



# Introducción a node.js, a excepciones y a promesas

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# node.js y sockets - Índice

1. <u>node.js, librería de módulos, entorno de ejecución y modo estricto</u> .....	3
2. <u>Módulos node.js: module.exports, require, __dirname y __filename</u> .....	10
3. <u>Paquete npm, directorio de un proyecto, package.json y node_modules</u> .....	18
4. <u>Timers, eventos, flujos (streams), stdin, stdout y stderr</u> .....	25
5. <u>Acceso a Ficheros: readFile, writeFile, appendFile, readStream, writeStream,</u> .....	35
6. <u>Gestión de la concurrencia, bucle de eventos y nextTick</u> .....	42
7. <u>Errores, excepciones y sentencias throw y try...catch...finally</u> .....	47
8. <u>Promesas: new Promise(..), resolve, reject, then, catch</u> .....	56
9. <u>Más ejemplos con Promesas</u> .....	64



# node.js, librería de módulos, entorno de ejecución y modo estricto

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# node: JavaScript en el servidor



## ◆ node

- Entorno de desarrollo de aplicaciones JavaScript para S.O. UNIX
  - ◆ Incluye un **comando UNIX** que ejecuta aplicaciones JavaScript adaptadas al nuevo entorno
    - Ejecuta programas enteros, o en modo interactivo
  - ◆ Incluye una **librería de paquetes** de acceso a los servicios de UNIX y de Internet
    - Descarga y documentación de node: <https://nodejs.org/>
- node ha sido portado a ES6 desde la v4, <https://nodejs.org/es/docs/es6/>

## ◆ node crea un entorno de desarrollo modular

- Donde los módulos tienen espacios de nombres separados
  - ◆ Y donde el programa principal y los módulos pueden importar otros módulos

## ◆ node ha tenido mucho éxito y se utiliza en múltiples portales

- E-bay, PayPal, LinkedIn, Netflix, Yahoo, Google, ...

```
var express = require('express');
var path = require('path');

var app = express();

app.use(express.static(path.join(__dirname, 'public')));

app.listen(8000);
```

Ejemplo de servidor estático de páginas Web en node.js

venus-2:~ jq\$ node --help  
Usage: node [options] [-e script] [args]  
node debug script.js [args]  
  
Options:  
-v, --version  
-e, --eval script  
-p, --print 4  
--interactive  
print no  
evaluate  
print re  
always a

# Documentacion node.js (v6)

<https://nodejs.org/dist/latest-v6.x/docs/api/>

## ◆ La librería de **node** incluye dos tipos de módulos

- Módulos accesibles siempre
  - **Console**: acceso a la consola de ejecución
  - **Globals**: entorno global de node.js
  - **Modules**: implementación de los módulos node.js
  - **Process**: proceso donde se ejecuta la aplicación
  - **Timers**: métodos de gestión de temporizadores
- Los demás módulos deben importarse con "require(...)"
  - **Assertion Testing**: testing de programas
  - **Child Processes**: creación de procesos hijos
  - **Cluster**: despliegue en clusters de procesadores
  - **Crypto**: cifrado de información
  - **Debugger**: depurador de aplicaciones
  - **Events**: librería de eventos del sistema
  - **File System (fs)**: accesos al sistema de ficheros
  - **HTTP**: programación de transacciones HTTP
  - **HTTPS**: transacciones HTTP seguras
  - **Net**: librería de sockets TCP (apl. cliente-servidor)
  - **OS**: Acceso a datos del S.O.
  - **Readline**: entrada por línea de comandos
  - **UDP/Datagram**: librería de sockets UDP
  - **URL**: gestión de URLs
  - **ZLIB**: compresión de información
  - .....

# Node.js v8.9.1 Documentation

[Index](#) | [View on single page](#) | [View as JS](#)

## Table of Contents

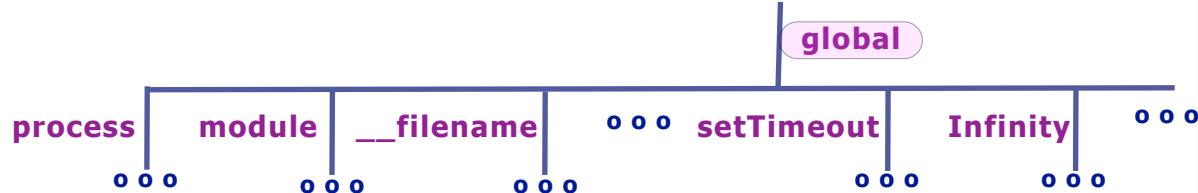
- [About these Docs](#)
- [Usage & Example](#)

---

- [Assertion Testing](#)
- [Async Hooks](#)
- [Buffer](#)
- [C++ Addons](#)
- [C/C++ Addons - N-API](#)
- [Child Processes](#)
- [Cluster](#)
- [Command Line Options](#)
- [Console](#)
- [Crypto](#)
- [Debugger](#)
- [Deprecated APIs](#)
- [DNS](#)
- [Domain](#)
- [ECMAScript Modules](#)
- [Errors](#)
- [Events](#)
- [File System](#)
- [Globals](#)

0 0 0 0 0 0

# Objeto global y módulos



```
console.log(`\nEnumerable global properties  
obtained with Object.keys(global):\n`);
```

```
console.log(Object.keys(global));
```

Programa 01-global.js que muestra las propiedades enumerables de global.

```
ej --- bash --- 42:00
. .
. . node 01-global.js
Muestra las propiedades enumerables de global.

Enumerable global properties obtained with Object.keys(global):
[ 'DTRACE_NET_SERVER_CONNECTION',
  'DTRACE_NET_STREAM_END',
  'DTRACE_HTTP_SERVER_REQUEST',
  'DTRACE_HTTP_SERVER_RESPONSE',
  'DTRACE_HTTP_CLIENT_REQUEST',
  'DTRACE_HTTP_CLIENT_RESPONSE',
  'global',
  'process',
  'Buffer',
  'clearImmediate',
  'clearInterval',
  'clearTimeout',
  'setImmediate',
  'setInterval',
  'setTimeout',
  'console' ]
. $
```

## ◆ Objeto global

- Objeto compartido por todos los módulos que crea el ámbito global
  - Incluye elementos del lenguaje JavaScript: **Nan**, **Infinity**, **Date**, **Number**, ...
  - Incluye elementos específicos de node: **process**, **console**, ...
    - Documentación: <http://nodejs.org/api/globals.html>

## ◆ Las propiedades de global se pueden referenciar solo con el nombre

- Por ejemplo, **process** se referencia como **global.process** o **process**

# Objeto process y argv

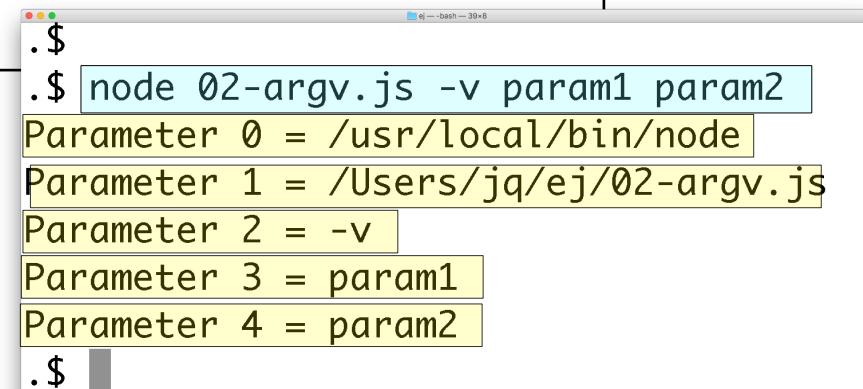
## ◆ Node se ejecuta en un proceso del S.O. (UNIX, ...)

- **global.process**: es el interfaz con el proceso desde node.js, por ejemplo
  - **process.exit([code])** termina la ejecución del proceso de node
  - **process.env** contiene un objeto con las variables de entorno del proceso
    - Documentación: <https://nodejs.org/dist/latest-v6.x/docs/api/process.html>

## ◆ **process.argv**: es un array con los parámetros de la invocación

- Permite acceder a ellos desde el programa node.js

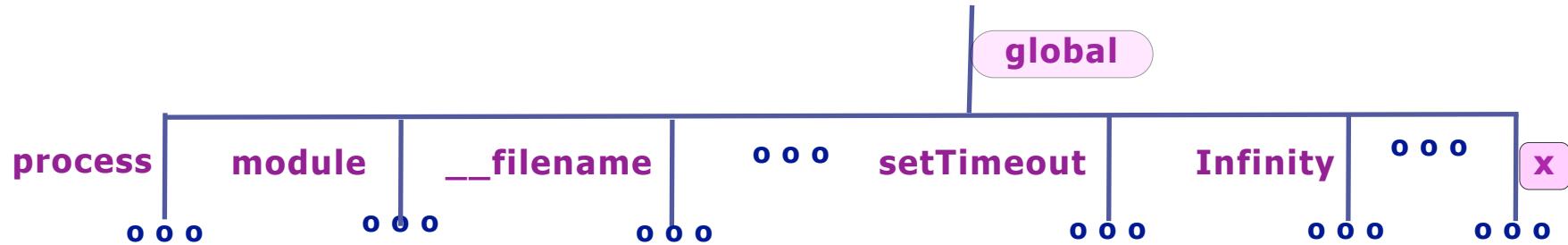
```
var i;
for(i = 0; i < process.argv.length; i++) {
    console.log('Parameter ' + i + " = " + process.argv[i]);
}
```



The screenshot shows a terminal window with the following text:

```
.$
.$ node 02-argv.js -v param1 param2
Parameter 0 = /usr/local/bin/node
Parameter 1 = /Users/jq/ej/02-argv.js
Parameter 2 = -v
Parameter 3 = param1
Parameter 4 = param2
.$
```

# Propiedades globales y entorno de ejecución



- ◆ Un programa JavaScript se ejecuta con el objeto **global** como entorno
  - Cuando la **variable x no está definida** la asignación **x = 1;**
    - ◆ Crea una nueva **propiedad de global** de nombre **x** en ámbito el global (objeto global)
      - **x = 1;** es equivalente a **global.x = 1;**
    - ◆ Esta propiedad será **visible en todos los módulos**, porque global se comparte en todos ellos
  - ◆ Olvidar definir una variable, es un **error muy habitual**
    - y al **asignar un valor a la variable no definida**, JavaScript no da error
      - ◆ sino que crea una **nueva propiedad del entorno global**
    - Es un **error de diseño de JavaScript** y hay que tratar de evitarlo
      - ◆ Ejecutar programas JavaScript en **modo estricto** ('use strict') permite evitarlo

# Modo estricto: 'use strict'

## ◆ El modo estricto de ejecución de un programa JavaScript

- Protege contra el uso de las 'partes malas' y refuerza la seguridad
  - ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

## ◆ El modo estricto prohíbe usar ciertas partes, lanzando errores

- Así evita el uso de las partes no recomendadas de JavaScript, p.e.
  - ◆ Crear propiedades dinámicamente en el entorno global por error
  - ◆ Utilizar variables o funciones todavía no definidas
  - ◆ Asignar, borrar o crear propiedades del entorno global o no permitidas
    - Por ejemplo global.NaN, global.infinity, Object.prototype, ..
  - ◆ Incluir varias propiedades o parámetros de funciones de igual nombre
  - ◆ Añadir propiedades a valores primitivos
  - ◆ Utilizar la **sentencia with**
  - ◆ Crear **variables globales** con eval(...)
  - ◆ Borrar variables con delete
  - ◆ ...

