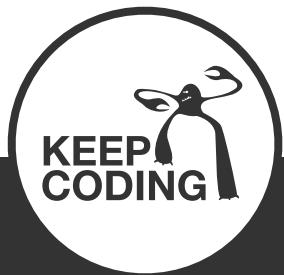




■ Introducción



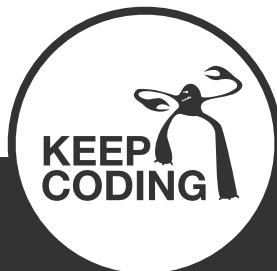
- Es un interprete de Javascript
- Inicialmente diseñado para correr en servidores





Orientado a eventos

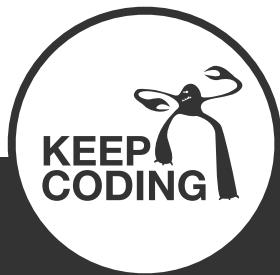
- En la programación secuencial es el programador quien decide el flujo
- En la programación orientada a eventos, el usuario o los programas clientes son quienes definen el flujo





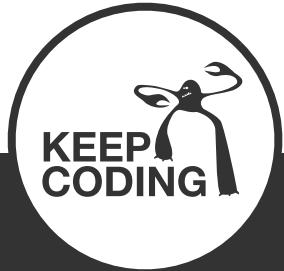
Servidores de aplicaciones

- No necesitamos Tomcat, IIS, etc
- Tampoco necesitamos un servidor web como Apache, nginx, etc
- Nuestra aplicación realiza todas las funciones de un servidor



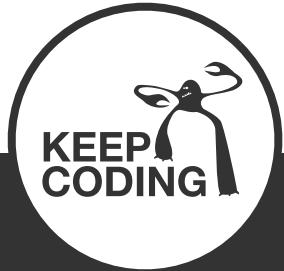
Motor V8

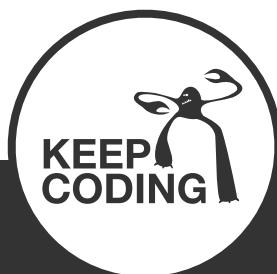
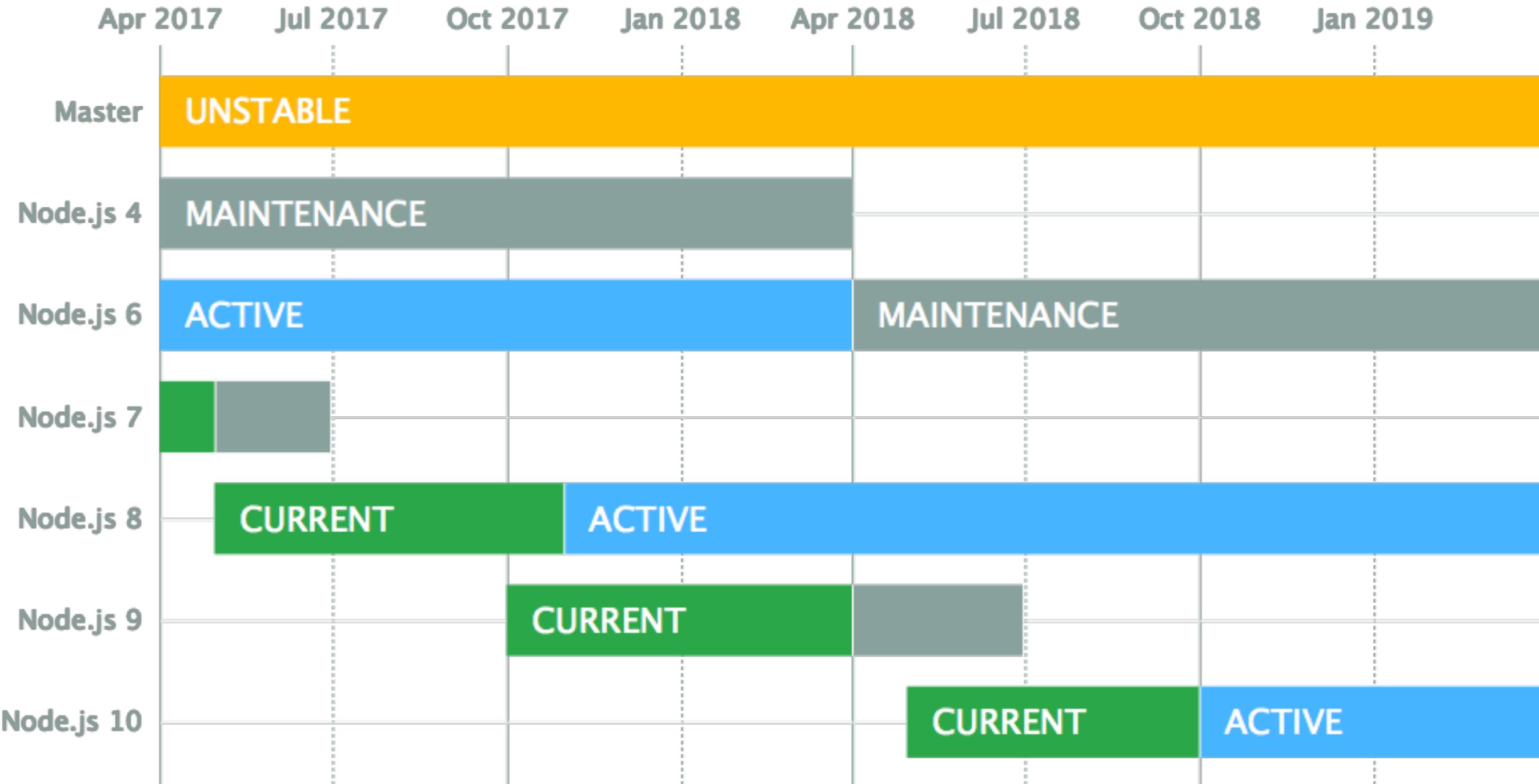
- Motor Javascript creado por Google para Chrome.
- Escrito en C++.
- Multiplataforma (Win, Linux, Mac)





Visiones



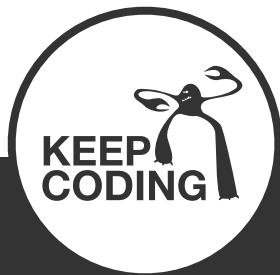




Límite de memoria

Actualmente, por defecto tiene un límite de memoria de 512 MB en sistemas de 32 bits, **1gb en sistemas de 64 bits.**

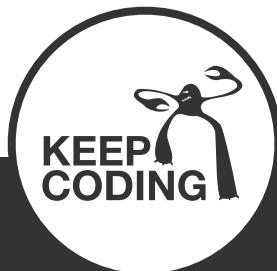
El límite se puede aumentar estableciendo--
max_old_space_size un máximo de ~ 1,024 (~ 1 GiB) (32 bits) y
~ 1,741 (~ 1.7GiB) (64 bits), pero se recomienda dividir el
proceso en varios workers si se llega a los límites.



Ventajas

Node.js tiene grandes ventajas reales en:

- Aplicaciones de red, como APIs, servicios en tiempo real, servidores de comunicaciones, etc
- Aplicaciones cuyos clientes están hechos en Javascript, pues compartimos código y estructuras entre el servidor y el cliente.

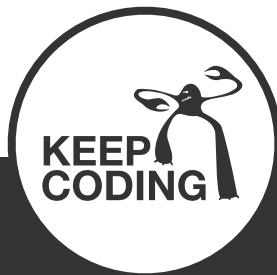


■ ECMAScript 2015 (ES6)

Desde la versión 4.0 de Node.js

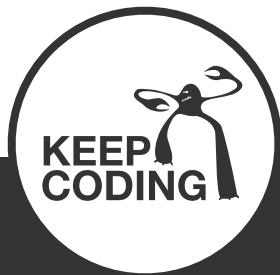
Una de sus principales mejoras es que incluye la librería V8 en la versión v4.5, incluyendo de forma estable muchas características de ES2015 que harán nuestra vida más fácil.

Block scoping, classes, typed arrays, generators, Promises, Symbols, template strings, collections (Map, Set, etc.) and arrow functions.



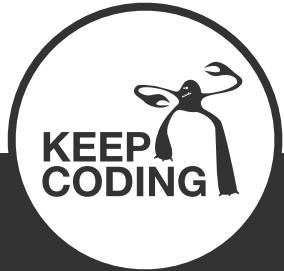
<http://es6-features.org/>

<http://node.green/>





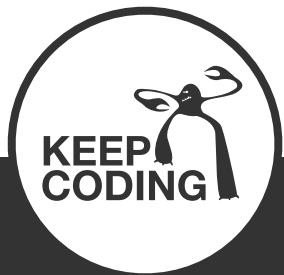
■ Instalando Node.js



■ Distintas formas

Se puede hacer:

- Desde el instalador oficial en nodejs.org
- Con un instalador de paquetes
- Compilando manualmente
- nvm



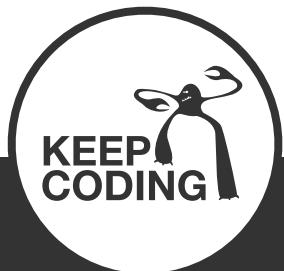
■ Distintas formas - ¿Como decido?

