



Introducción a node.js, a excepciones y a promesas

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

node.js y sockets - Índice

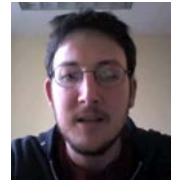
1.	<u>node.js, librería de módulos, entorno de ejecución y modo estricto</u>	<u>3</u>
2.	<u>Módulos node.js: module.exports, require, __dirname y __filename</u>	<u>11</u>
3.	<u>Paquete npm, directorio de un proyecto, package.json y node_modules</u>	<u>20</u>
4.	<u>Timers, eventos, flujos (streams), stdin, stdout y stderr</u>	<u>28</u>
5.	<u>Acceso a Ficheros: readFile, writeFile, appendFile, readStream, writeStream,</u>	<u>40</u>
6.	<u>Gestión de la concurrencia, bucle de eventos y nextTick</u>	<u>49</u>
7.	<u>Errores, excepciones y sentencias throw y try...catch...finally</u>	<u>55</u>
8.	<u>Promesas: new Promise(..), resolve, reject, then, catch</u>	<u>65</u>
9.	<u>Más ejemplos con Promesas</u>	<u>75</u>



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

El comando se
denomina **node** o
nodejs en algunos
UNIX o Linux

```
venus-2:~ jq$ node --help
Usage: node [options] [ -e scri
node debug script.js [ar

Options:
  -v, --version          print no
  -e, --eval script      evaluate
  -p, --print 4          print re
  -i, --interactive      always
```

Documentación 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**: comparision de información
 - ◆

Node.js v8.9.1 Documentation

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

Table of Contents

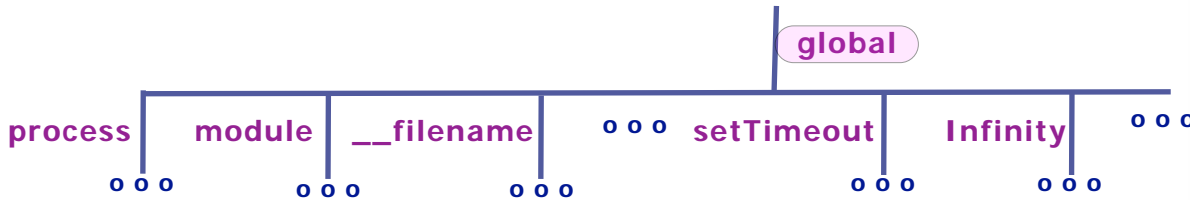
- [About these Docs](#)
- [Usage & Example](#)

- [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

- [Globals](#)
- [HTTP](#)
- [HTTPS](#)
- [Modules](#)
- [Net](#)
- [OS](#)
- [Path](#)
- [Process](#)
- [Punycode](#)
- [Query Strings](#)
- [Readline](#)
- [REPL](#)
- [Stream](#)
- [String Decoder](#)
- [Timers](#)
- [TLS/SSL](#)
- [TTY](#)
- [UDP/Datagram](#)
- [URL](#)
- [Utilities](#)
- [V8](#)
- [VM](#)
- [ZLIB](#)
- [GitHub Repo & Issue Tracker](#)
- [Mailing List](#)

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.

```
. $  
. $ node 01-global.js  
  
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' ]  
. $
```

Muestra las propiedades enumerables de global.

◆ 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

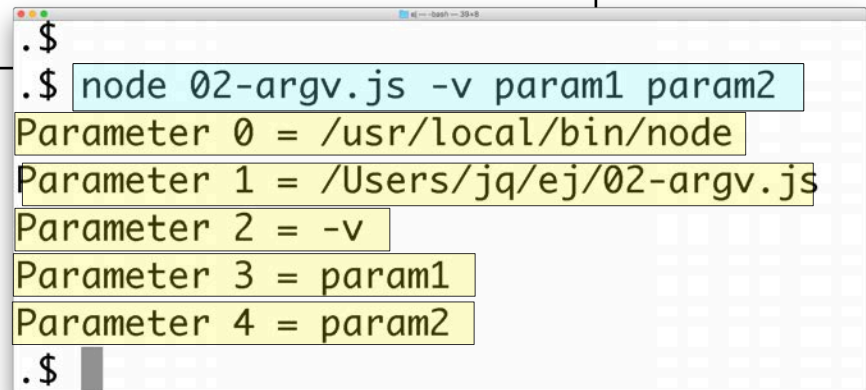
◆ Node se ejecuta en un proceso del S.O. (UNIX, ...)