## ◆ El string 'use strict' activa el modo estricto

- '**use strict**' al principio de un programa activa el modo estricto en todo el programa
  - ◆ En node lo activa en todo el módulo que comienza por 'use strict'
- '**use strict**' al principio de una función lo activa solo en el código de la función
- **Ojo:** si 'use strict' no está antes de la primera sentencia en ambos casos, no tiene efecto

En modo **no estricto** asignar a una **variable no definida** crea una **propiedad de global!**

```
x = 1;  
x // => 1
```

En modo **estricto** asignar una **variable no definida** provoca un **error de ejecución**.

```
'use strict'  
x = 1; // => Error
```

En modo **estricto** si es posible **crear explícitamente propiedades de global.**

```
'use strict'  
global.y = 1;  
y // => 1
```



# Módulos node.js: module.exports, require, \_\_dirname y \_\_filename

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Módulos node.js

## ◆ **node.js** permite partir un programa en **módulos**

- Cada **módulo** es un **fichero diferente** con dos partes
  - ◆ **Interfaz e Implementación**

## ◆ **Interfaz**

- Parte pública que permite acceso al módulo desde otros módulos
  - ◆ El interfaz da acceso a la implementación o parte privada del módulo

## ◆ **Implementación**

- Parte privada o local del módulo que crea la funcionalidad exportada
  - ◆ El código y el espacio de nombres es local y no es accesible desde el exterior

## ◆ **ES6 ha definido también su propio sistema de módulos**

- Los módulos de ES6 son diferentes (de node.js) y su uso es todavía limitado
  - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export>
  - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>

# Interfaz e implementación de un módulo node.js

## ◆ Interfaz de un módulo node.js

- Parte del módulo que otros módulos pueden importar y utilizar
- El interfaz se define de dos formas
  - ◆ **exports.nombre = <nombre>** exporta una colección de propiedades y métodos individuales
  - ◆ **module.exports = <objeto\_interfaz>** exporta solo un objeto principal o constructor
    - Debe exportarse solo un objeto principal o una colección de métodos y propiedades

## ◆ Implementación de un módulo node.js

- Parte del código que implementa la funcionalidad exportada a otros módulos
  - ◆ Son las variables, funciones y otras definiciones locales aisladas del exterior
    - Esta parte se encapsula en un **cierre** con un **espacio de nombres aislado**

## ◆ Objeto global de JavaScript está accesible en todos los módulos

- Documentación: <http://nodejs.org/api/modules.html>

## ◆ 'use strict' al comienzo del módulo activa el modo estricto

- Permite reforzar la seguridad del código JavaScript del módulo

# Importar módulos: método require

## ◆ El método **require(<module>)** importa todo lo que exporta **<module>**, donde

- **<module>** puede ser el nombre de un módulo instalado, por ejemplo
  - ◆ Los módulos de la **librería** de node.js: 'fs', 'net', 'http', ..
  - ◆ Los módulos instalados con **npm** en **node\_modules**: 'express', 'sequelize', 'sqlite', ..
- **<module>** puede ser una **ruta a un fichero**, que debe comenzar por: .. o /
  - ◆ '**./mod.js**' identifica el fichero con el módulo **mod.js** en el **mismo directorio**
  - ◆ '**../mod.js**' identifica el fichero con el módulo **mod.js** en el **directorio padre**
  - ◆ '**/usr/u7/mod.js**' identifica el fichero con el módulo **mod.js** con la **ruta absoluta** dada
- **<module>** puede identificar también un directorio (la casuística es compleja)
  - ◆ Ver: [https://nodejs.org/dist/latest-v6.x/docs/api/modules.html#modules\\_file\\_modules](https://nodejs.org/dist/latest-v6.x/docs/api/modules.html#modules_file_modules)

## ◆ Cache de módulos

- **require** utiliza una **cache** para cargar una sola vez un módulo requerido varias veces
  - ◆ Por ejemplo, el **programa principal** importa el **módulo A** y el **módulo B**
    - Si el **módulo B** importa también el **módulo A**, la cache solo carga el **módulo A** una vez

## ◆ Control de **ciclos**

- La cache comprueba si un modulo se importa cíclicamente, evitando bucles infinitos
  - ◆ Por ejemplo, el **módulo A** importa el **módulo B** y el **módulo B** importa el **módulo A**

# Módulo: Nombres locales y exportados

## ◆ Todos los módulos tienen definidos 5 propiedades locales

- **\_\_dirname**: ruta al directorio del módulo
- **\_\_filename**: ruta al fichero con el módulo
- **module**: referencia al propio módulo
- **exports**: referencia a **module.exports**
- **require**: método para importar módulos

## ◆ El espacio de nombres local

- incluye además todas las definiciones locales
  - Definiciones de variables con **var**, **let** o **const**
  - Definiciones de **funciones**
  - Definiciones de **clases**
  - ...

## ◆ El módulo exporta solo los objetos asignados asignados explícitamente a

- **module.exports** o a **exports.xx**

## ◆ Doc: <https://nodejs.org/dist/latest-v6.x/docs/api/globals.html>

```
console.log();
console.log("__dirname: " + __dirname);
console.log();
console.log("__filename: " + __filename);
```

The screenshot shows a terminal window titled 'ej - bash - 42x7'. The command '. \$ . node 05-filename.js' is entered. The output shows two lines: '\_\_dirname: /Users/jq/ej' and '\_\_filename: /Users/jq/ej/05-filename.js'. A dashed arrow points from the code block above to the terminal window.

```
.$
.$ node 05-filename.js
__dirname: /Users/jq/ej
__filename: /Users/jq/ej/05-filename.js
.$
```



# Ejemplo de módulos de node.js

- ◆ Programa con dos módulos, cada uno en un fichero diferente

- **Programa principal:** fichero **main.js**

- ◆ importa el módulo con: `var circle = require('./circle.js');`
  - ambos ficheros están en el mismo directorio y hay que utilizar el path `'./circle.js'`

- **Módulo importado:** fichero **circle.js**

- ◆ Exporta los dos métodos de la interfaz, **area** y **circumference**, con `exports.<método> = .....`

```
// Main program (file 06-main.js), imports circle.js (7-circle.js)
```

**06-main.js**

```
var circle = require('./07-circle.js'); // path to 07-circle.js in the same directory is ./circulo.js
console.log( 'Area (radius 4): ' + circle.area(4)); // => Area (radius 4): 50.26548245743669
```

```
// Module: file 07-circle.js with library of methods
// -> exports methods in module.exports
```

**07-circle.js**

```
var _PI = Math.PI; // private module variable, not visible outside
// convention: private names start usually with _
exports.area = function (r) { return _PI * r * r; }; // exported method
exports.circumference = function (r) { return 2 * _PI * r; }; // exported method
```

```

module.exports = function agenda (title, init) {
  let _title = title;
  let _content = init;

  return {
    title: function() { return _title; },
    add: function(nombre, tf) { _content[nombre]=tf; },
    tf: function(nombre) { return _content[nombre]; },
    remove: function(nombre) { delete _content[nombre]; },
    toJSON: function() { return JSON.stringify(_content); },
    fromJSON: function(agenda) { Object.assign(_content, JSON.parse(agenda)); }
  }
}

```

10-mod\_ag\_closure.js

Modulo: **10-mod\_ag\_closure.js**  
exporta un cierre (closure) que devuelve un objeto con los métodos de acceso a agenda.

## Agenda: modulo node.js que encapsula un cierre

var Agenda = require('./11-mod\_ag\_closure.js') importa el módulo **05-mod\_ag\_closure.js** y lo guarda en la variable agenda.

La variable **friends** guarda una instancia de la agenda con teléfonos de amigos.

```
const agenda = require('./10-mod_ag_closure');
```

```

let friends = agenda ("friends",
  { Peter: 913278561,
    John: 957845123
  });
friends.add("Mary", 978512355);

```

```

let work = agenda ("Work", {});
work.fromJSON('{"Peter Tobb": 913278561, "Paul Smith": 957845123}');

```

```

console.log('Peter: ' + friends.tf("Peter"));
console.log('Mary: ' + friends.tf("Mary"));
console.log('Edith: ' + friends.tf("Edith"));
console.log();
console.log('Peter Tobb: ' + work.tf("Peter Tobb"));
console.log('Work agenda: ' + work.toJSON());

```

```

$ node 11-main_ag_closure.js
Peter: 913278561
Mary: 978512355
Edith: undefined

Peter Tobb: 913278561
Work agenda: {"Peter Tobb":913278561,"Paul Smith":957845123}
$ 

```

11-main\_ag\_closure.js

La variable **work** guarda otra instancia de la agenda con teléfonos del trabajo.

11-main\_ag\_closure.js es el **programa principal** que importa el fichero **10-mod\_ag\_closure.js** (modulo), crea 2 agendas (friends, work) y muestra por consola partes de su contenido.

```

function Agenda (title, init) {
  this.title = title;
  this.content = init;
};

Agenda.prototype = {
  title: function() { return this.title; },
  add: function(name, tf) { this.content[name]=tf; },
  tf: function(name) { return this.content[name]; },
  remove: function(name) { delete this.content[name]; },
  toJSON: function() { return JSON.stringify(this.content); },
  fromJSON: function(agenda) { Object.assign(this.content, JSON.parse(agenda)); }
}
module.exports = Agenda;

```

10-mod\_ag\_class.js

Modulo: 12-mod\_ag\_class.js  
exporta un cierre (closure) que devuelve un objeto con los métodos de acceso a agenda.

## Agenda: modulo node.js que define una clase

**var Agenda = require('./12-mod\_ag\_class.js')** importa el módulo 12-mod\_ag\_class.js y lo guarda en la variable agenda.

La variable **friends** guarda una instancia de la agenda con teléfonos de amigos.

```
const Agenda = require('./12-mod_ag_class');
```

```

let friends = new Agenda ("friends",
  { Peter: 913278561,
    John: 957845123
  });
friends.add("Mary", 978512355);

```

```

.$
.$ node 13-main_ag_class.js
Peter: 913278561
Mary: 978512355
Edith: undefined

Peter Tobb: 913278561
Work agenda: {"Peter Tobb":913278561,"Paul Smith":957845123}
.$

```

```

let work = new Agenda ("Work", {});
work.fromJSON('{"Peter Tobb": 913278561, "Paul Smith": 957845123}');

```

```

console.log('Peter: ' + friends.tf("Peter"));
console.log('Mary: ' + friends.tf("Mary"));
console.log('Edith: ' + friends.tf("Edith"));
console.log();
console.log('Peter Tobb: ' + work.tf("Peter Tobb"));
console.log('Work agenda: ' + work.toJSON());

```

11-main\_ag\_class.js

La variable **work** guarda otra instancia de la agenda con teléfonos del trabajo.

13-main\_ag\_class.js es el **programa principal** que importa el fichero 12-mod\_ag\_class.js (modulo), crea 2 agendas (friends, work) y muestra por consola partes de su contenido.



JavaScript



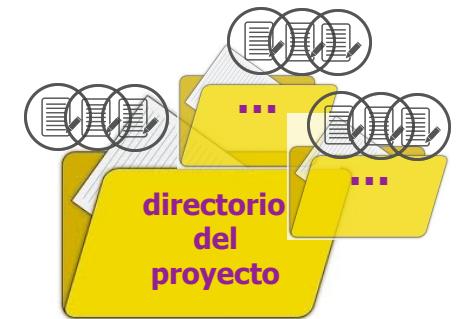
Paquete npm, directorio de un proyecto,  
package.json y node\_modules

Juan Quemada, DIT - UPM

# Estructura del directorio de un proyecto

## ◆ Directorio de un proyecto

- Contiene **todos los ficheros y subdirectorios** del proyecto
- Por ejemplo, en la mayoría de los proyectos suelen existir estos ficheros o directorios
  - **README.md** fichero resumen (formato GitHub markdown)
    - GitHub markdown: <https://github.com/github/markup/blob/master/README.md>
  - **LICENSE** fichero con licencia de distribución del proyecto
  - **public**: directorio donde almacenar **recursos Web**
  - **bin**: directorio donde almacenar **programas ejecutables**
  - **lib**: directorio con las **librerías** de software utilizadas
  - **test**: directorio con las pruebas de funcionamiento correcto



## ◆ Herramientas de gestión de un proyecto:

- Suelen utilizar **ficheros y subdirectorios** prefijados, por ejemplo
  - Herramienta de gestión de paquetes **npm**
    - Fichero **package.json**: guarda información del proyecto
    - Directorio **node\_modules**: contiene los paquetes de los que depende el proyecto
  - Herramienta de gestión de versiones **git**
    - Directorio **.git**: guarda el repositorio de versiones
    - Fichero **.gitignore**: indica los ficheros a ignorar por git
  - ....