Instalador:

- Más sencillo de instalar

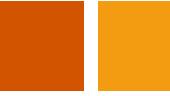
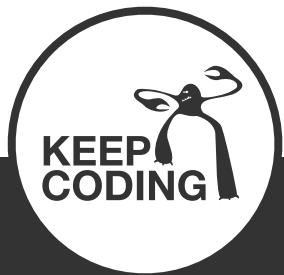
Instalador de paquetes (Homebrew, Chocolatey, apt-get, ...)

- Se instala sin permisos de administrador
- Más fácil de desinstalar





Desde el instalador oficial



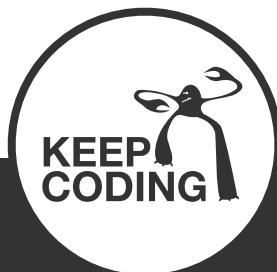
A través del instalador

- Ir a <https://nodejs.org/>
- Install → descargar
- Lanzar .pkg|.exe
- Siguiente, siguiente, ...

(Solo OSX y Windows)



The screenshot shows the official Node.js website at https://nodejs.org/. The header features the Node.js logo and navigation links for HOME, ABOUT, DOWNLOADS, DOCS, FOUNDATION, GET INVOLVED, SECURITY, and NEWS. Below the header, a main text area describes Node.js as a JavaScript runtime built on Chrome's V8 engine, using an event-driven, non-blocking I/O model. It highlights the large npm package ecosystem. A green button labeled "Download for OS X (x64)" is prominently displayed. Other download options like "Other Downloads", "Changelog", and "API Docs" are also visible.



Instalará los ejecutables de node
y npm en la ruta:

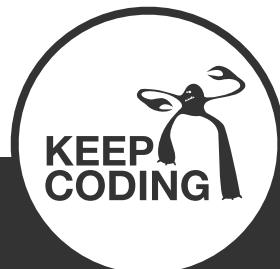
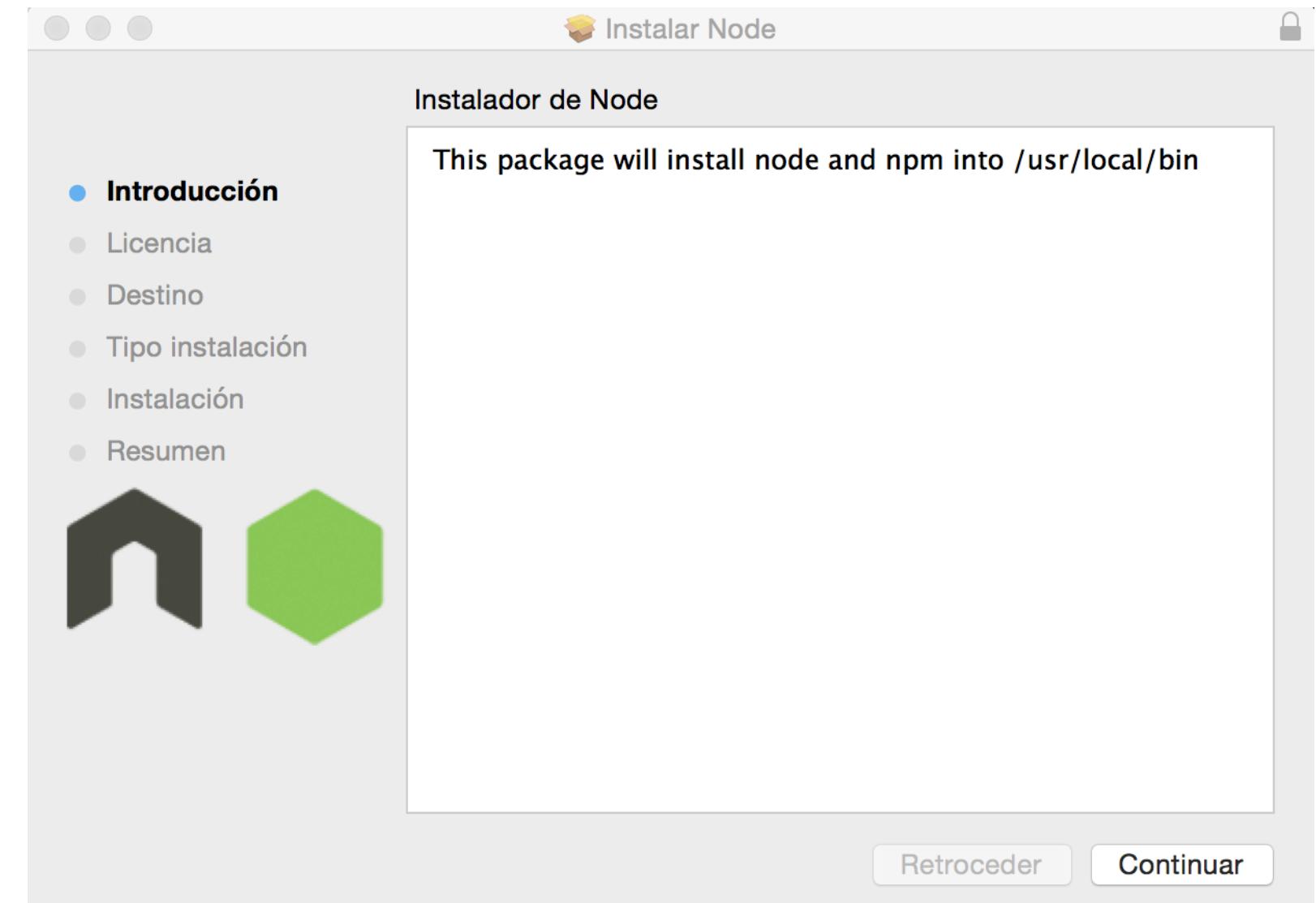
/usr/local/bin

Pondrá la carpeta node_modules
global en:

/usr/local/lib

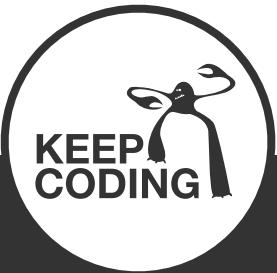
Vínculo dinámico de npm en bin:

```
npm@ -> ../../lib/node_modules/npm/bin/npm-cli.js
```



A partir de aquí ya podremos ejecutar:

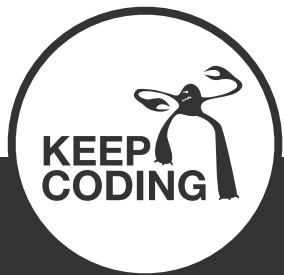
```
~$ node -v  
v4.0.0  
~$ npm -g list --depth=0  
/usr/local/lib  
├── bower@1.5.2  
├── cordova@5.1.1  
├── express-generator@4.13.1  
├── grunt-cli@0.1.13  
├── gulp@3.9.0  
├── ionic@1.6.4  
├── ios-deploy@1.7.0  
├── ios-sim@4.1.1  
├── jasmine@2.3.1  
├── jsdoc@3.3.2  
├── jshint@2.8.0  
├── live-server@0.7.1  
├── nodemon@1.4.1  
├── npm@2.14.2  
└── ssh-tunnel@2.3.23
```



A través del instalador - actualizar

Para actualizar se repite el proceso:

- Descargar el instalador de la versión elegida
- Instalar encima
- Comprobar la nueva versión



A través del instalador - desinstalar

Se ha de hacer manualmente.

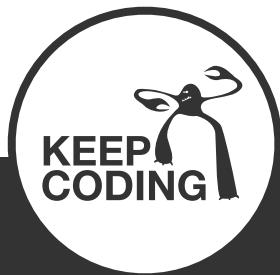
Hay varios scripts que nos ayudan, por ejemplo:

<https://gist.github.com/nicerobot/2697848>

Nota importante:

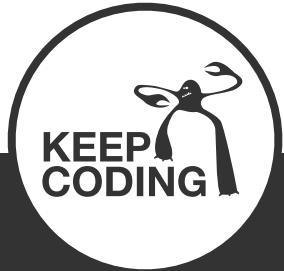
Siempre que usemos un script de un tercero, **revisar su código y entender que es lo que hace!**

(antes de ejecutarlo)





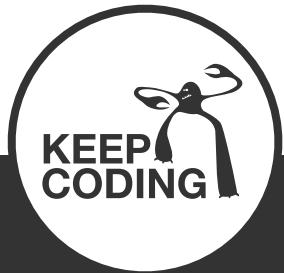
A través de un instalador de paquetes



A través de un instalador de paquetes

Por ejemplo:

- Chocolatey (Windows)
- Homebrew (macOS)
- En Linux: apt en Ubuntu, pacman en arch, etc



A través de un instalador de paquetes

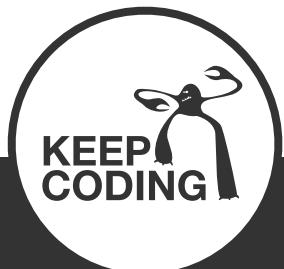
Instalación:

1. Abre el Terminal
2. Escribe:

```
brew install node
```

Tras la instalación comprueba las versiones:

```
~$ node -v  
v0.12.7  
  
~$ npm -g list --depth=0  
/usr/local/lib  
└── bower@1.4.1  
    ├── cordova@5.1.1  
    ├── grunt-cli@0.1.13  
    ├── gulp@3.9.0  
    ├── ionic@1.6.4  
    ├── ios-deploy@1.7.0  
    ├── ios-sim@4.1.1  
    ├── jasmine@2.3.1  
    ├── live-server@0.7.1  
    ├── nodemon@1.4.1  
    └── npm@2.11.3  
        └── ssh-tunnel@2.3.23
```



A través de un instalador de paquetes - actualizar

Actualización:

1. Abre el Terminal

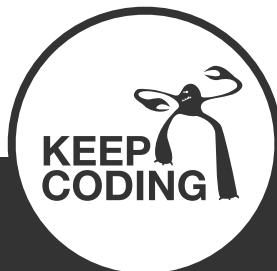
2. Escribe:

`brew update`

3. Luego:

`brew upgrade node`

Tras la actualización comprueba las versiones.