- **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

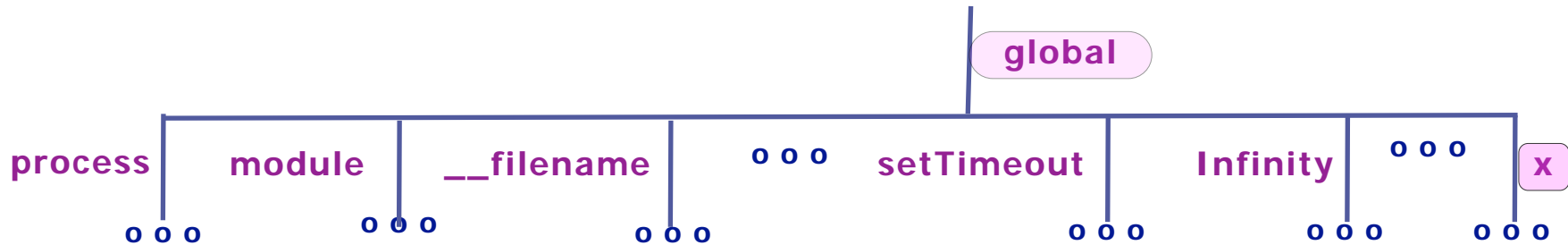
- Permite acceder a ellos desde el programa node.js

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



A terminal window showing the execution of a Node.js script. The prompt is `.$`. The command entered is `node 02-argv.js -v param1 param2`. The output shows five lines of log messages: `Parameter 0 = /usr/local/bin/node`, `Parameter 1 = /Users/jq/ej/02-argv.js`, `Parameter 2 = -v`, `Parameter 3 = param1`, and `Parameter 4 = param2`. The prompt `.$` is shown again at the bottom.

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'

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

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

◆ 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

◆ 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
 - ◆ ...

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
```

◆ 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

Cuestionario

- ◆ Dado el programa node.js **my_prog.js**, que se invoca en la consola con
\$ node my_prog -a 1 2
- ◆ y cuyas 4 primeras líneas son

```
'use strict'
```

```
let x = 1;  
this.y = 2;
```

- ◆ Cual será el resultado de evaluar la siguiente expresión justo después de estas 4 primeras líneas

y;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
y=2;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
z=2;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
x;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
x=2;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
this.y;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
this.x = 1;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
this.y = 3;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
process.argv.length;	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
process.argv[2];	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
process.argv[4];	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución
process.argv[7];	=> 1, "1", 2, "2", 3, "3", 4, "4", 5, "5", "-a", undefined, ErrorDeEjecución



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

◆ 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

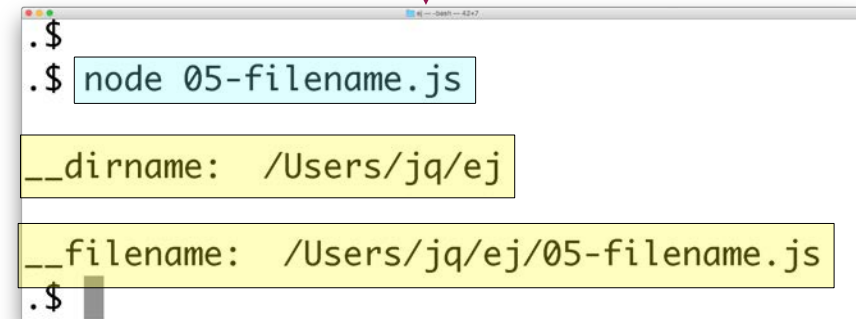
◆ Todos los módulos tienen definidos 5 propiedades locales

- **__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

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

◆ 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**
 - ◆ ...



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

◆ 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>

Ejemplo de módulos de node.js

06-main.js

07-circle.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,
    Mary: 978512355,
    John: 957845123
  });
friends.add("Mary", 978512355);
```

```
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());
```

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

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

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.

Cuestionario

- ◆ La primera línea del programa **my_prog.js** es: **let x = require("./my_mod.js")**. Si invocamos **pwd** en la consola con este resultado y luego este programa

```
$ pwd
```

```
/Users/eva/
```

```
$ node my_prog
```

- ◆ y si en el mismo directorio donde está **my_prog.js**, existe otro fichero **my_mod.js**, que solo contiene la línea **"exports.f = x => x + 2;"**, cual será el resultado de evaluar, como segunda instrucción de **my_prog.js**, la expresión

```
process.argv[1];
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
process.argv[2];
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
__dirname;
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
__filename;
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
x.f(1);
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
f(1);
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
x.f(-1);
=> 1, 2, 3, 4, 5, "/Users/eva/my_prog.js", "/Users/eva/my_mod.js", "/Users/eva/", undefined, Error
```



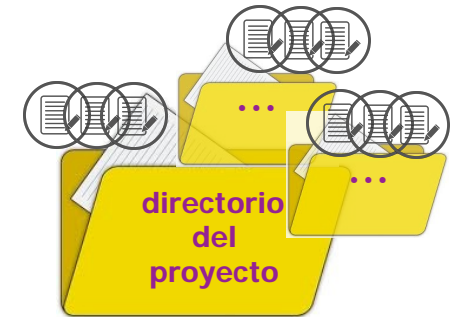
Paquete npm, directorio de un proyecto,
package.json y node_modules