# Paquetes npm y modulos node

## ◆ Paquete npm

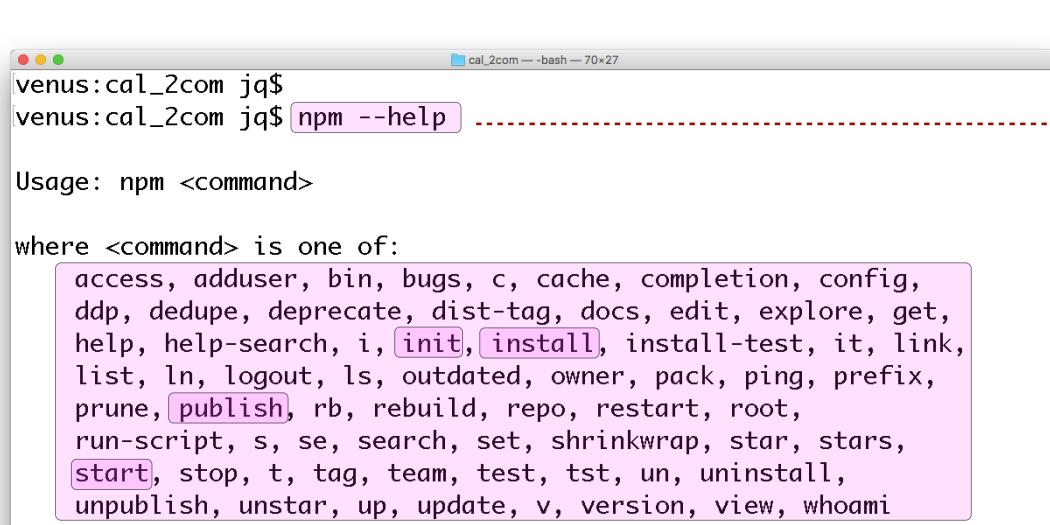
- Fichero o directorio descrito por un fichero **package.json**
  - El paquete facilita la documentación, búsqueda, inspección, distribución, instalación, uso, ...
    - <https://docs.npmjs.com/getting-started>

## ◆ comando npm (o cli-npm)

- Comando de UNIX y otros S.O. para creación y gestión de paquetes npm
  - Documentación: <https://www.npmjs.org/doc/>
  - Instalación: <https://docs.npmjs.com/getting-started/installing-node>, <http://blog.npmjs.org/post/85484771375/how-to-install-npm>

## ◆ Los paquetes npm son fundamentalmente **programas ejecutables** o **módulos**

- Un **programa ejecutable npm** se instala con **npm install ..** y se suele ejecutar con **npm start**
- Un **modulo npm** se instala con **npm install ..** y se importa con **require(..)**, ...
  - <https://docs.npmjs.com/getting-started/packages>



```
venus:cal_2com jq$ npm --help
Usage: npm <command>
where <command> is one of:
  access, adduser, bin, bugs, c, cache, completion, config,
  ddp, dedupe, deprecate, dist-tag, docs, edit, explore, get,
  help, help-search, i, init, install, install-test, it, link,
  list, ln, logout, ls, outdated, owner, pack, ping, prefix,
  prune, publish, rb, rebuild, repo, restart, root,
  run-script, s, se, search, set, shrinkwrap, star, stars,
  start, stop, t, tag, team, test, tst, un, uninstall,
  unpublish, unstar, up, update, v, version, view, whoami
```



# Fichero package.json

## ◆ Fichero package.json

- Contiene la información de gestión del proyecto en formato JSON
  - Está en el directorio raíz del proyecto
- Se puede crear con el comando: **npm init**
- Documentación
  - <https://docs.npmjs.com/files/package.json>

## package.json



## ◆ Contenido de package.json

- **Metadatos** del programa: **nombre**, **versión**, ...
- **Scripts** normalizados: arranque (**start**), tests (**test**), ....
- **Dependencias** de instalación del programa
  - Lista de paquetes **npm** necesarios para este proyecto
- .....

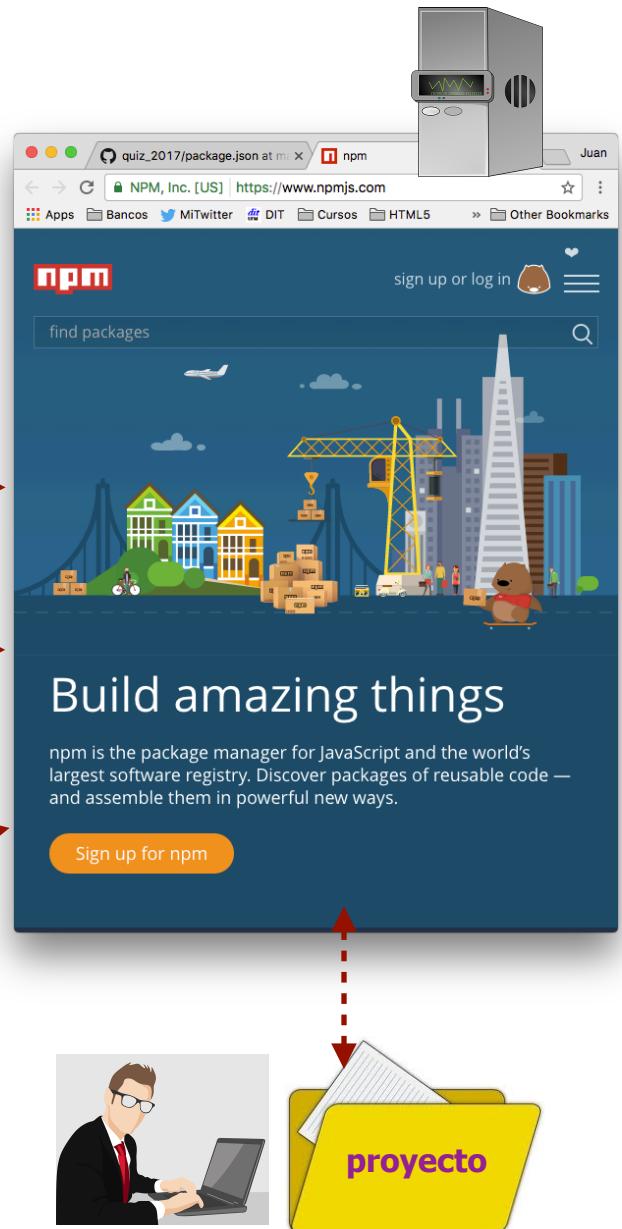
```
npm init <paquete>      // Inicializa un directorio como paquete añadiéndole  
                         // fichero package.json  
  
npm start                // Ejecuta script start de package.json  
  
npm test                 // Ejecuta script test de package.json (si existiese)  
.....
```

```
{  
  "name": "quiz",  
  "version": "0.0.0",  
  "private": true,  
  "scripts": {  
    "start": "node ./bin/www"  
  },  
  "dependencies": {  
    "body-parser": "~1.13.2",  
    "cookie-parser": "~1.3.5",  
    "debug": "~2.2.0",  
    "ejs": "~2.3.3",  
    "express": "~4.13.1",  
    "morgan": "~1.6.1",  
    "serve-favicon": "~2.3.0"  
  }  
}
```

# Registro central npm

## ◆ Registro central npm

- Existe un repositorio central de paquetes npm que contiene los paquetes desarrollados por la comunidad
  - Permite **acceso Web** y con **comandos npm**
- Está accesible en: <https://www.npmjs.org>



## ◆ npm publish <paquete>

- Publica paquetes en el **repositorio central**
  - Un paquete publicado en el **registro** está accesible a cualquiera que desee instalárselo
  - Se necesita cuenta en el registro para poder publicar

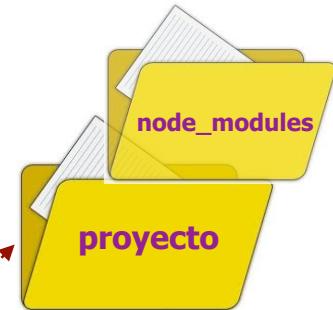
## ◆ npm install <paquete>

- Trae <paquete> del **registro** y lo instala en un **proyecto**
  - Un paquete instalado queda listo para usarse o ejecutarse

## ◆ Modelo de negocio similar a GitHub

- Los paquetes públicos se albergan gratis en el registro central
- los repositorios privados son de pago

# Dependencias de un paquete



## ◆ Dependencias de un paquete npm

- Conjunto de otros paquetes de tipo módulo utilizados por el paquete

## ◆ Instalación de un paquete con: **npm install <paquete>**

- Instala **<paquete>** y dependencias en el directorio **node\_modules**

## ◆ **node\_modules**

- Contiene los paquetes instalados en un proyecto
  - **node\_modules** suele estar en el directorio raíz del proyecto
    - Pero puede estar en cualquier directorio padre del directorio raíz

## ◆ Entorno de un proyecto

- Entorno de producción
  - "**dependencies**": dependencias del despliegue público del servicio
- Entorno de desarrollo
  - "**devDependencies**": dependencias adicionales necesarias durante el desarrollo
- .....

A screenshot of a package.json file. The file contains the following JSON code:

```
{  
  "name": "quiz",  
  "version": "0.0.0",  
  "private": true,  
  "scripts": {  
    "start": "node ./bin/www"  
  },  
  "engines": {  
    "node": "4.2.x",  
    "npm": "2.14.x"  
  },  
  "dependencies": {  
    "body-parser": "~1.13.2",  
    "cookie-parser": "~1.3.5",  
    "debug": "~2.2.0",  
    "ejs": "~2.3.3",  
    "express": "~4.13.1",  
    "express-flash": "0.0.2",  
    "express-partial": "0.0.1",  
    "express-session": "~1.14.0",  
    "method-override": "~2.3.0",  
    "morgan": "~1.6.1",  
    "pg": "^4.4.6",  
    "pg-hstore": "^2.3.2",  
    "sequelize": "^3.19.3",  
    "serve-favicon": "~2.3.0"  
  },  
  "devDependencies": {  
    "sqlite3": "^3.1.1"  
  }  
}
```

Dashed arrows point from the 'node\_modules' folder in the diagram to the 'dependencies' and 'devDependencies' sections of the package.json code.

# Instalar paquetes npm

## ◆ npm install

- Instala todos los **paquetes** para cualquier entorno (dependencies, devDependencies, ..) en **node\_modules**
  - El proyecto que listo para ejecución (una vez realizada la instalación)

## ◆ npm install [--production]

- Instala solo los **paquetes** incluidos en **dependencies** en **node\_modules**
  - El proyecto que listo para ejecución en modo producción

## ◆ npm install <paquete>

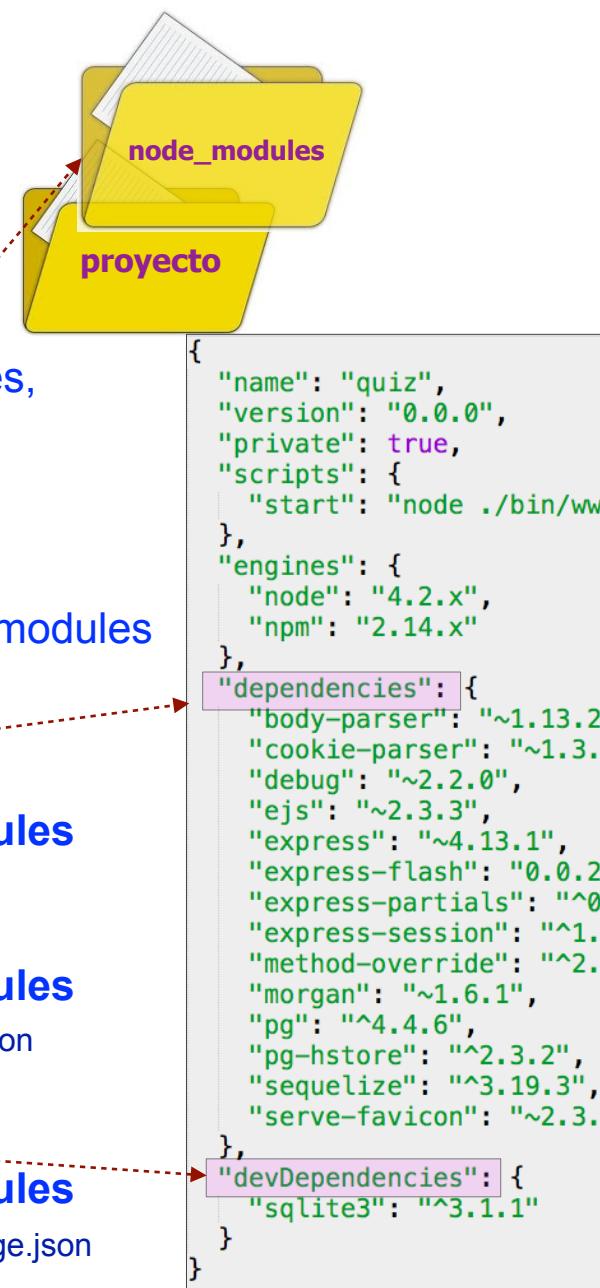
- Trae <paquete> del **registro central** y lo instala en **node\_modules**

## ◆ npm install --save|-S <paquete>

- Trae <paquete> del **registro central** y lo instala en **node\_modules**
  - Además añade el paquete instalado a la lista de **dependencies** de package.json

## ◆ npm install --save-dev|-D <paquete>

- Trae <paquete> del **registro central** y lo instala en **node\_modules**
  - Además añade el paquete instalado a la lista de **devDependencies** de package.json