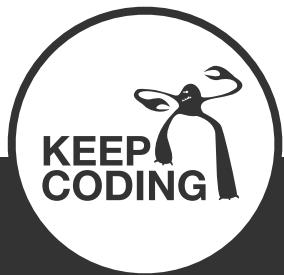


A través de un instalador de paquetes - desinstalar

Desinstalación:

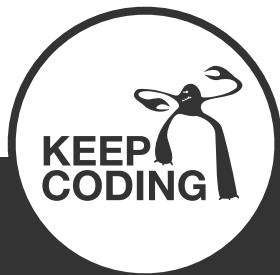
1. Abre el Terminal
2. Escribe:

```
brew uninstall node
```





■ Version managers



Instalar

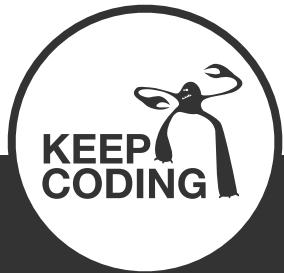
- En linux/mac - <https://github.com/nvm-sh/nvm>
- En windows - <https://github.com/coreybutler/nvm-windows>

```
nvm list
```

```
nvm install <version>
```

```
nvm use <version>
```

```
nvm use system (en linux)
```



nvm - cambio automático

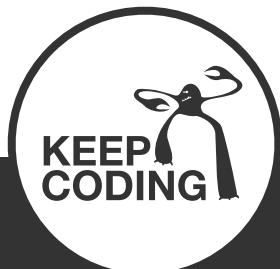
<https://github.com/nvm-sh/nvm#nvmrc>

```
echo "12.13.1" > .nvmrc
```

```
#posteriormente, tras entrar a la carpeta  
nvm use
```

```
#tras salir de la carpeta para volver a la versión default  
nvm use default
```

Tip adicional para usuarios de linux/mac (f en el chat para windows)
<https://github.com/nvm-sh/nvm#deeper-shell-integration>



■ Version Managers Alternativos

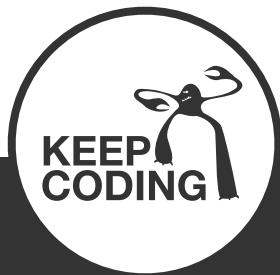
- n es una alternativa de nvm de larga data que logra lo mismo con comandos ligeramente diferentes y se instala a través de npm en lugar de un script bash.
- fnm es un administrador de versiones más reciente, que afirma ser mucho más rápido que nvm. (También usa Azure Pipelines).
- Volta es un nuevo administrador de versiones del equipo de LinkedIn que afirma una velocidad mejorada y soporte multiplataforma.
- asdf-vm es una única CLI para varios idiomas, como gvm, nvm, rbenv y pyenv (y más), todo en uno.
- nvs (Node Version Switcher) es una alternativa a nvm multiplataforma con la capacidad de integrarse con VS Code.





■ Un servidor básico

Ejercicio

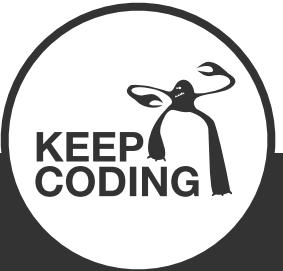


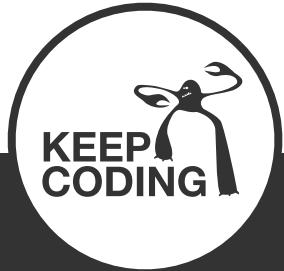
Ejecutar

```
$ node index.js
```

Con nodemon:

```
$ npm install nodemon -g  
$ nodemon
```



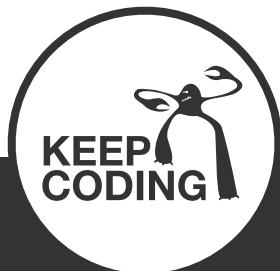
 NPM

Node Package Manager es un gestor de paquetes que nos ayuda a gestionar las dependencias de nuestro proyecto.

Entre otras cosas nos permite:

- Instalar librerías o programas de terceros
- Eliminarlas
- Mantenerlas actualizadas

Generalmente se instala conjuntamente con Node.js de forma automática.



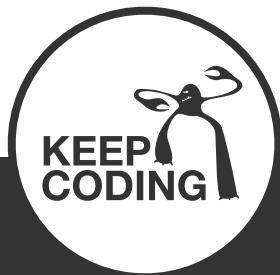
NPM - package.json

Se apoya en un fichero llamado package.json para guardar el estado de las librerías.

```
npm init
```

Crea este fichero.

Documentación en <https://docs.npmjs.com/files/package.json>



NPM - package.json

\$ npm init

This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (myapp)

version: (1.0.0)

description: Esta es la descripción

entry point: (index.js) index.js

test command:

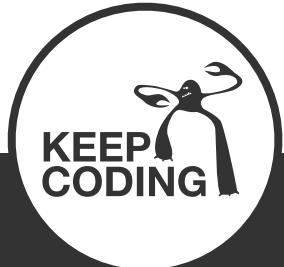
git repository:

keywords:

author: Javier Miguel

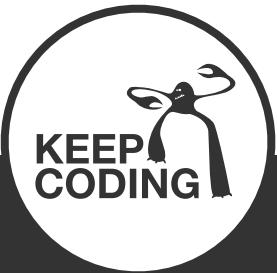
license: (ISC)

About to write to \Users\javi\www\cursonode\keepcoding\myapp\package.json:



NPM - package.json

```
{  
  "name": "myapp",  
  "version": "1.0.0",  
  "description": "Esta es la descripción",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Javier Miguel",  
  "license": "ISC"  
}
```

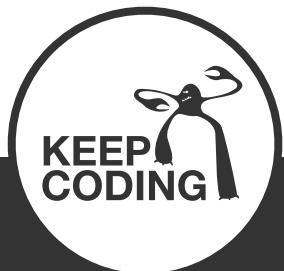


NPM - package.json

Instalar una librería (como por ejemplo chance)

```
npm install chance --save
```

```
{  
  "name": "myapp",  
  "version": "1.0.0",  
  "description": "Esta es la descripción",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Javier Miguel",  
  "license": "ISC"  
  "dependencies    "chance": "^0.8.0"  
  }  
}
```



NPM - global o local

Instalación local, en la carpeta del proyecto

```
npm install <paquete> [--save]
```

Instalación global (en */usr/local/lib/node_modules*)

```
npm install -g <paquete>
```

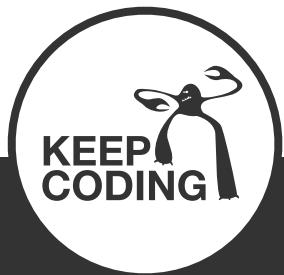
Si el paquete tiene ejecutables hará un vínculo a ellos en */usr/local/bin*





■ Instalando un módulo

Ejercicio

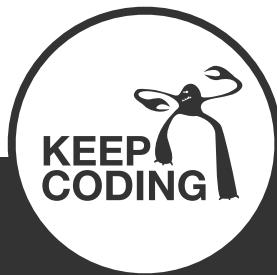


■ Instalando un módulo

Instalar el módulo chance (<https://github.com/victorquinn/chancejs>) y usar su generador de nombres en el servidor básico.

[npm init]

npm install chance [--save]

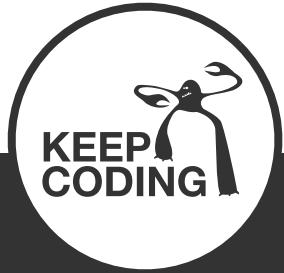




■ JS intermedio & avanzado



■ 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

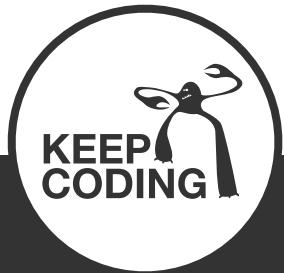
ejemplos/hoisted.js



■ Tipos y variables - Hoisting

Que valores se escribirán en la consola?