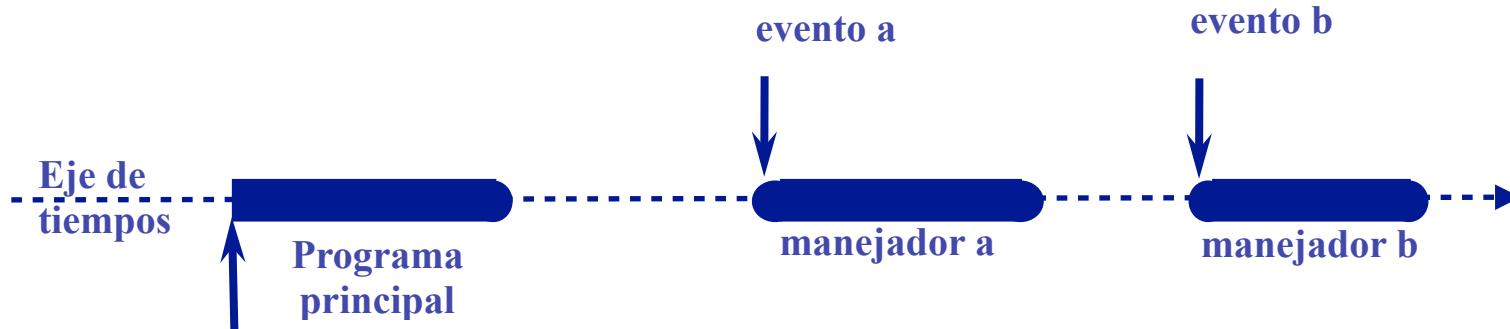


# Timers, eventos, flujos (streams), stdin, stdout y stderr

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Eventos y Manejadores

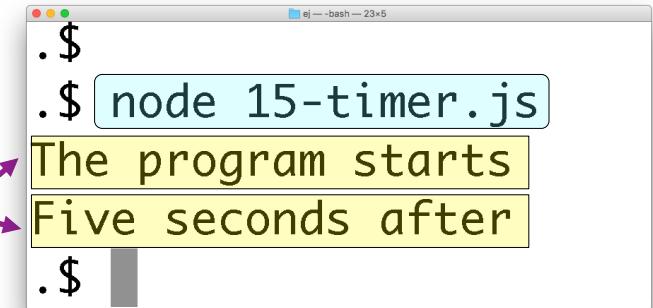
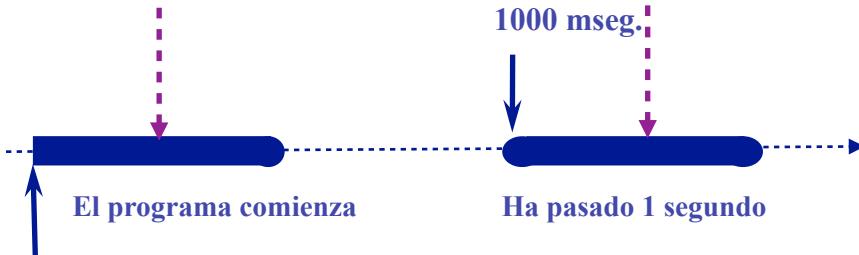
- ◆ El **entorno** interacciona con un programa JavaScript utilizando **eventos**
  - El evento indica al programa que ha ocurrido algo en su exterior
    - ◆ Tipos eventos: temporizadores, click de raton, llegada de datos, ...
- ◆ Un **evento** se atiende con un **manejador** (listener)
  - El manejador es una **función** que se ejecuta al ocurrir el evento
- ◆ La parte inicial del programa configura los manejadores de eventos
  - Después de ejecutar la parte inicial **solo se ejecutan eventos**
    - ◆ Estos pueden programar nuevos eventos, si fuesen necesarios



# setTimeout de node.js

- ◆ La función **setTimeout(..)** de node.js es similar a la del navegador Web
  - Configura un evento interno, que al ocurrir ejecuta el manejador asociado

```
setTimeout(  
  function(){  
    console.log("Five seconds after");  
  },  
  5000  
);  
  
console.log("The program starts");
```



`setTimeout(manejador, milisegundos)`

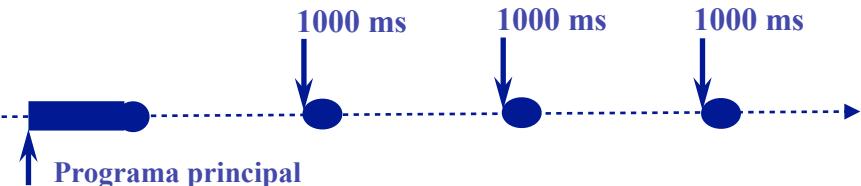
ejecutará manejador() al cabo del tiempo dado en milisegundos.

El **manejador** se crea con un **literal de función** directamente en el primer parámetro de **setTimeout(...)**.

# Eventos: Clase EventEmitter

- ◆ Todas las clases que emiten eventos derivan de **EventEmitter**
  - Heredan métodos: **on(...)**, **addListener(...)**, **removeListener(...)**, **emit(...)**, ...
    - Documentación en: <https://nodejs.org/api/events.html>
- ◆ Un manejador (callback) se define con el método **on** (o **addListener**)
  - **obj.on('event', function (params) {.. < código de manejador > ..})**
    - El manejador del evento se añade al objeto y a partir de ese momento lo atenderá cuando ocurra
      - El método **on(<event>, <listener>)** es equivalente a **addListener(<event>, <listener>)**
      - **removeListener(<event>, <listener>)** desinstala el manejador <listener>
- ◆ El método **obj.emit(<evento>, <p1>, <p2>, ...)**
  - envia **<evento>** al objeto **obj**, pasando los parámetros **<p1>, <p2>, ...** al manejador
    - El evento se atenderá por un manejador de dicho objeto, si existe y no afecta a otros objetos.
- ◆ Un evento tiene por lo tanto 3 elementos asociados
  - **nombre**, **manejador** (callback) y **objeto** asociado

# Ejemplo con evento



## ◆ El ejemplo crea la clase **MyEmitter** derivada de **EventEmitter**

- Y añade un manejador del **evento 'event'** al objeto **myEmitter** de la clase
  - El manejador envía un mensaje a consola cada vez que ocurre el evento

## ◆ Además se genera un evento periódico interno con **setInterval()**

- La función asociada emite el **evento 'event'** sobre el objeto **myEmitter**

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {};
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
setInterval( ()=>myEmitter.emit('event'), 1000);
```

Importar el módulo **events**

Se crea la clase **MyEmitter** que deriva de la clase **EventEmitter**.

Se crea el objeto **myEmitter** de la clase **MyEmitter**.

Se instala un manejador del **evento 'event'** en el objeto **myEmitter** que envía un mensaje a consola con cada evento.

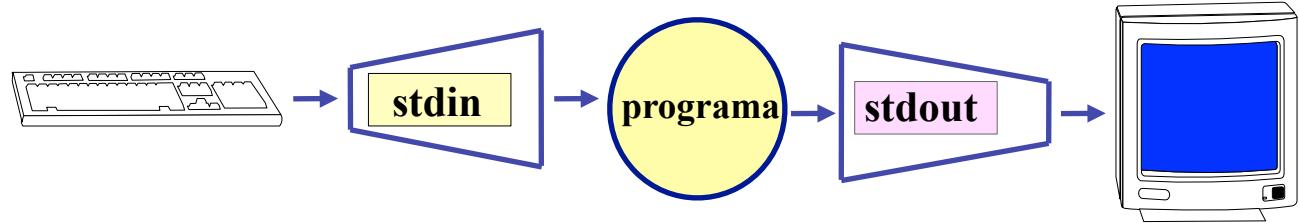
**setInterval(..)** programa un **evento interno periódico** que ejecuta la función y envía el **evento 'event'** cada 1000 ms.

\$ node 15-timer\_obj.js  
an event occurred!  
an event occurred!  
an event occurred!  
an event occurred!  
an event occurred!

# Modulo stream de node.js

- ◆ **stream** define una interfaz genérica para gestionar flujos de datos
  - Asociados a ficheros, a consola, a transacciones HTTP, a circuitos virtuales, etc.
    - ◆ Suelen ser secuencias de octetos binarios o strings
- ◆ **Stream** deriva de eventEmitter y puede utilizar eventos
  - Modulo Stream: <https://nodejs.org/api/stream.html>
- ◆ Clase **stream.Readable** (flujo de entrada)
  - Eventos:
    - ◆ 'data' (llegada de datos), 'end' (final de flujo), 'close' (cierre de flujo), ...
  - Métodos:
    - ◆ setEncoding([encoding]), pause(), resume(), destroy(), ...
- ◆ Clase **stream.Writable** (flujo de salida)
  - Eventos:
    - ◆ 'pipe', 'drain', 'error' , 'finish', ...
  - Métodos (son bloqueantes):
    - ◆ write(string, [encoding], [fd]), write(buffer), end(), .., destroy(), ...

# E/S

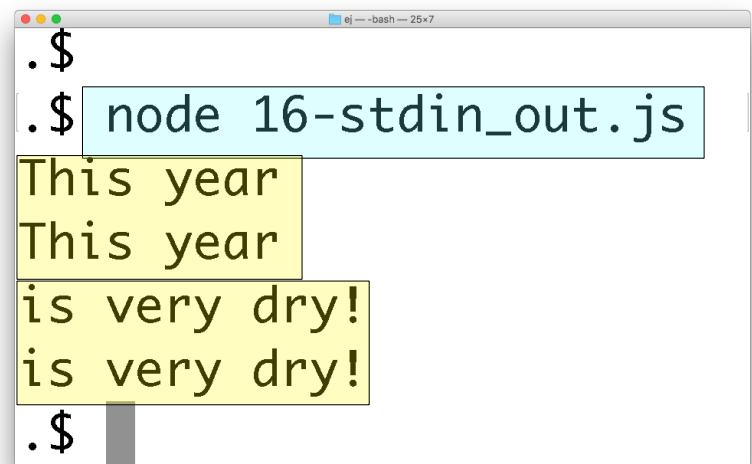


## ◆ El módulo **process** incluye los streams de acceso a la E/S estándar

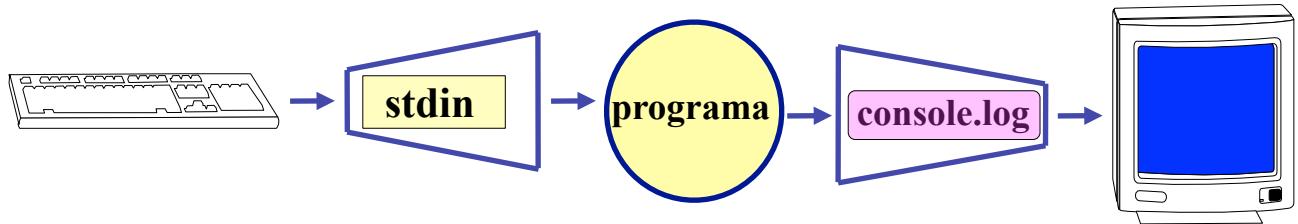
- **stdin**: entrada estándar (teclado), que recibe líneas tecleadas con el evento **data**
  - stdin se para con `pause()` y arranca con “`resume()`”
  - El flujo de entrada se cierra desde el teclado con `^D` o `^C` y desde programa con `close()`
- **stdout**: salida estándar asignada a pantalla
- **stderr**: salida de error asignada a pantalla

```
// Input characters interpreted in UTF-8
process.stdin.setEncoding('utf8');

// Event listener for 'data'
// -> receives input lines
process.stdin.on('data', function(line) {
  process.stdout.write(line);
});
```



# console.log

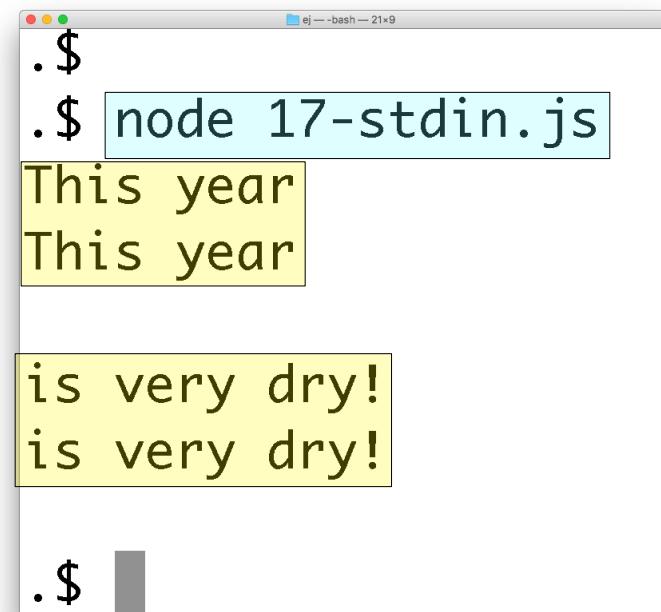


## ◆ console.log(...)

- método de escritura en consola que formatea la salida
  - Más amigable que “process.stdout.write()”
- “process.stdout.write()” no formatea la salida

```
// Input characters interpreted in UTF-8
process.stdin.setEncoding('utf8');