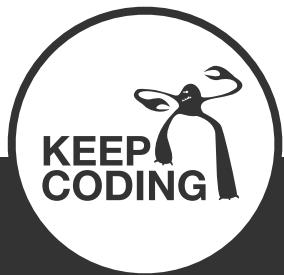
```
var x = 100;  
var y = function () {  
  
    if (x === 20) {  
        var x = 30;  
    }  
    return x;  
};  
  
console.log( x, y() );
```



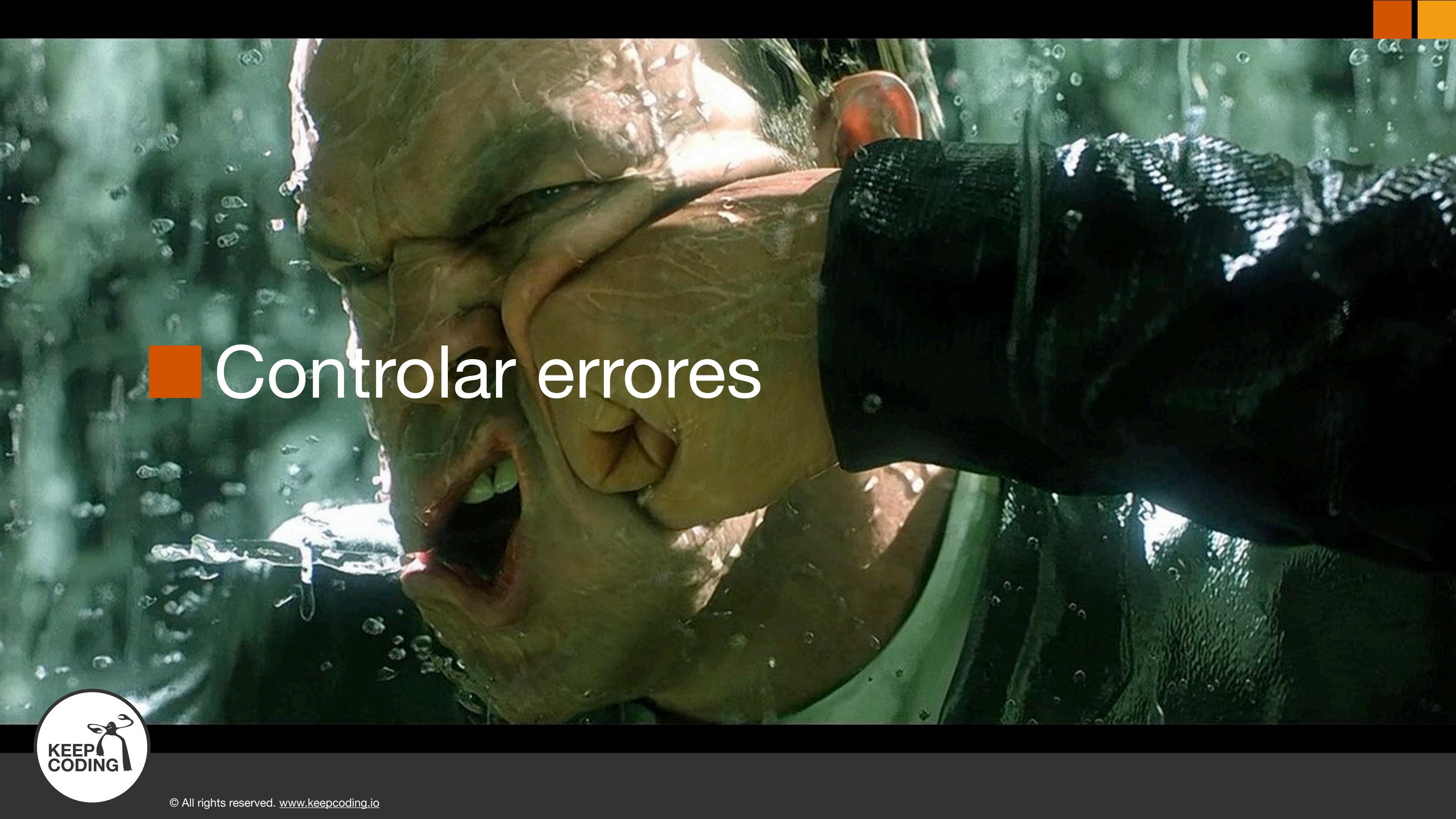
■ Tipos y variables - Hoisting

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
```



KEEP
CODING



■ Controlar errores

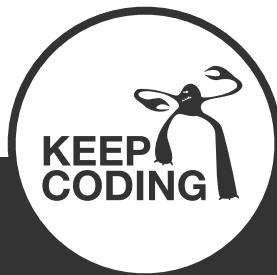


■ Control de flujo - Errores

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

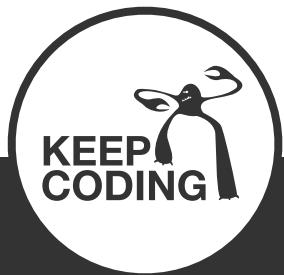
1. Lanzando una excepción, para que el código llamante la gestione con try/catch
2. Devolviéndolo en el callback de la llamada

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



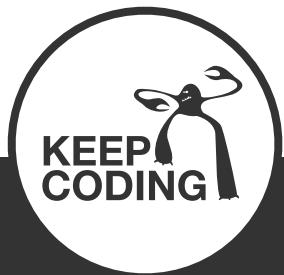
■ Control de flujo - Síncrono

```
try {  
  const objeto = JSON.parse(datos);  
} catch (err) {  
  console.log('No se pudo leer el JSON');  
}
```



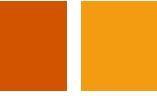
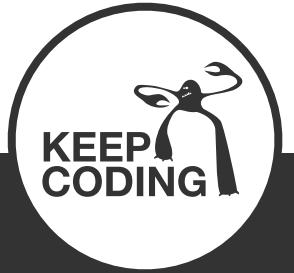
■ Control de flujo - Asíncrono

```
function getUser(id, callback) {
  readFile('usuario.json', 'utf8', function(err, data) {
    if (err) {
      callback(err);
    }
  });
}
```



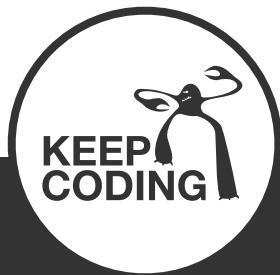


JSON



JavaScript Object Notation

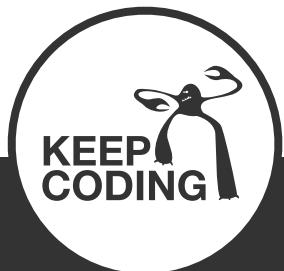
- Es un formato para intercambio de datos, derivado de la notación literal de objetos de Javascript.
- Se usa habitualmente para serializar objetos o estructuras de datos.
- Se ha popularizado mucho principalmente como alternativa a XML, por ser más ligero que este.



■ JSON

Convirtiendo un objeto a JSON:

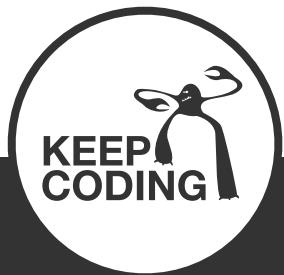
```
var empleado = {  
    nombre: 'Thomas Anderson',  
    profesion: 'Developer'  
};  
  
JSON.stringify(empleado);  
  
// produce un string  
'{"nombre": "Thomas Anderson", "profesion": "Developer"}'
```



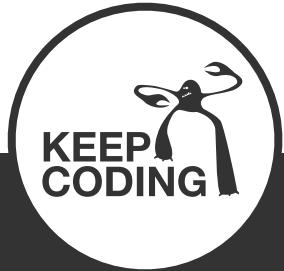
■ JSON

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"}
```



'use strict';