// Event listener for 'data'
// -> receives input lines
process.stdin.on('data', function(line) {
  console.log(line);
});
```



# Ejemplo de un reloj

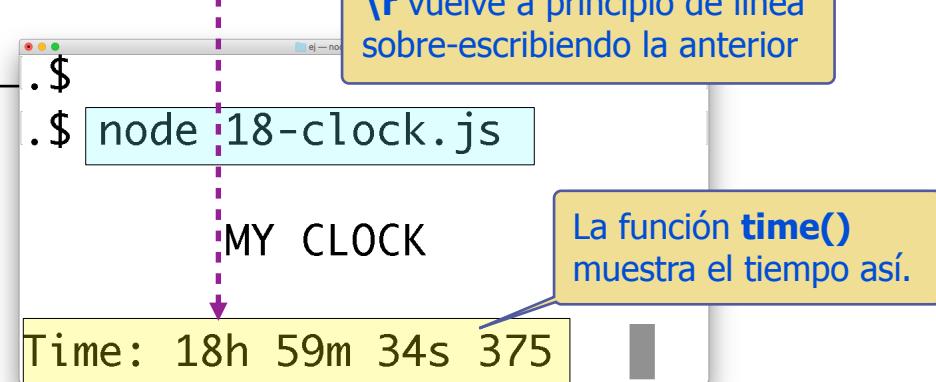
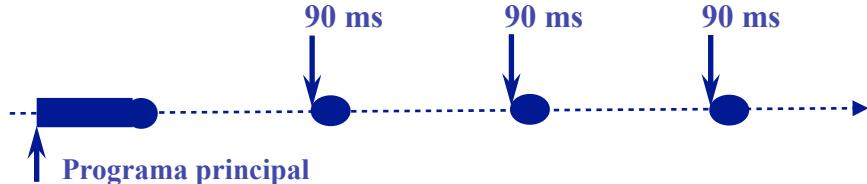
- ◆ El ejemplo genera un evento periódico interno con **setInterval()**
  - `process.stdout.write(...)` deja el cursor al final de la línea mostrada
    - ◆ `\r` (retorno de carro) lleva el cursor a principio de línea para sobre-escribir la anterior

```
function time(){  
  let t = new Date();  
  return "Time: " + t.getHours() + "h " + t.getMinutes() + "m "  
    + t.getSeconds() + "s " + t.getMilliseconds();  
}
```

`setInterval(..)` programa un evento periódico que ejecuta la función cada 90 ms

```
setInterval(() => process.stdout.write('\r' + time() + " "), 90)
```

```
console.log("\n      MY CLOCK\n");
```



# Modulo Readline



- ◆ El módulo readline crea interfaces de línea para flujos de entrada
  - Facilita la programación de interfaces de comando
    - ◆ <https://nodejs.org/api/readline.html>
- ◆ Las nuevas líneas tecleadas se reciben con el evento **line**

const readline = require('readline');

```
const rl = readline.createInterface({  
    input: process.stdin,  
    output: process.stdout,  
    prompt: '\nType something: '  
});  
  
rl.prompt();  
  
rl.on('line', (line) => {  
    console.log(`Did you type '${line}'`);  
    rl.prompt();  
}).on('close', () => {  
    console.log('\nHave a great day!');  
    process.exit(0);  
});
```

**readline.createInterface(..) crea la interfaz rl.**

El objeto `{input: .., output: .., prompt: ..}` configura el interfaz con `process.stdin` como flujo de entrada, `process.stdout` como flujo de salida y `'\nType something: '` como saludo (prompt).

**rl.prompt()** envia el "prompt" al flujo de salida (stdout).

**line:** indica nueva línea tecleada

**close:** indica cierre de stream.

**rl.prompt()**

\$ node 18-readline.js

Type something: Hi  
Did you type 'Hi'

Type something: Hello  
Did you type 'Hello'

Type something:  
Have a great day!

\$



# Ficheros: readFile, writeFile, appendFile, readStream, writeStream, pipe, ..

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Módulo fs: File System

## ◆ El módulo fs: File System

- Da acceso al sistema de ficheros del sistema operativo (p.e. UNIX)
  - Documentación: <http://nodejs.org/api/fs.html>
- Sus métodos y eventos permiten acceder a
  - Ficheros: **open**, **read**, **write**, **append**, **rename**, **close**, ...
  - Directorios: **readdir**, **rmdir**, **exists**, **stats**, ....
  - Gestionar permisos: **chown**, **chmod**, **fchown**, **lchown**, ...
  - Enlaces simbólicos: **link**, **symlink**, .....
  - .....

## ◆ Hay que importar el módulo antes de utilizarlo con

- **require('fs')**

# Ejemplo de lectura de un fichero

El programa importa el paquete fs de node.

```
var fs = require('fs');
```

35-file.js

Se invoca **fs.readFile(<file>, <format>, <callback>)** que da la orden lectura del fichero **<file>** con formato **<format>** e instala el manejador **<callback>** muestra por consola el fichero cuando se haya leido.

```
fs.readFile('35-file.js',
  'ascii',
  function(err, data){ console.log(data)}
```

El manejador es una función que se asocia al evento de final de lectura. La función se ejecuta cuando ocurre el evento.



```
$
$ node 35-file.js
var fs = require('fs');

fs.readFile('35-file.js',
  'ascii',
  function(err, data){ console.log(data)}
```

# Ejemplo: Copy



```
var fs = require('fs'); // Imports file system module
```

```
if (process.argv.length != 4){ // Wrong parameters?  
  console.log('  syntax: "node copy <orig> <dest>"');  
  process.exit() // Finalizes node process  
}
```

El programa importa el paquete fs de node.

```
fs.readFile(  
  process.argv[2], // <orig> file  
  function(err, data) { // callback when read finishes  
    if (err) throw err;  
    fs.writeFile(  
      process.argv[3], // <dest> file  
      data, // <orig> data to be written  
      function (err) { // callback when write finishes  
        if (err) throw err;  
        console.log('  file copied');  
      }  
    );  
  }  
);
```

A continuación se invoca `fs.readFile(<file>, <callback>)` que da la orden lectura del fichero `<file>` e instala el manejador `<callback>` para que procese el fichero cuando se haya leido.

El manejador se invoca al finalizar la lectura. Si no hay errores (`err` se evalúa a `false`), la lectura ha sido exitosa y el contenido estará en `data`.

El manejador (`<callback>`) comprueba que no hay error y ordena la escribir el contenido en el fichero destino, instalando un manejador que indicará "file copied" si no ha habido errores.

```
$ node 40-copy.js  
syntax: "node copy <orig> <dest>"  
$ node 40-copy.js xx.txt ss.txt  
file copied
```

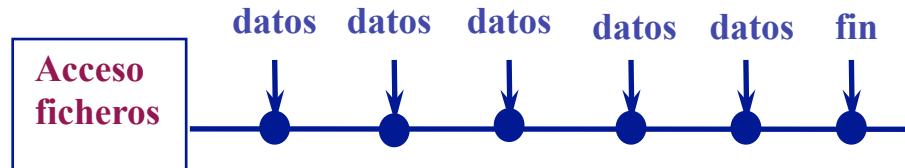
# Ordenación de Callbacks en serie

- ◆ La anidación de callbacks garantiza orden de ejecución
  - Es un patrón de programación muy habitual en node.js

```
fs.readFile(  
  process.argv[2],          // <orig> file  
  function(err, data) {    // callback when read finishes  
    if (err) throw err;  
    fs.writeFile(  
      process.argv[3],       // <dest> file  
      data,                 // <orig> data to be written  
      function (err) {       // callback when write finishes  
        if (err) throw err;  
        console.log('  file copied');  
      }  
    );  
  }  
);
```



# Copy con pipe



## ◆ Los streams permiten acceder a los ficheros por bloques de datos

- El método **pipe(..)** de **fs** realiza la copia con mayor paralelismo
  - ◆ Lee bloques del fichero origen y los escribe en el destino a medida que llegan del disco
  - No espera a leer el fichero completo para escribir (como en el caso anterior)

```
var fs = require('fs');

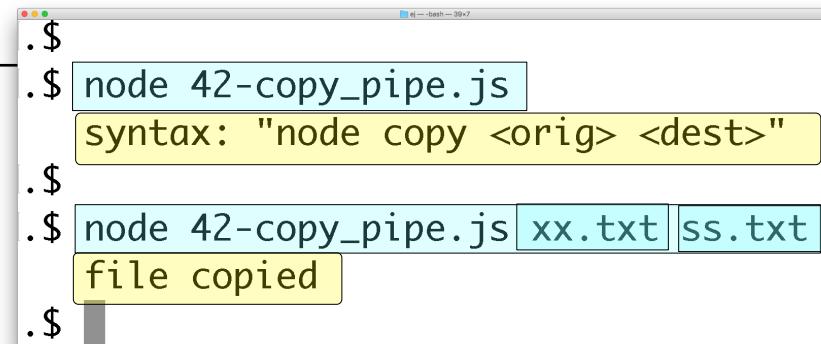
if (process.argv.length != 4){ // Wrong parameters?
  console.log('  syntax: "node copy <orig> <dest>"');
  process.exit() // Finalizes node process
}


```

```
var readStream = fs.createReadStream(process.argv[2]); // Open read stream
var writeStream = fs.createWriteStream(process.argv[3]); // Open write stream
```

```
// Connects input & output with pipe, finishes when input stream finishes
readStream.pipe(writeStream);
```

```
console.log('  file copied');
```



# Ejemplo: Append



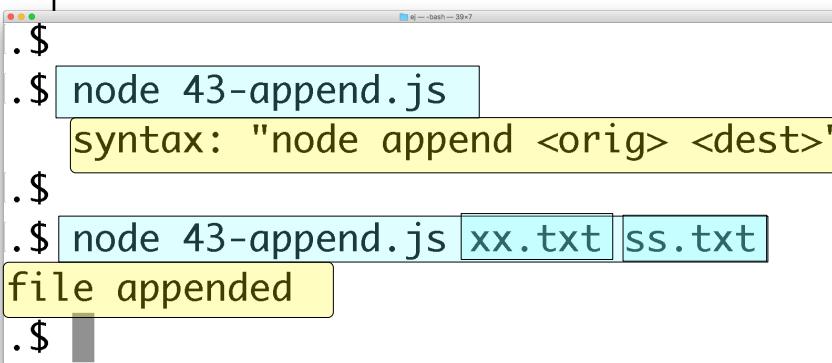
```
var fs = require('fs');           // Imports file system module

if (process.argv.length != 4){   // Wrong parameters?
  console.log('  syntax: "node append <orig> <dest>"');
  process.exit()                // Finalizes node process
}

fs.readFile(
  process.argv[2],               // <orig> file
  function(err, data) {          // callback when read finishes
    if (err) throw err;
    fs.appendFile(
      process.argv[3],           // <dest> file
      data,                      // <orig> data to be written
      function (err) {            // callback when write finishes
        if (err) throw err;
        console.log('file appended');
      }
    );
  }
);
```

Los mensajes también se modifican.

Programa similar a copy que utiliza el método **appendFile (...)** de node en vez de **writeFile (...)**



Los mensajes también se modifican.

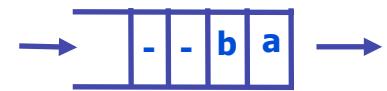


# Gestión de la concurrencia, bucle de eventos y nextTick

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# El bucle de eventos

Cola de Eventos

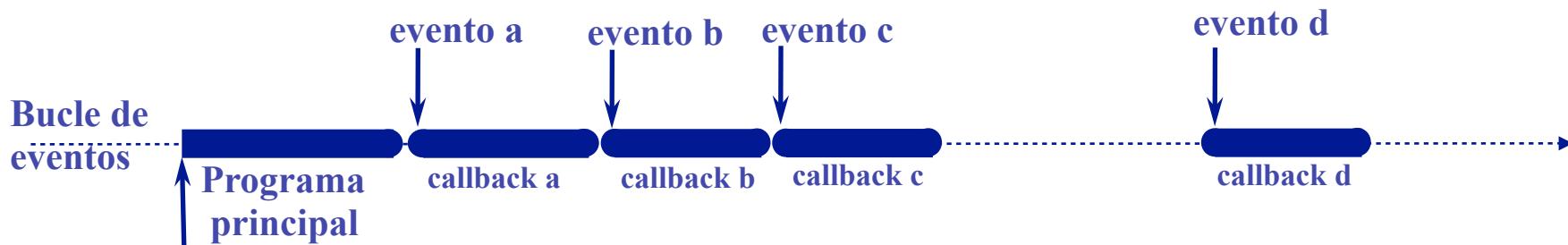


## ◆ node.js se ejecuta en un único hilo (**thread**) de ejecución

- Al arrancar el proceso, el hilo ejecuta primero el **programa principal**
  - Después **atiende a los eventos** que llegan a la **cola de eventos**
    - Ejecutando sus **manejadores** (o callbacks)

## ◆ node.js no consume recursos extra mientras no hay eventos que atender

- El resto de actividades del S.O. se puede ejecutar sin problemas
  - node finaliza cuando no hay ningún manejador (callback) de evento programado



# Bucle de eventos y nextTick()

## ◆ nextTick(<callback>)

- Método de **process**
  - Introduce <callback> adelantando a los otros eventos
- **nextTick:** estrategia FIFO

## ◆ setTimeout() con retardo “0”

- entra en cola inmediatamente

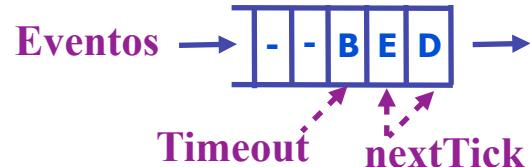
```
5->setTimeout(function() { console.log('Event A'); }, 5);
4->setTimeout(function() { console.log('Event B'); }, 0);