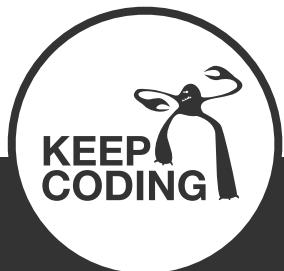
■ Modo estricto

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';  
}
```



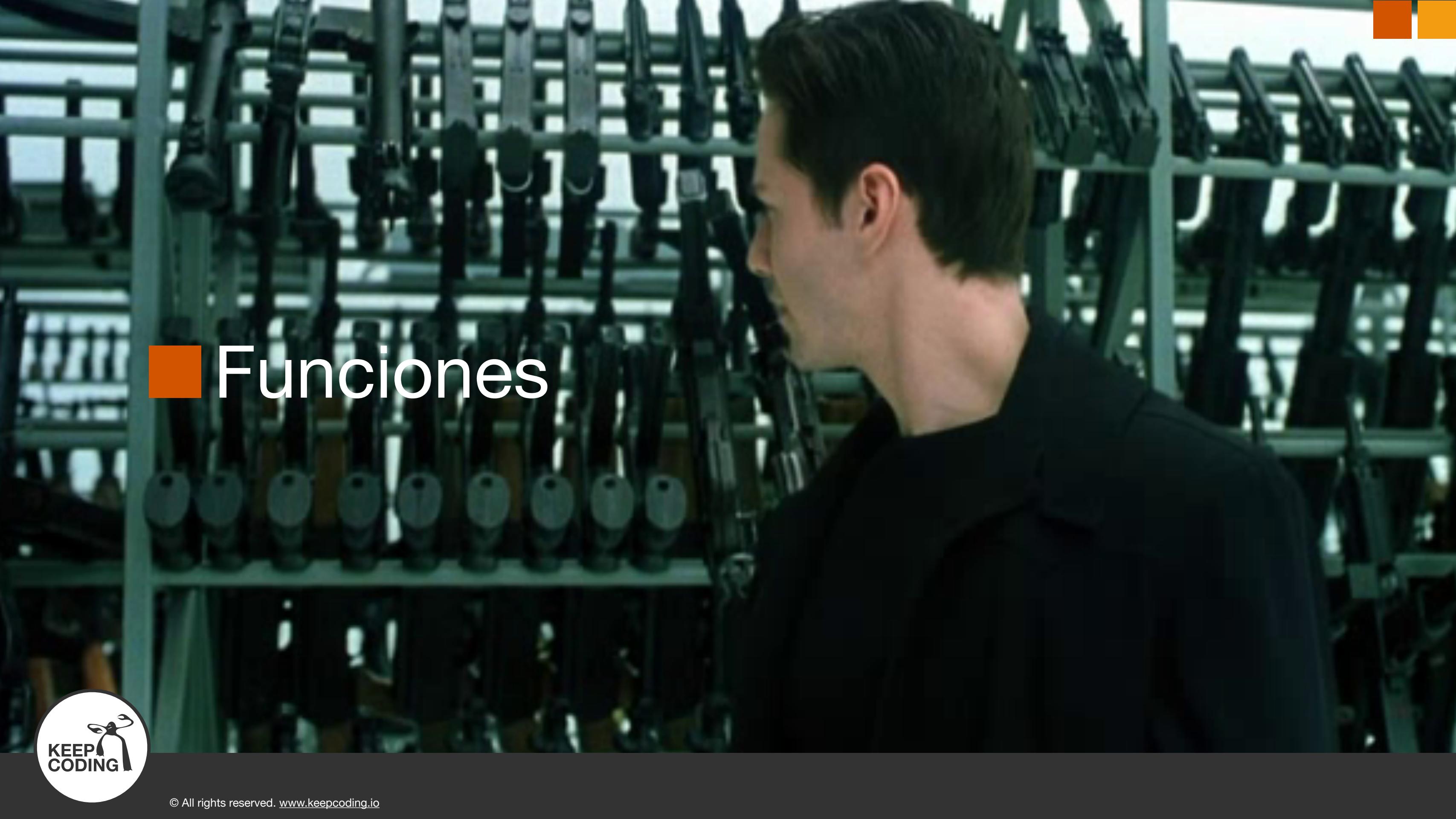
■ Modo estricto

Algunos ejemplos de los beneficios del modo estricto:

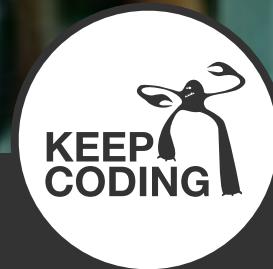
- **Las variables deben ser declaradas.** En *slippery mode*, una variable mal escrita se crearía global, en *strict* falla.
- Reglas menos permisivas para los parámetros de funciones, por ejemplo no se pueden repetir.
- Los objetos de argumentos tienen menos propiedades (arguments.callee por ejemplo)
- **En funciones que no son métodos, this será undefined.**
- Asignar y borrar propiedades inmutables fallará con una excepción, en *slippery 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*





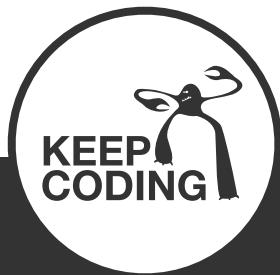
■ Funciones



■ Funciones

Las funciones son objetos

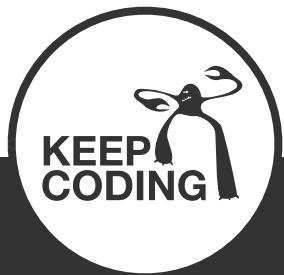
Por tanto también tienen propiedades y métodos



■ Funciones - declaración

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

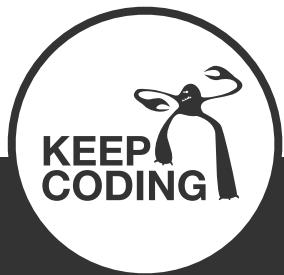
- Requieren un nombre
- Solo a nivel de programa o directamente en el cuerpo de otra función
- Hacen hoisting



■ Funciones - expression

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

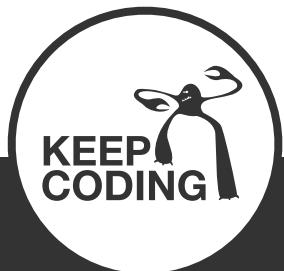
- Como son expresiones, se pueden definir en cualquier sitio donde pueda ir un valor. Por ejemplo, podemos pasarlas como parámetro
- No hacen 'hoisting', se pueden usar solo después de su definición
- Pueden tener un nombre, pero solo sería visible dentro de su cuerpo



Métodos

```
var calculadora = {  
    suma : function(a, b) { return a + b; },  
    mult : function(a, b) { return a * b; }  
}
```

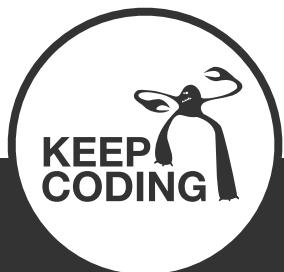
Cuando una función es una propiedad de un objeto se le llama **método**.



■ Funciones - instancias

```
function Fruta() {  
  var nombre, familia;  
  this.getNombre = function() { return nombre; };  
  this.setNombre = function(value) { nombre = value; };  
}  
  
var limon = new Fruta();  
limon.setNombre("Citrus limon");
```

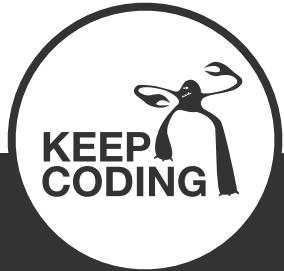
Cuando se usa **new** al invocar una función, se comporta como un constructor de objetos.



ejemplos/instancias.js



Callbacks



■ 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');

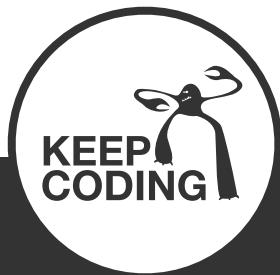
setTimeout(function() {
  console.log('he terminado');
}, 2000);
```



■ Callbacks

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

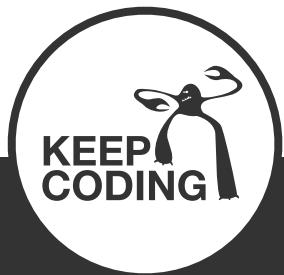
Si tras una llamada a una función asíncrona queremos hacer algo, como comprobar su resultado o si hubo errores, le pasaremos **un argumento más**, una expresión de tipo función (callback), para que la invoque cuando termine.



■ Callbacks

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

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



Callbacks

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

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



Pasamos una función como tercer parámetro.
Le decimos: "cuando termines haz esto"



Callbacks

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

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

Esa función la recibimos y la llamamos callback

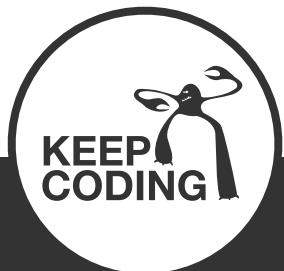


Callbacks

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

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

Cuando hemos terminado llamamos a la
función de callback



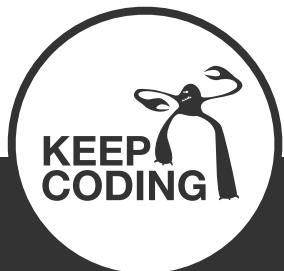
Callbacks

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

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



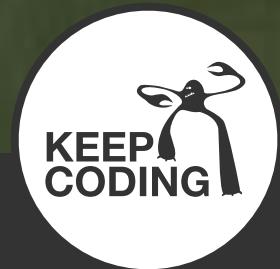
El cuerpo del callback recibe el resultado y lo utiliza





Ejercicio

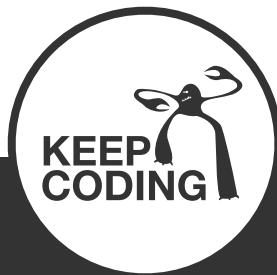
Haciendo una función asíncrona



Ejercicio - función asíncrona

Hacer una función que reciba un texto y tras 2 segundos lo escriba en la consola.

La llamaremos *escribeTras2Segundos*

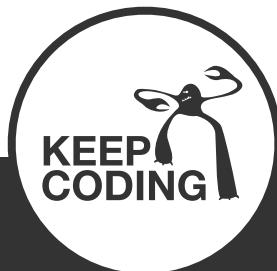


Ejercicio - función asíncrona

Lamarla 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





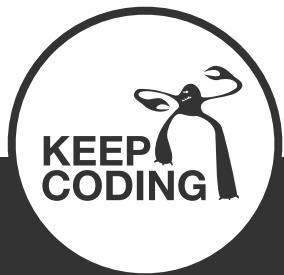
Ejemplo

Haciendo un bucle asíncrono



Ejercicio - bucle asíncrono

Repitamos la llamada a `escribeTras2Segundos` varias veces



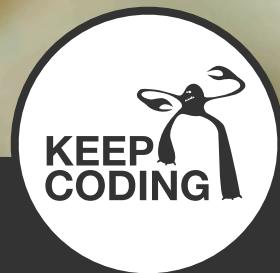


"No intentes doblar la cuchara,
eso es *impossible*..."

En vez de eso procura comprender la verdad

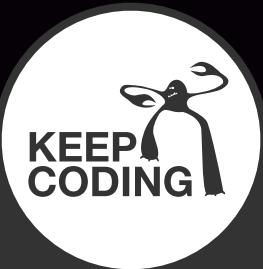
- ¿Que verdad?

Que no hay cuchara"



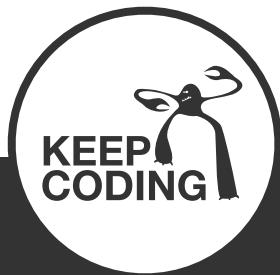


■ Truthy and Falsy



■ Que son Truthy y Falsy

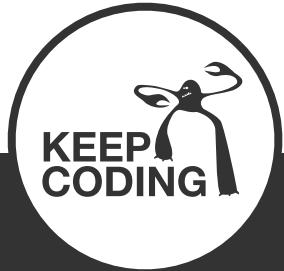
En javascript todo tiene un valor booleano, generalmente conocido como truthy o falsy.