2->process.nextTick(function() { console.log('Tick D'); });
3->process.nextTick(function() { console.log('Tick E'); });

1->console.log('End of Main Program');
```

```
.$
.$ node 19-nextTick.js
1->End of Main Program
2->Tick D
3->Tick E
4->Event B
5->Event A
.$
```

Cola de Eventos después de ejecutar el programa principal



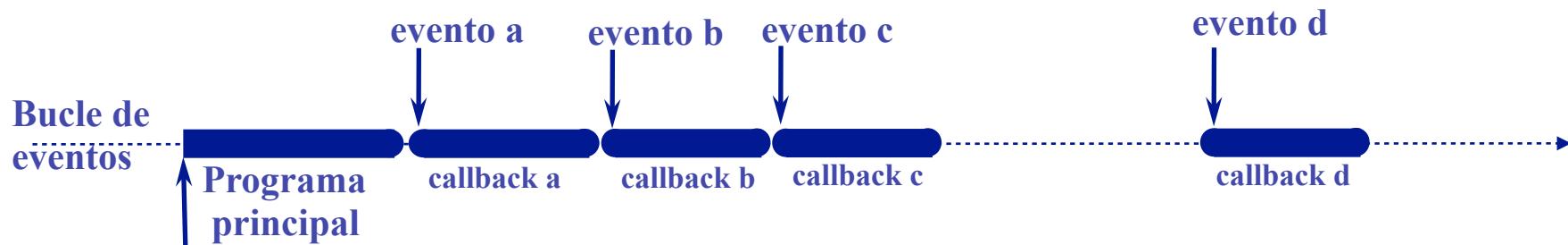
# node.js garantiza exclusión mutua

## ♦ node es muy sencillo de programar

- Los manejadores de eventos se ejecutan en serie

## ♦ La gestión de la cola de eventos

- garantiza exclusión mutua en el acceso a variables y objetos
  - No se necesitan mecanismos de exclusión mutua: zonas críticas, monitores, ...



# Bloqueo en node.js

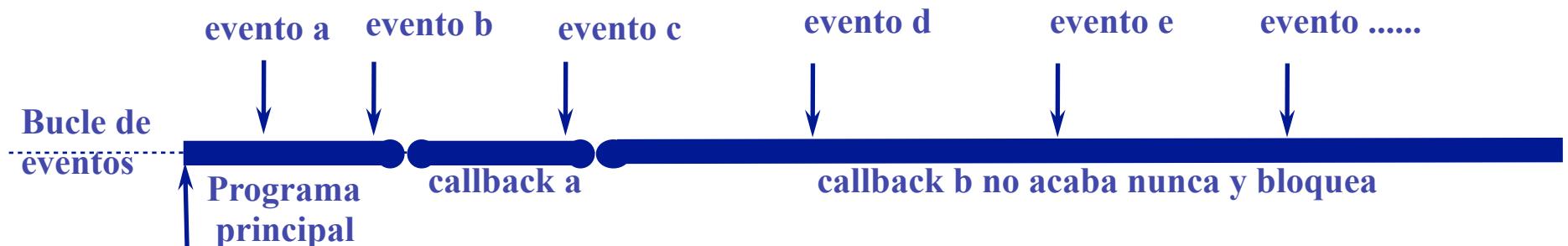


## ◆ Bloqueo

- **Problema importante** de la programación concurrente
  - Un programa, o parte de él, **deja de ejecutarse**, esperando que otro acabe

## ◆ Programa principal y manejadores de node.js

- pueden bloquear al resto **solo por inanición ("starvation")**
  - Si un manejador **no finaliza**, no se atienden mas eventos y el servidor **se bloquea**
- Un manejador debe **finalizar rápidamente** para que node.js atienda los siguientes eventos **lo antes posible**





# Excepciones, errores y sentencias: throw y try...catch...finally

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Excepciones y sentencia throw

- ◆ Una **excepción** es una señal que puede interrumpir la ejecución de un programa
  - La señal (excepción) se lanza con la sentencia **throw <msj>**
    - ◆ Doc: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw>
- ◆ La excepción lleva un valor asociado que se utiliza para identificarla
  - Que se puede utilizar para identificar la excepción ocurrida
    - ◆ Ejemplo: **throw "Abort execution"**

**throw "Abort execution"** aborta la ejecución.  
La siguiente instrucción no se ejecuta.

```
.$  
.$ node 20-excep.js  
This msg WILL be shown
```

```
/Users/jq/ej/20-excep.js:4  
throw "Abort execution";  
^  
Abort execution  
. $
```

```
console.log("This msg WILL be shown");
```

```
throw "Abort execution";
```

```
console.log("This msg WON'T be shown");
```

El programa finaliza sin haber ejecutado la última instrucción. Indica que ha ocurrido una excepción y muestra el mensaje de la excepción.

# Errores

## ◆ Los errores son excepciones con un valor de la clase predefinida Error

- Los errores se lanzan también con la sentencia **throw**
  - ◊ Ejemplo: `throw new Error("Se aborta la ejecución")`
- Documentación de la clase **Error**
  - ◊ [http://www-db.deis.unibo.it/courses/TW/DOCS/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Error.html](http://www-db.deis.unibo.it/courses/TW/DOCS/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error.html)

The screenshot shows a terminal window titled "ej -- bash". The command ". \$ node 21-error.js" is run, followed by the output of the script. The script content is shown in a callout box:

```
console.log("This msg WILL be shown");
throw new Error("Abort execution");
console.log("This msg WON'T be shown");
```

A dashed arrow points from the word "throw" in the script to the corresponding line in the terminal output, which shows the error message "Error: Abort execution" and a stack trace.

**Terminal Output:**

```
.$
.$ node 21-error.js
This msg WILL be shown
/Users/jq/ej/21-error.js:4
throw new Error("Abort execution");
^
Error: Abort execution
    at Object.<anonymous> (/Users/jq/ej/21-error.js:4:9)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
    at run (bootstrap_node.js:389:7)
    at startup (bootstrap_node.js:149:9)
    at bootstrap_node.js:504:3
```

El interprete de JavaScript detecta que se ha lanzado un **error** y muestra una traza con datos sobre la sentencia en ejecución al ocurrir el error.

Los números del final indican la línea y el carácter de la línea donde se ha generado el error, así como la pila de invocaciones.

49

© Juan Quemada, DIT, UPM

# Errores de ejecución

- ◆ El interprete JavaScript **analiza** las instrucciones del programa **al ejecutarlo**
  - Si encuentra algún tipo de **error** lanza una excepción con un valor de la **clase Error**
    - El ejemplo invoca una **función no definida**, que lanza un **ReferenceError** (clase derivada de Error)
      - [http://www-db.deis.unibo.it/courses/TW/DOCS/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/ReferenceError.html](http://www-db.deis.unibo.it/courses/TW/DOCS/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError.html)
    - Cada tipo de error está asociado a una clase predefinida: ReferenceError, SyntaxError, RangeError, ....

- ◆ Por ej., el interprete lanza errores al

- invocar funciones no definidas
- invocar métodos no definidos
- utilizar variables no definidas
- detectar errores de sintaxis
- ....

Al ejecutar el programa, el interprete JS detecta que **funcionIndefinida()** no existe y lanza un error.

```
console.log("This msg WILL be shown");

undefinedFunction(); // -> Execution error

console.log("This msg WON'T be shown");
```

The screenshot shows a terminal window titled "ej -- bash -- 69x19". The command ". \$ node 22-error\_func\_indef.js" is run. The output shows:

```
.$
.$ node 22-error_func_indef.js
This msg WILL be shown
/Users/jq/ej/22-error_func_indef.js:4
undefinedFunction(); // -> Execution error
```

Below the terminal, a detailed stack trace of the error is displayed:

```
ReferenceError: undefinedFunction is not defined
at Object.<anonymous> (/Users/jq/ej/22-error_func_indef.js:4:1)
at Module._compile (module.js:570:32)
at Object.Module._extensions..js (module.js:579:10)
at Module.load (module.js:487:32)
at tryModuleLoad (module.js:446:12)
at Function.Module._load (module.js:438:3)
at Module.runMain (module.js:604:10)
at run (bootstrap_node.js:389:7)
at startup (bootstrap_node.js:149:9)
at bootstrap_node.js:504:3
.$
```

# Errores, excepciones y sentencia try...catch...finally

## ◆ Las excepciones y los errores **interrumpen la ejecución** de un programa

- Salvo si se capturan dentro del **bloque try** de la sentencia **try...catch...finally**
  - ◆ **catch** captura excepciones o errores ocurridos dentro de **try**
    - La ejecución continua en el bloque de instrucciones del catch
  - ◆ El bloque **finally** se ejecuta siempre, haya o no excepciones o errores
- Doc: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch>

## ◆ **try...catch...finally** captura cualquier error

- Excepciones o errores del **programa**
  - ◆ p.e., **throw new Error(..)** lanza un error
- Errores lanzados por el interprete

```
.....
try {
    .....
    -> throw "Exception"
    or throw new Error("Error")
    .....
} catch (exception) {
    .....
} finally {
    .....
}
```

# Sentencia try...catch

- Este ejemplo ilustra el uso de la sentencia sin el **bloque finally** (opcional)
  - Al invocar **undefinedFunction()** dentro del bloque **try** se produce un error de JavaScript
    - La ejecución continuará en el **bloque catch**, que recibirá en el parámetro **err** el mensaje de error
    - La instrucción siguiente (a la que provocó el error) del bloque **try** no se ejecutará

```
try {
    console.log("This msg WILL be shown");

    undefinedFunction(); // -> execution error

    console.log("This msg WON'T be shown");

} catch (err) {
    console.log('ERROR CAPTURED: \n -> ' + err);
};

console.log("This msg WILL be shown");
```

```
$ node 24-try_catch.js
This msg WILL be shown
ERROR CAPTURED:
-> ReferenceError: undefinedFunction is not defined
This msg WILL be shown
$
```

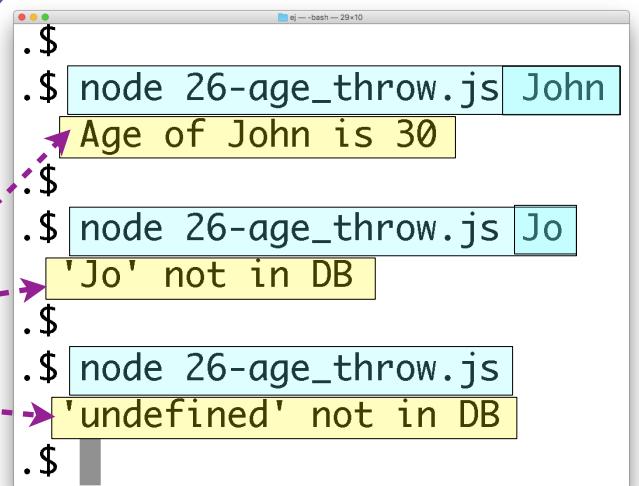
# Ej. de captura de excepción: 26-age\_throw.js

- ◆ Programa similar al anterior que evita que se produzca el error
  - `find(...)` busca en el array y devuelve el **registro de la persona** o sino **`undefined`**
    - ◆ La sentencia `if` detecta si el resultado de la búsqueda es **`undefined`** y lanza una excepción
      - Así se evita que `person.name` o `person.age` provoquen un error de ejecución

```
try {  
  let people = [{name:'Peter', age:22},  
                {name:'Anna',   age:23},  
                {name:'John',   age:30}];  
  
  let person = people.find(p => p.name === process.argv[2]);  
  
  if (!person) { throw "  " + process.argv[2] + " not in DB";}  
  
  console.log("  Age of " + person.name + " is " + person.age);  
}  
catch (exception) { console.log(exception); }
```

Lanza excepción.

Si el array **people** no contiene el nombre indicado, la búsqueda devuelve **`undefined`**. La sentencia `if` lo detecta y lanza una excepción.



The terminal window shows the following interactions:

- \$ node 26-age\_throw.js John  
Age of John is 30
- \$ node 26-age\_throw.js Jo  
'Jo' not in DB
- \$ node 26-age\_throw.js undefined  
'undefined' not in DB

# Ejemplo de captura de error: 25-age\_error.js

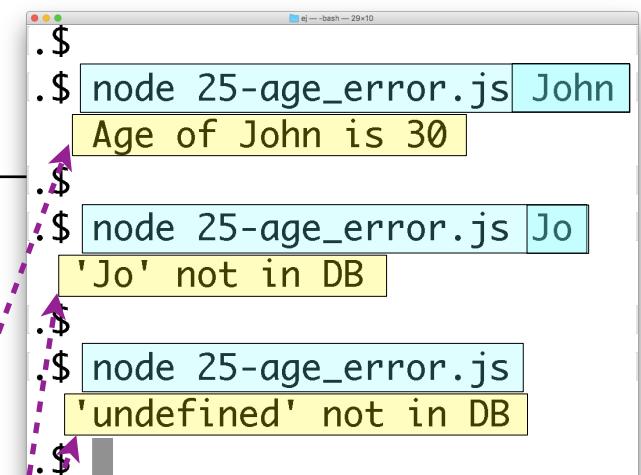
## ◆ El ejemplo: node 25-age\_error.js <nombre>

- Busca la persona indicada en el primer parámetro en el array people
  - ◆ Si la persona existe devuelve **su registro** y sino devuelve **undefined**
    - **undefined.person** provoca un error de ejecución, por lo que se comprueba

Si el array **people** no contiene el nombre indicado, la búsqueda devuelve **undefined** y **person.name** o **person.age** lanzará un **error**.

```
try {  
  let people = [{name:'Peter', age:22},  
                {name:'Anna',  age:23},  
                {name:'John',   age:30}];  
  
  let person = people.find(p => p.name === process.argv[2]);  
  
  console.log(" Age of " + person.name + " is " + person.age);  
}  
} catch (err) {console.log(" '" + process.argv[2] + "' not in DB");}
```

Lanza Error.



The terminal window shows three separate executions of the script:

- Execution 1: \$ node 25-age\_error.js John  
Output: Age of John is 30
- Execution 2: \$ node 25-age\_error.js Jo  
Output: 'Jo' not in DB
- Execution 3: \$ node 25-age\_error.js undefined  
Output: 'undefined' not in DB

Lanza Error.

© Juan Quemada, DIT, UPM

# uncaught-Exception

```
// Event listener: will capture execution error
process.on('uncaughtException', function(err) {
  console.log('PROGRAM ABORTED: ERROR:\n  -> ' + err);
});

console.log('This msg will be shown in the console');

undefinedFunction(); // Generates execution error

console.log('This msg won\'t be shown in the console');
```

## ◆ uncaughtException

- Evento que ocurre cuando una excepción o error aborta el programa
  - ◆ Si se define un manejador, el interprete no envía el mensaje habitual
    - Solo se ejecuta el manejador de uncaughtException

The screenshot shows a terminal window titled "ej -- bash -- 55x6". The user has typed ". \$" followed by ". \$ node 30-uncaughtEx.js". The output of the script is displayed below the command line. It starts with "This msg will be shown in the console", followed by "PROGRAM ABORTED: ERROR:", and then "-> ReferenceError: undefinedFunction is not defined". The entire output of the script is highlighted with a yellow background.

```
ej -- bash -- 55x6
. $
. $ node 30-uncaughtEx.js
This msg will be shown in the console
PROGRAM ABORTED: ERROR:
-> ReferenceError: undefinedFunction is not defined
. $
```



# Promesas: new Promise(..), resolve, reject, then, catch, ....

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Promesas de ES6

- ◆ ES6 incluye una nueva clase llamada **Promise** (Promesa)
  - Una **promesa**: es una tarea que promete generar un valor en el futuro
    - ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)
- ◆ Una promesa tiene tres estados para gestionar este proceso
  - **pendiente**: antes de ejecutar la tarea asociada
  - **cumplida**: la tarea tiene **éxito** y genera el valor prometido
  - **rechazada**: la tarea **falla** y genera un código de rechazo (y no el valor prometido)
- ◆ Las promesas **simplifican** la programación asíncrona
  - Conservan la eficiencia de ejecución de los "callbacks" asíncronos
    - ◆ Permiten separar claramente el **código normal**, del código de **atención a errores**

# Promesas: constructor, resolve y reject

◆ **Promesa:** objeto de la clase **Promise** construido con **new Promise(<ejecutor>)**

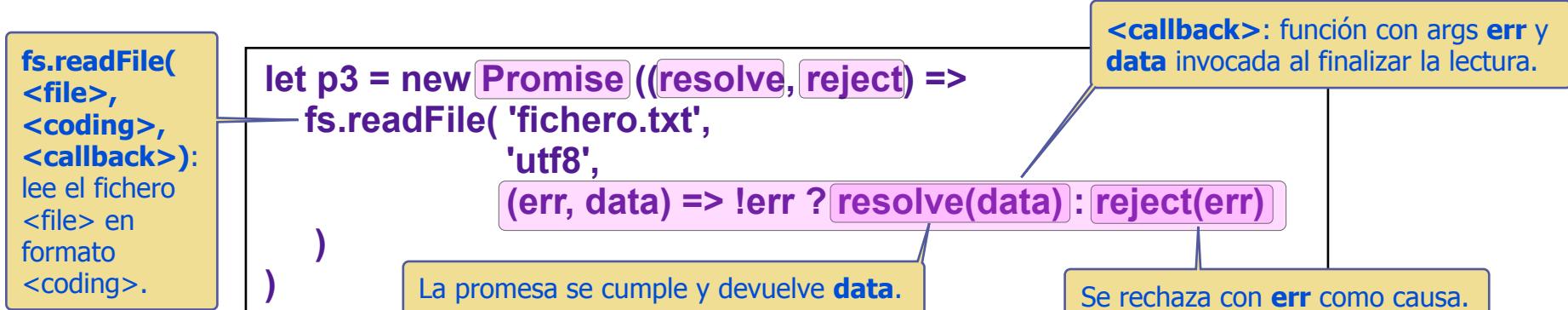
- El **<ejecutor>** es una función **((resolve, reject) => <sentencias>)**, donde
  - **resolve** y **reject** son funciones que deben ser invocadas para cumplir o rechazar la promesa

◆ **Parámetro: resolve(<value>)**

- Hay que llamar a **resolve** para indicar que la promesa ha finalizado **con éxito**
  - **<value>** es el valor generado por la promesa
- **<value>** puede ser de cualquier tipo: string, number, array, object o incluso otra promesa
  - Si **<value>** es una promesa se devolverá el valor de la segunda promesa cuando esta se resuelva
    - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/resolve](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)

◆ **Parámetro: reject(<reason>)**

- Hay que llamar a **reject** para indicar que la promesa se **rechaza**
  - **<reason>** es el código de rechazo generado por la promesa, que suele describir la causa del fallo
    - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/reject](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)



# Promise.resolve(..) y Promise.reject(..)

## ◆ Método de clase: **Promise.resolve(<value>)**

- Crea una promesa que finaliza con **éxito** y genera **<value>**
  - La promesa siempre finalizará con éxito salvo que ocurra un error en la generación de **<value>**
    - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/resolve](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)
- Se utiliza para generar el **primer valor** de una cadena
  - Esta promesa calcula y devuelve **<value>** inmediatamente

## ◆ Método de clase: **Promise.reject(<reason>)**

- **Promise.reject(<reason>)** crea una promesa que siempre se **rechaza** con **<reason>**
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/reject](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)

Ejemplos de promesa sencilla que **siempre se cumple** con **Promise.resolve()** o con constructor.

```
let p2 = Promise.resolve([7, 4, 1, 23]); // equivale a:  
let p1 = new Promise((resolve, reject) => resolve([7, 4, 1, 23]));
```

Ejemplos de promesa sencilla que **siempre se rechaza** con **Promise.reject()** o con constructor.

```
let r2 = Promise.reject("Promesa rechazada"); // equivale a:  
let r1 = new Promise((resolve, reject) => reject("Promesa rechazada"));
```

# Método then

## ◆ <promesa>.then(<manejador\_de\_exito>, <manejador de rechazo>)

- El método **then(..)** recibe el valor de éxito o rechazo generado por <promesa>
  - Invocará el **manejador de éxito o rechazo** que corresponda al resultado de <promesa> cuando esta se resuelva
    - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/then](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)
- El método **then(..)** devuelve una promesa, por lo que se puede encadenar con otro método **then(..)**

## ◆ El manejador de éxito (**data => <sentencias>**) se invoca si la promesa anterior se cumple

- El manejador recibirá en el parámetro (data) el valor generado por la promesa anterior

## ◆ El manejador de rechazo (**err => <sentencias>**) se invoca si la promesa se rechaza

- El manejador recibirá en el parámetro (err) la razón del rechazo generada por la promesa anterior

Programa completo: lee un fichero y lo muestra por pantalla.

```
const fs = require('fs');

new Promise((resolve, reject) =>
  fs.readFile(
    process.argv[2],
    'utf8',
    (err, data) => !err ? resolve(data) : reject(err)
  )
  .then(data => console.log("FILE:\n" + data),
        err => console.log("ERROR\n: " + err))
```

**then** asocia 2 manejadores. Si la promesa se cumple, el primero muestra el contenido (data), sino el segundo muestra el **error** (err).

Si el fichero existe, muestra su contenido.

```
$ node 50-readF_promise.js xx.txt
FILE:
This is the content of file xx.txt
$ node 50-readF_promise.js
ERROR:
TypeError: path must be a string or Buffer
$
```

Si no se pasa el nombre o el fichero no existe, la promesa se rechaza y se muestra un error.

# Método catch

- ◆ **then(..)** puede invocarse solo con el manejador de éxito: **then(<manejador\_de\_exito>)**
  - En este caso atiende solo al éxito y si recibe un rechazo lo propaga a la siguiente promesa
    - ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/then](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)
- ◆ **catch(..)** es un método complementario de **then(..)**, que solo atiende rechazos y propaga éxitos
  - **then(..)** y **catch(..)** crean promesas que puede encadenarse unas con otras
    - ◆ Esto permite crear cadenas donde la posición en la cadena define el orden de ejecución
- ◆ **catch(..)** se invoca solo con el manejador de rechazo: **catch(<manejador de rechazo>)**
  - **catch(<manejador de rechazo>)** equivale a **then(undefined, <manejador de rechazo>)**
    - ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch)

```
const fs = require('fs');

new Promise((resolve, reject) =>
  fs.readFile(
    process.argv[2],
    'utf8',
    (err, data) => !err ? resolve(data) : reject(err)
  )
  .then(data => console.log("FILE:\n" + data))
  .catch(err => console.log("ERROR\n: " + err))
```

Este ejemplo es **equivalente** al anterior, pero ahora **then** solo incluye el manejador asociado al éxito de la promesa anterior y **catch** incluye el manejador asociado al rechazo.

Si el fichero existe, muestra su contenido.

```
$ node 51-readFCatch.js xx.txt
FILE:
This is the content of file xx.txt
```

Si no se pasa el nombre o el fichero no existe, la promesa se rechaza y se muestra un error.

```
$ node 51-readFCatch.js
ERROR:
TypeError: path must be a string or Buffer
```

# Resolver promesas con return y throw

## ◆ then(<manejador>) o catch(..) se resuelven con éxito invocando return <expr>

- Al invocar **return <expr>** la promesa devuelve <expr> inmediatamente, salvo
  - Si <expr> es una **promesa**, se espera a su resolución y se devuelve su **éxito** o **rechazo**
  - Si el cálculo de <expr> genera algún error o excepción, se devolverá el rechazo asociado
- Para **retornar** de una promesa **desde un "callback"** asociado a algún evento
  - Se debe retornar con **resolve()**, el uso de return no funciona en este caso

## ◆ La promesa se **rechaza** lanzando **errores o excepciones** con throw <code>

- **throw** rechaza la promesa y devuelve la excepción o el error como **código de rechazo**

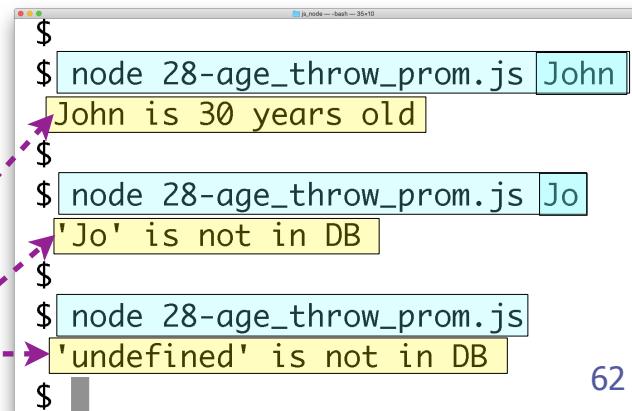
```
Promise.resolve([ {name:'Peter', age:22},  
                 {name:'Anna',  age:23},  
                 {name:'John',   age:30} ]  
)
```

```
.then( people => {  
    let person = people.find(p => p.name === process.argv[2])  
    if (!person) throw " " + process.argv[2] + " is not in DB";  
    return person;  
})
```

```
.then( person =>  
    console.log(" " + person.name + " is " + person.age + " years old")  
)
```

```
.catch( exception => console.log(exception) );
```

**throw " " + process.argv[2] + " is not in DB"**  
**rechaza** la promesa y ejecuta el siguiente **catch** pasando un string como "**'Jo' is not in DB**" como razón de rechazo.