■ Como se produce

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

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

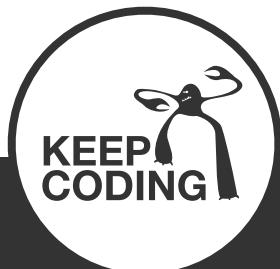


■ Reglas de truthy / falsy

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:

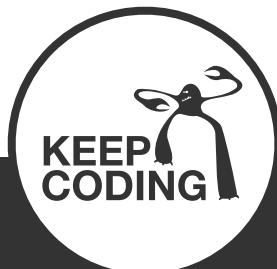
```
var c = (false == 0); // true
var d = (false == ""); // true
var e = (0 == ""); // true
```

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

```
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)
```

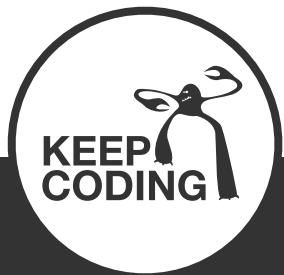
```
if ( "0" ) {console.log('si')} // "si" (se evalúa el string)
```







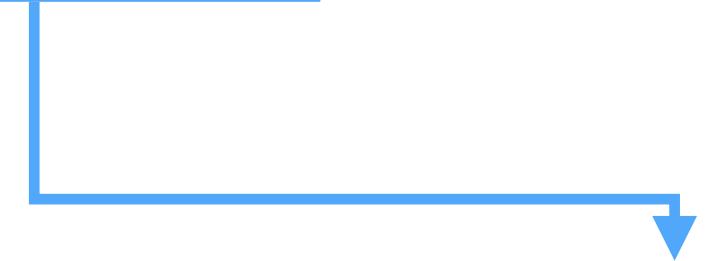
■ Una solución por favor!



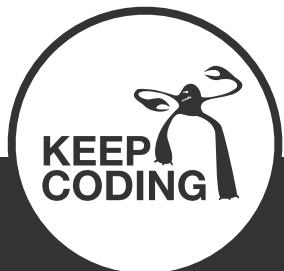
■ Truthy and Falsy

Usamos el **igual estricto** (`==`) y el **distinto estricto** (`!=`)

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



Se comparan **primero por su tipo** y luego
por su valor





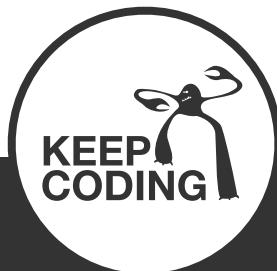
this



this

La palabra clave `this` trata de representar al objeto que llama nuestra función como método, no donde está definida.

Por lo general, su valor hace referencia **al objeto propietario** de la función que la está invocando o en su defecto, al objeto donde dicha función es un método.



CUIDADO!

De forma general, cuando se usa en algo **distinto a un método** su valor es el contexto global (o *undefined* si estamos en modo estricto).

```
console.log(this); // window en un browser, global en node

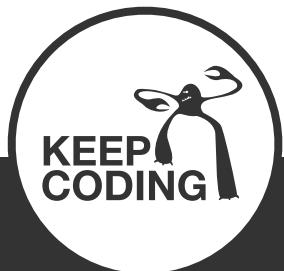
function pinta(){
  console.log(this); // window en un browser, global en node
}
```



Averigua porque...

```
var persona = {  
    name: 'Luis',  
    surname: 'Gomez',  
    fullname: function() {  
        console.log(this.name + ' ' + this.surname);  
    }  
};  
  
persona.fullname(); // Luis Gomez  
setTimeout(persona.fullname, 1000); // undefined undefined  
  
// pista: quien está invocando realmente la función fullName?
```

ejemplos/this.js



this

Como manejarlo? Le asignamos this con bind

```
var persona = {  
    name: 'Luis',  
    surname: 'Gomez',  
    fullname: function() {  
        console.log(this.name + ' ' + this.surname);  
    }  
};  
  
setTimeout(persona.fullname.bind(persona), 1000); // Luis Gomez
```

ejemplos/this.js



this

Otras formas de asignar this a una función

```
funcion.call(thisArg[, arg1[, arg2[, ...]]])
```

```
persona.iniciales.call(programa, 2, true);
```

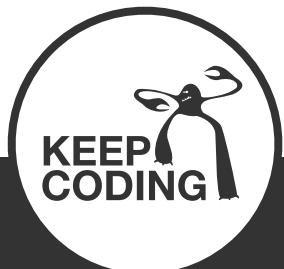
```
funcion.apply(thisArg[, argsArray])
```

```
persona.iniciales.apply(programa, [2, false]);
```

La diferencia es:

- en **call** hay que poner todos los argumentos separados por **comas**
- en **apply** se le pasan como un **array**

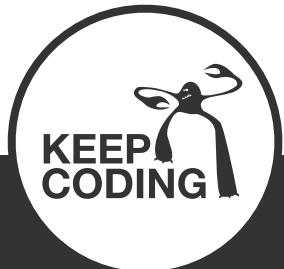
ejemplos/this2.js



this - uso en constructores

Cuando usamos this en un constructor de objetos, este apuntará al objeto returned por el constructor. Pero cuidado con quien llama a los métodos!

```
function Coche() {  
    this.ruedas = 4;  
    this.logRuedas = function() {  
        console.log('tiene ' + this.ruedas);  
    }  
}  
  
var coche = new Coche();  
console.log(coche.ruedas); // 4  
coche.logRuedas(); // tiene 4  
setTimeout(coche.logRuedas, 1000); // tiene undefined
```



A scene from The Matrix featuring Neo (Keanu Reeves) standing in a field of glowing green spheres. He is wearing his signature black trench coat and sunglasses. His right hand is raised, palm facing forward, with several small glowing circles floating around it. The background is a dense field of similar glowing spheres.

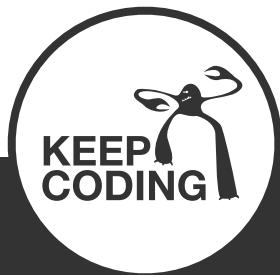
■ Closures



Closures

Un closure se construye con una función (A) que devuelve otra (B).

La función devuelta (B), sigue manteniendo el acceso a todas las variables de la función que la creó (A).

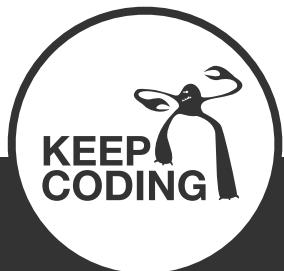


Closures

Ejemplo:

```
function creaClosure(valor) {  
    return function() {  
        return valor;  
    }  
}
```

ejemplos/closure.js



Closures

Algo más elaborado:

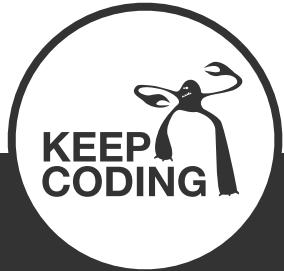
```
function creaAgente(nombre) {
    var edad = 0;
    return {
        ponNombre: function(nuevoNombre) {
            nombre = nuevoNombre;
        },
        leeNombre: function() {
            return nombre;
        },
        ponEdad: function(nuevaEdad) {
            edad = nuevaEdad;
        },
        leeEdad: function() {
            return edad;
        }
    }
}
```

ejemplos/closure2.js





Prototipos



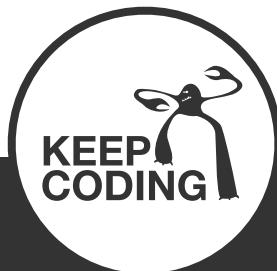
■ Prototype chain

Casi todo en Javascript es un objeto. Cada objeto tiene una propiedad interna llamada prototype que apunta a otro objeto.

Su objeto prototipo tiene a su vez una propiedad prototype que apunta a otro objeto, y así sucesivamente.

A esto se le llama cadena de prototipos.

Si sigues la cadena en algún momento llegarás al objeto Object, cuyo prototipo es null.

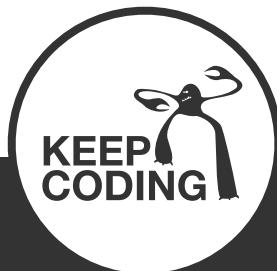


■ Prototipos

Cuando pides una propiedad a un objeto el interprete mira a ver si la tiene ese objeto. Si no la encuentra mira a ver si la tiene su prototipo, y así hasta llegar al final de la cadena.

Si no lo encuentra devuelve el error correspondiente.

Esto nos permitiría hacer algo parecido a clases e implementar herencia.

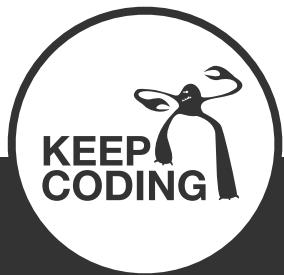


■ Prototipos

```
function Persona(name) {  
    this.name = name;  
}  
  
Persona.prototype.saluda = function() {  
    console.log("Hola, me llamo " + this.name);  
};
```

Esto hará que todas las personas sepan saludar, ¡incluso las que ya estén creadas!