```
$ node 28-age_throw_prom.js John  
John is 30 years old  
$ node 28-age_throw_prom.js Jo  
'Jo' is not in DB  
$ node 28-age_throw_prom.js  
'undefined' is not in DB  
$
```

# Ejemplo de captura de errores de ejecución

- ◆ Una promesa se **rechaza** si se produce un **error de ejecución** cuando se está ejecutando
- ◆ Las expresiones **person.name** y **person.age** pueden provocar un error de ejecución
  - Esto ocurre cuando el parámetro **person** contiene el valor **undefined**
    - El error de ejecución provoca un rechazo de la promesa que es recogido por `catch(..)`

Si el array **people** no contiene el nombre indicado, la búsqueda devuelve **undefined** y **person.name** o **person.age** lanza un **error** y se pasa a ejecutar el siguiente **catch**.

```
Promise.resolve([{name:'Peter', age:22},  
                {name:'Anna',  age:23},  
                {name:'John',   age:30}])  
  .then( people => people.find(p => p.name === process.argv[2]))  
  .then( person =>  
    console.log(" " + person.name + " is " + person.age + " years old")  
  )  
  .catch( err => console.log(" " + process.argv[2] + " is not in DB"));
```

```
$ node 27-age_error_prom.js John  
John is 30 years old  
$ node 27-age_error_prom.js Jo  
'Jo' is not in DB  
$ node 27-age_error_prom.js  
'undefined' is not in DB
```



# Más sobre Promesas: `async`, `await`, `all`, `race` y más ejemplos

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Promesas: `async/await`

- ◆ ES6 incluye la sintaxis **`async/await`** para facilitar el uso de promesas

- Los programas basados en **`async`** y **`await`** son más legibles
    - `async` y `await` solo es azúcar sintáctico, que define y sincroniza promesas de ES6

- ◆ **`async`** es una palabra reservada que se **antepone** a la **definición de una función**

- **`async`** modifica la función para que el **valor de retorno** sea una **promesa**
    - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async\\_function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async_function)

- ◆ **`await`** es una palabra reservada que se **antepone** a una **promesa**

- **`await`** espera a que la promesa se resuelva con éxito o fracaso
    - El **valor de éxito generado** por una promesa precedida por `await` **puede asignarse** directamente a una **variable**
  - **`await`** solo se puede utilizar dentro de una función de tipo **`async`**
    - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>

**60-msg\_prom.js**

```
new Promise(  
  ( resolve, reject ) => resolve("Hi Peter")  
)  
.then( (msg) => console.log("Msg:" + msg));
```

The terminal window shows the command \$ node 60-msg\_prom.js being run. The output is \$ node 60-msg\_prom.js followed by a yellow box containing the text Msg:Hi Peter.

**62-msg\_async\_await.js**

```
async function await_example(){  
  async function message () {  
    return "Hi Peter"  
  }  
  
  let msg = await message();  
  console.log("Msg:" + msg);  
}  
await_example();
```

# Captura del rechazo con try...catch...finally

- ◆ **await** espera a que una **promesa** finalice con éxito o rechazo
  - Es como **then(..)**, espera a que el **valor se genere** o a **propagar el rechazo**, cuando este ocurre
- ◆ El **rechazo** de una promesa puede capturarse con
  - El método **catch(..)** de las promesas, o con
  - La sentencia **try...catch...finally** dentro de una función **async**
- ◆ Los 3 ejemplos siguientes son equivalentes e ilustran la distintas formas de capturar rechazos

51-readF\_catch.js

```
const fs = require('fs');

new Promise((resolve, reject) =>
  fs.readFile(process.argv[2], 'utf8',
    (err, data) => !err ? resolve(data) : reject(err))
)
.then(data => console.log("FILE:\n" + data))
.catch(err => console.log("ERROR:\n" + err))
```

```
$ node 51-readF_catch.js xx.txt
FILE:
This is the content of file xx.txt

$ node 51-readF_catch.js
ERROR:
TypeError: path must be a string or Buffer
```

52-readF\_async\_catch.js

```
const fs = require('fs');

async function catch_example(){

  let data = await new Promise((resolve, reject) =>
    fs.readFile(process.argv[2], 'utf8',
      (err, data) => !err ? resolve(data) : reject(err))
  );

  console.log("FILE:\n" + data);
}

catch_example().catch( err =>
  console.log("ERROR:\n" + err));
```

52-readF\_async\_try\_catch.js

```
const fs = require('fs');

async function tryCatch_example(){

  try {

    let data = await new Promise((resolve, reject) =>
      fs.readFile(process.argv[2], 'utf8',
        (err, data) => !err ? resolve(data) : reject(err))
    );

    console.log("FILE:\n" + data);
  } catch (err) {
    console.log("ERROR:\n" + err);
  }
}

tryCatch_example()
```

# Ejemplo: Copy I



```
var fs = require('fs');      // Imports file system module

if (process.argv.length != 4){ // Wrong parameters?
    console.log('  syntax: "node copy <orig> <dest>"');
    process.exit()           // Finalizes node process
}
```

```
fs.readFile(<file> e instala el manejador <callback> p
process.argv[2], // <orig> file
function(err, data) { // callback when read finishes
  if (err) throw err;
  fs.writeFile(
    process.argv[3], // <dest> file
    data, // <orig> data to be written
    function (err) { // callback when write finishes
      if (err) throw err;
      console.log('  file copied');
    }
  );
}
```

El manejador (<callback>) comprueba que no hay error y ordena la escribir el contenido en el fichero destino, instalando un manejador que indicará "file copied" si no ha habido errores.

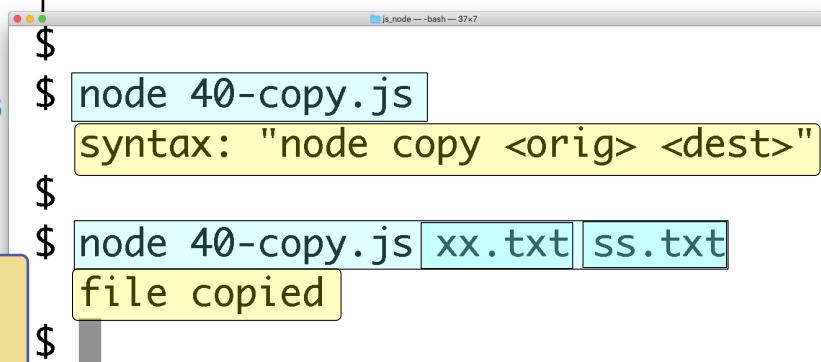
El programa importa el paquete fs de node.

Después comprueba si se han incluido los nombres de los ficheros origen y destino

A continuación se invoca `fs.readFile(<file>, <callback>)` que da la orden lectura del fichero `<file>` e instala el manejador `<callback>` para que procese el fichero cuando se haya leido.

El manejador se invoca al finalizar la lectura. Si no hay errores (**err** se evalúa a **false**), la lectura ha sido exitosa y el contenido estará en **data**.

El manejador (<callback>) comprueba que no hay error y ordena la escribir el contenido en el fichero destino, instalando un manejador que indicará "file copied" si no ha habido errores.



- ◆ Las primera versión del programa **copy** utiliza **promesas**
  - Es equivalente al anterior con callbacks, pero mejor estructurado

- ◆ Las segunda versión mezcla **promesas** con **async/await**
  - Es más legible legible y facil de entender

```
var fs = require('fs');
const orig = process.argv[2], dest = process.argv[2];
```

```
new Promise( (resolve, reject) => (process.argv.length != 4) ? reject(' syntax: "node copy <orig> <dest>"') : resolve() )
  .then( () => new Promise( (resolve, reject) => fs.readFile( orig, 'utf8', (err, data) => err ? reject(err) : resolve(data) )))
  .then( (data) => new Promise( (resolve, reject) => fs.writeFile( dest, data, (err) => err ? reject(err) : resolve('file copied') )))
  .then( (result) => console.log("Result: " + result))
  .catch( (err) => console.log("Error: " + err));
```

```
var fs = require('fs');
const orig = process.argv[2], dest = process.argv[2];
```

```
async function copy_with_await_example() {
  await new Promise( (resolve, reject) =>
    (process.argv.length != 4) ? reject(' syntax: "node copy <orig> <dest>"') : resolve());
  let data = await new Promise( (resolve, reject) => fs.readFile( orig, 'utf8', (err, data) => err ? reject(err) : resolve(data)));
  let res = await new Promise( (resolve, reject) => fs.writeFile( dest, data, err => err ? reject(err) : resolve('file copied')));
  console.log("Result: " + res);
}

copy_with_await_example().catch( (err) => console.log("Error: " + err));
```

## Ejemplo: Copy II

```
$ node 42-copy_prom.js
Error: syntax: "node copy <orig> <dest>"$ node 42-copy_prom.js xx.txt ss.txt
Result: file copied$
```

```
$ node 43-copy_async_await.js
Error: syntax: "node copy <orig> <dest>"$ node 43-copy_async_await.js xx.txt ss.txt
Result: file copied$
```

# Ejemplo: Reflejos

## ◆ Ejemplo de sincronización en el tiempo

- Primera promesa: introduce un retardo aleatorio de 0 a 5 seg.
- Segunda promesa: muestra tiempo transcurrido hasta pulsar return y finaliza

```
async function reflex_example() {          // función encapsuladora

    await new Promise((resolve, reject) => { // introduce un retardo aleatorio de 0 a 5 seg
        let time = (Math.random() * 5000).toFixed(0);
        setTimeout(() => resolve(), time)
    })

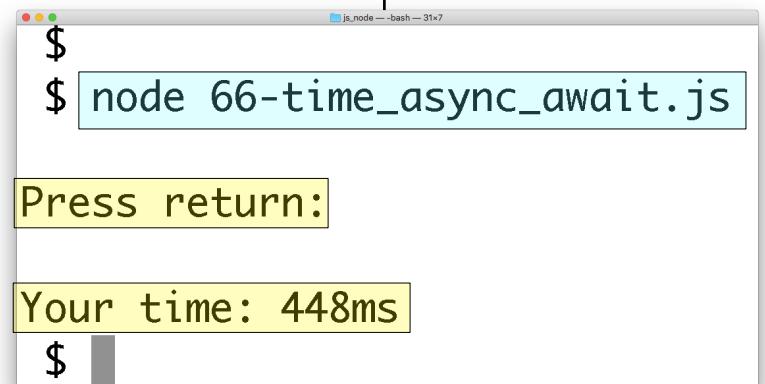
    console.log("\nPress return:");
    let start = new Date().getTime();           // saca mensaje por consola
                                                // guarda el instante actual

    await new Promise((resolve, reject) => {    // espera a se pulse "return"
        process.stdin.setEncoding('utf8');

        process.stdin.once('data', function(line) { // Manejador del evento "pulsar return"
            let time = new Date().getTime() - start;
            console.log("Your time: " + time + "ms");
            resolve();                           // Muestra tiempo transcurrido y finaliza
        })
    })

    process.exit(); // Finaliza programa
};

reflex_example(); // Ejecuta programa
```



# Concurrencia: Promise.all y Promise.race

## ◆ **Promise.all([<p1>, <p2>, .., <pn>])** crea una promesa a partir de N promesas (o valores)

- Finaliza con éxito cuando las N promesas acaban con éxito, generando un array con los N valores
  - ◆ Si alguna de las N promesas es rechazada antes de acabar, la promesa compuesta se rechaza también
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/all](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

## ◆ **Promise.race([<p1>, <p2>, .., <pn>])** crea una promesa a partir de N promesas (o valores)

- Finaliza con éxito en cuanto finaliza la primera promesa y genera el valor generado por esta
  - ◆ Si alguna de las N promesas es rechazada antes de acabar, la promesa compuesta se rechaza también
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/race](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)

```
const p = function (name) { // Promesa: retardo aleatorio
  return new Promise((resolve, reject) => {
    let time = (Math.random() * 5000).toFixed(0);
    setTimeout(() => resolve([time + "msec", name]), time);
  });
};
```

```
Promise.all([p("Peter"), p("Anne"), p("John")])
.then( (res) => { console.log(res) });
```

```
$ node 70-promise_all.js
[ [ '3591msec', 'Peter' ],
  [ '1225msec', 'Anne' ],
  [ '4279msec', 'John' ] ]
```

```
const p = function (name) { // Promesa: retardo aleatorio
  return new Promise((resolve, reject) => {
    let time = (Math.random() * 5000).toFixed(0);
    setTimeout(() => resolve(name), time);
  });
};
```

```
Promise.race([p("Peter"), p("Anne"), p("John")])
.then( (n) => { console.log("\nQuickest: " + n) });
```

```
$ node 71-promise_race.js
Quickest: Anne
```



Final del tema  
Muchas gracias!