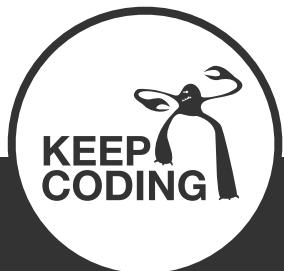
ejemplos/prototipos.js



■ Prototipos - herencia

```
function Agente(name) {  
    Persona.call(this, name);  
    // heredamos el constructor  
    // Esto ejecutará el constructor de Persona sobre el this de Agente  
    // definiendo en el this de Agente sus propiedades.  
    // Es como llamar a "super" en otros lenguajes  
}  
  
// heredamos las propiedades de prototipo  
Agente.prototype = new Persona(); // heredaría name y saluda()
```

Así Agente hereda de Persona.



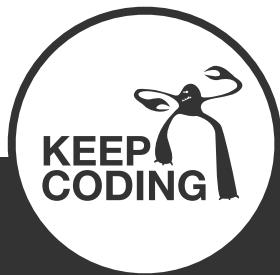
ejemplos/prototipos.js

Extender

Podemos copiar las propiedades de un objeto a otro.

Esto se suele llamar extender un objeto con otro.

```
const config = Object.assign({}, destino, origen);
```

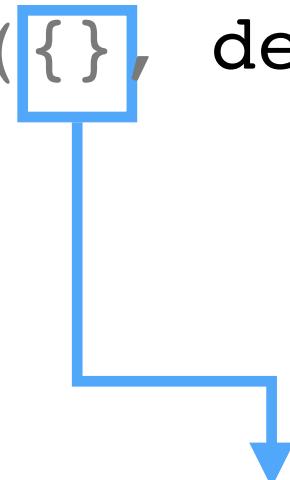


Extender

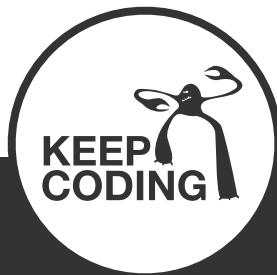
Podemos copiar las propiedades de un objeto a otro.

Esto se suele llamar extender un objeto con otro.

```
const config = Object.assign({}, destino, origen);
```



El objeto destino **es modificado**



■ Herencia múltiple - mixins

Una forma de conseguir herencia múltiple es usar el patrón mixin:

- Para conseguirlo heredamos del principal (ya lo hemos visto antes)
- Luego extendemos el prototipo del heredado con los mixins:

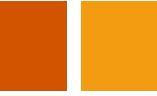
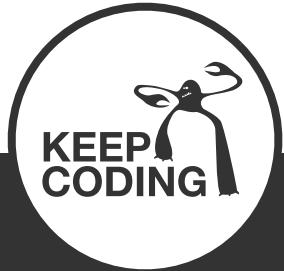
```
Agente.prototype = Object.assign(Agente.prototype, mixin);
```

ejemplos/mixin.js



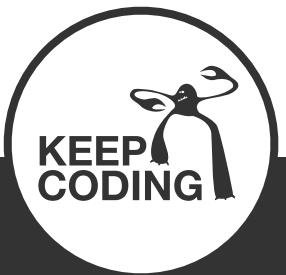


Clases



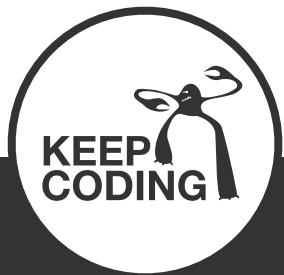
■ Clases

```
class Mascota {  
  
    constructor(nombre) {  
        this.nombre = nombre;  
    }  
  
    saluda() {  
        console.log(`Hola soy ${this.nombre}`);  
    }  
}
```



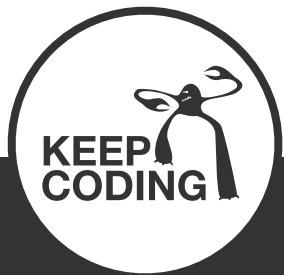
■ Clases

```
const mascota = new Mascota('Toby');  
  
mascota.saluda();
```



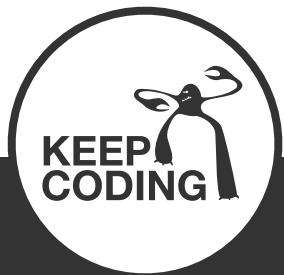
■ Clases

```
class Perro extends Mascota {  
    constructor(nombre) {  
        super(nombre);  
    }  
  
    let perro = new Perro('Niebla');  
  
    perro.saluda();
```





■ Process



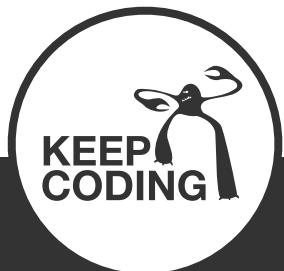
■ Process

El objeto global process tiene muchas propiedades que nos serán útiles, como `process.platform`, que en OS X nos dirá '`darwin`', en linux '`linux`', etc.

También tiene métodos útiles como `process.exit(int)` que para node estableciendo un exit code.

O eventos como `process.on('exit', callback)` donde podemos hacer cosas antes de salir.

[ejemplos/process.js](#)

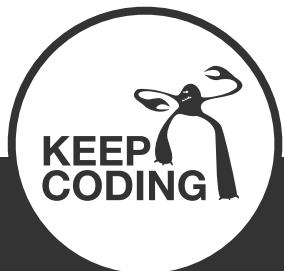


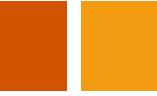
■ Process

Tiene un método muy interesante, `process.nextTick`, que recibe un solo argumento, un callback.

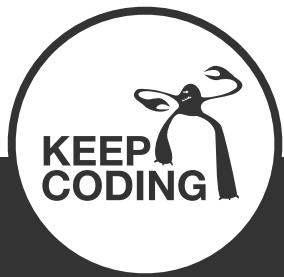
```
process.nextTick(function () {  
  console.log('Siguiente vuelta del event loop, whooouuu!')  
});
```

Lo que hará es colocar la función de nuestro callback al principio de la siguiente vuelta del event loop (ahora vemos que es eso).



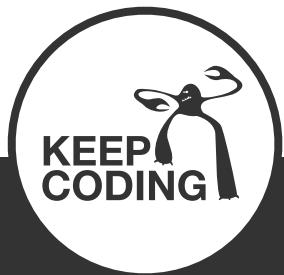


■ Events





■ Event loop

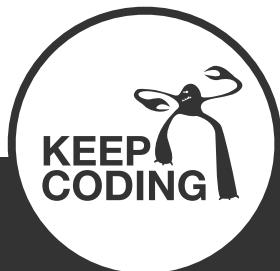


■ Event loop

Node usa un solo hilo.

Tiene un bucle interno, que podemos llamar 'event loop' donde en cada vuelta ejecuta todo lo que tiene en esa 'fase', dejando los callbacks pendientes para otra vuelta.

La siguiente vuelta mira a ver si ha terminado algún callback y si es así ejecuta su handlers.

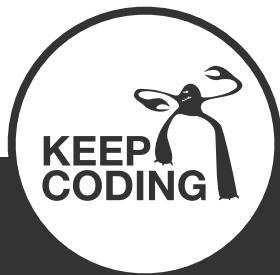


■ Event loop

El equivalente de esta construcción

```
process.nextTick(function () {  
  console.log('Siguiente vuelta del event loop, whooouuu!')  
});
```

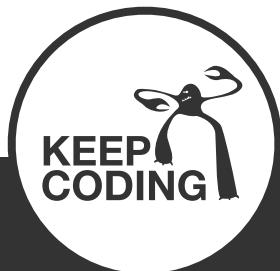
Es como decir a node "cuando vuelvas a comprobar los callbacks finalizados haz esto!".



■ Non blocking

Si node se quedara esperando hasta que termine una query o una petición a Facebook, acumularía demasiados eventos pendientes y dejaría de atender a las siguientes peticiones, ya que como dijimos **usa un solo hilo.**

Por eso todos las llamadas a funciones que usan IO (por ejemplo escritura o lectura en disco, la red, bases de datos, etc) se hacen de forma asíncrona. Se aparcan para que nos avisen cuando terminen.

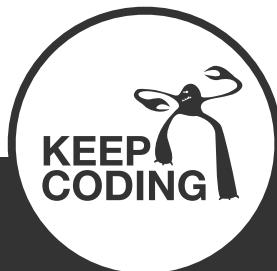


■ Events

Node nos proporciona una forma de manejar IO en forma de eventos.

Usando el EventEmitter podemos colgar eventos de un identificador.

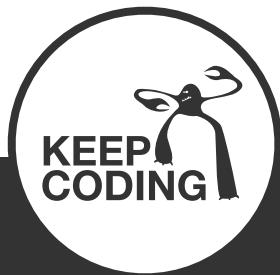
```
eventEmitter.on('llamar telefono', suenaTelefono);
```



■ Events

Y podemos emitir el identificador cuando queremos que sus eventos 'salten'

```
eventEmitter.emit('llamar telefono');
```



■ Events

Incluso podemos darle argumentos al identificador para que se los dé a los manejadores.

```
eventEmitter.emit('llamar telefono', 'madre');
```

Nuestro manejador recibirá el parametro

```
var suenaTelefono = function(quien) {  
    ...  
}
```

ejemplos/
eventemitter.js



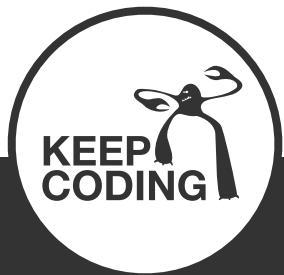
■ Events

Son cómodos para procesar streams.

Mandando a un fichero todo lo que se escriba en la consola.

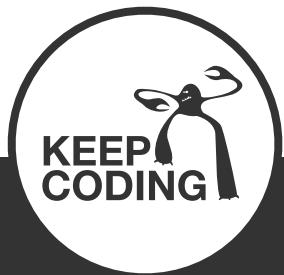
```
process.stdin.on('data', function(data) {  
  file.write(data);  
});
```

[ejemplos/eventedio.js](#)





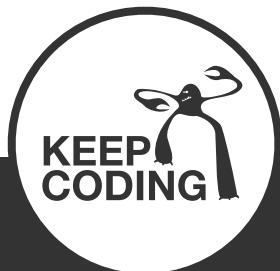
■ Módulos



Módulos

Los módulos de Node.js se basan en el estándar **CommonJS**.

- Los módulos usan **exports** para exportar cosas.
- Quien quiere usar un módulo lo carga con **require**.
- La instrucción require es **síncrona**.
- Un módulo es un **singleton**



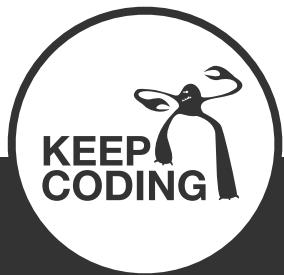
Módulos

Un módulo básico:

```
// modulo.js  
console.log('Hola desde un módulo!');
```

```
// index.js  
require('./modulo.js');
```

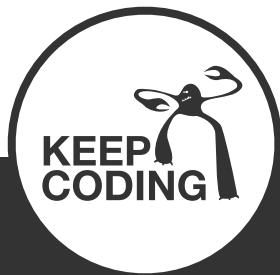
ejemplos/modulos/
basico.js



Módulos

Donde busca Node.js los módulos

1. Si es un módulo del core
2. Si empieza por './' or '/' or '../' va a la ruta calculada desde la situación del fichero actual
3. Módulos en la carpeta node_modules local
4. Módulos en la carpeta node_modules global



Módulos

```
require('../../../../lib/files/modulo.js');
```

Como evitar esto? Asignando NODE_PATH a la ruta donde meteremos nuestras librerías en el arranque de nuestra app.

```
NODE_PATH=lib node index.js
```

```
require("files/modulo.js"); // mejor así!
```

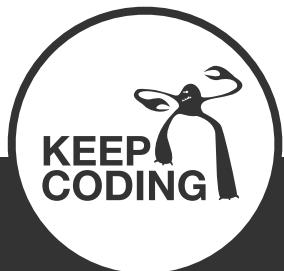


Módulos

O si usamos npm a partir de 2.0 podemos anotarlas en el package.json

```
{  
  "name": "baz",  
  "dependencies": {  
    "bar": "file:../foo/bar"  
  }  
}
```

<https://docs.npmjs.com/files/package.json#local-paths>

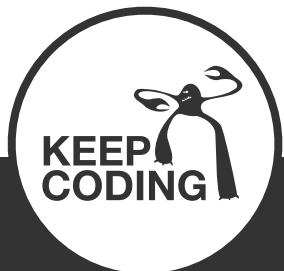


Módulos

¿module.exports o exports?

- exports es un alias de module.exports.
- Node lo crea automáticamente
- Para hacer exports con nombre se pueden usar los dos indistintamente

Si asignamos algo directamente a exports deja de ser un alias.
Solo podemos usarlo con exports.loquesea

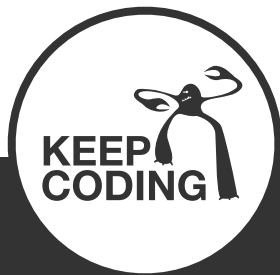


Módulos

Node carga un módulo una sola vez. En el primer require.

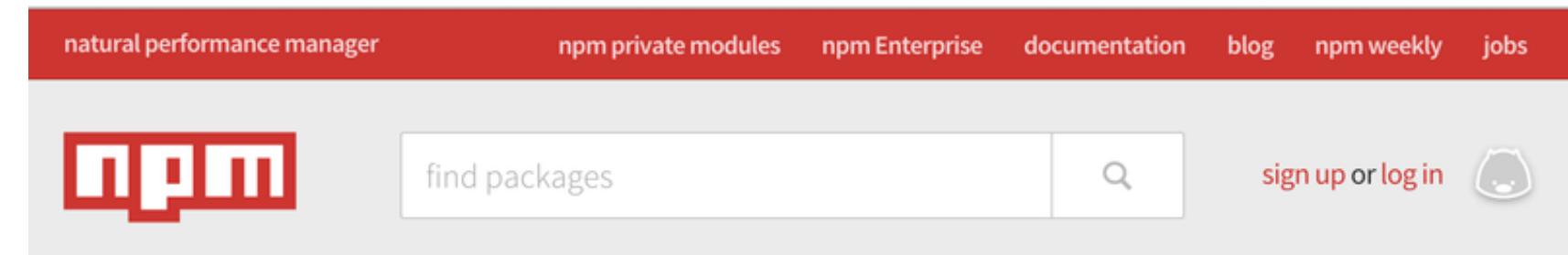
Las siguientes llamadas a require con la misma ruta devolverán el mismo objeto que se creo en la primera llamada.

Esto nos vendrá muy bien con las conexiones a bases de datos, por ejemplo.



Módulos

Principalmente podemos encontrar módulos de terceros en npmjs.com



npm is the package manager for javascript.

179.374
total packages

37.682.863
downloads in the last day

598.105.505
downloads in the last week

2.198.864.721
downloads in the last month

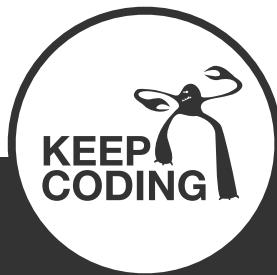
packages people 'npm install' a lot



■ ES Modules

Los módulos de EcmaScript están disponibles de manera estable en Node.js

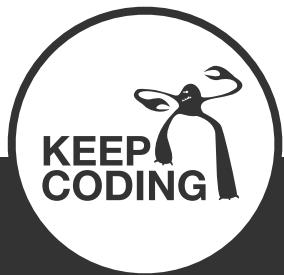
<https://medium.com/dailyjs/javascript-module-cheatsheet-7bd474f1d829>





Ejercicio

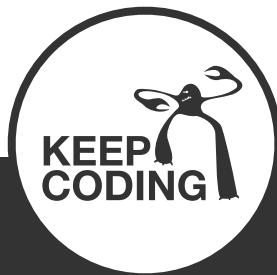
Versión de un módulo



Ejercicio - versión de un módulo

Hacer una función llamada **versionModulo** que reciba un nombre de módulo y un callback.

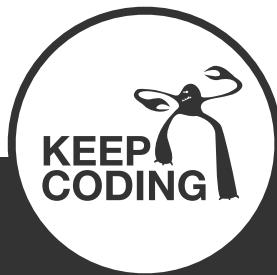
Debe devolver un posible error y la versión del módulo en el callback.



Ejercicio - versión de un módulo

Ingredientes:

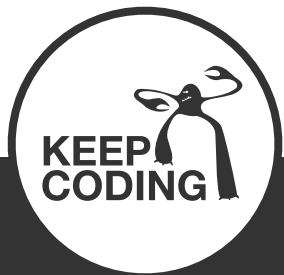
- **fs.readFile** (https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback)
- **path.join** (https://nodejs.org/api/path.html#path_path_join_path1_path2)
- **JSON.parse** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse)



Ejercicio - versión de un módulo

La probaremos con un código como este:

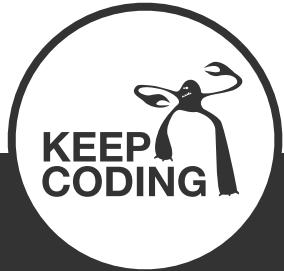
```
versionModulo( 'chance' , function(err, str) {  
  if (err) {  
    console.error('Hubo un error: ', err);  
    return;  
  }  
  console.log('La version del módulo es:', str);  
} );
```





Ejercicio

Versión módulos

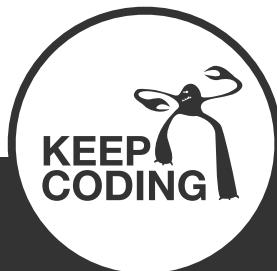


Ejercicio - versión de módulos

Modificar la función **versionModulo** para que consiga la versión de todos los módulos instalados en mi proyecto.

Más ingredientes:

- `fs.readdir` (https://nodejs.org/api/fs.html#fs_fs_readdir_path_callback)
- `async.concat` (<https://github.com/caolan/async#concat>)
- `async common pitfalls` <https://github.com/caolan/async#common-pitfalls-stackoverflow>
- `fs.statSync` (https://nodejs.org/api/fs.html#fs_fs_statsync_path)



Ejercicio - versión de un módulo

Recibirá un callback. Debe devolver un posible error y un array con los nombres y versiones de los módulos que encuentre.

La probaremos con un código como este:

```
versionModulos(function(err, moduleArr) {  
  if (err) {  
    console.error('Hubo un error: ', err);  
    return;  
  }  
  console.log('Los módulos son:', moduleArr);  
});
```