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 módulos 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>

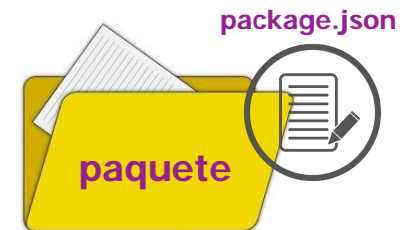
◆ Los paquetes npm son fundamentalmente **programas ejecutables** o **módulos**

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

```
venus:cal_2com jq$
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
-

```
{
  "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"
  }
}
```

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

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>

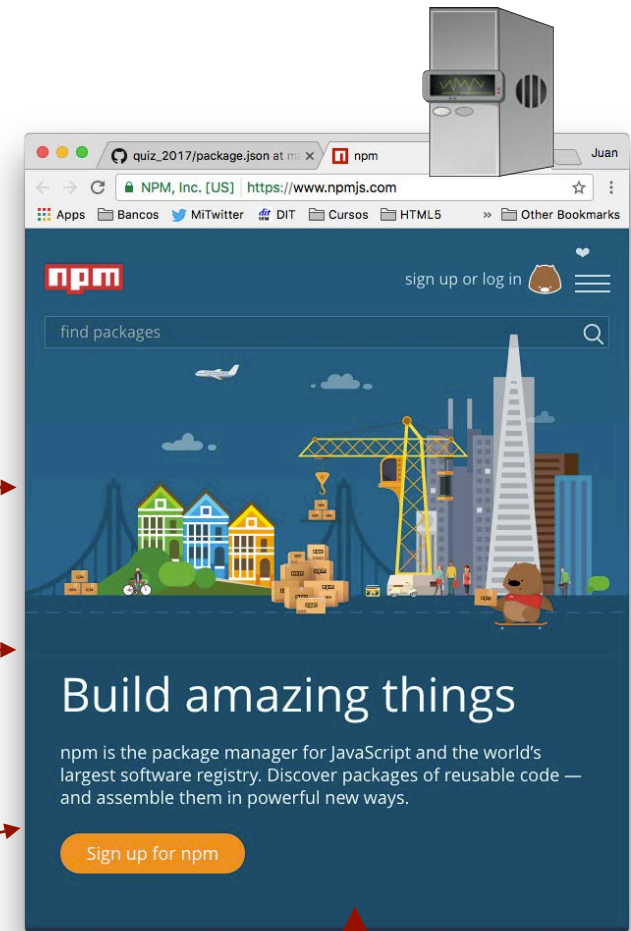
- **Publica** paquetes en el **repositorio central**
 - Un paquete publicado en el **registro** está accesible a cualquiera que desee instalárselo
 - Se necesita cuenta en el registro para poder publicar

◆ npm install <paquete>

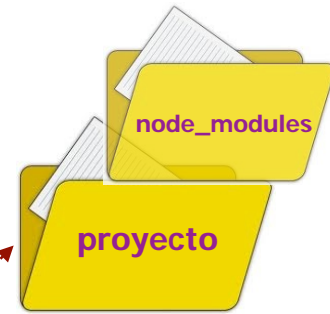
- **Trae** <paquete> del **registro** y lo instala en un **proyecto**
 - Un paquete instalado queda listo para usarse o ejecutarse

◆ 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**

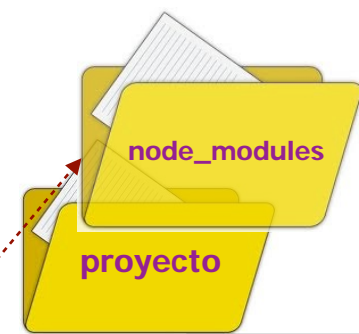
- Contiene los paquetes instalados en un proyecto
 - **node_modules** suele estar en el directorio raíz del proyecto
 - Pero puede estar en cualquier directorio padre del directorio raíz

◆ Entorno de un proyecto

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

```
{
  "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-partials": "^0.
    "express-session": "^1.1
    "method-override": "^2.3
    "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"
  }
}
```

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

```
{
  "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": "~1.3.3",
    "debug": "~2.2.0",
    "ejs": "~2.3.3",
    "express": "~4.13.1",
    "express-flash": "0.0.2",
    "express-partials": "^0.0.4",
    "express-session": "^1.12.1",
    "method-override": "^2.3.5",
    "morgan": "~1.6.1",
    "pg": "^4.4.6",
    "pg-hstore": "^2.3.2",
    "sequelize": "^3.19.3",
    "serve-favicon": "~2.3.2"
  },
  "devDependencies": {
    "sqlite3": "^3.1.1"
  }
}
```

Cuestionario

◆ Que respuesta describe mejor lo siguiente

```
node_modules
bin
test
package.json
npm init
npm install
npm install express
npm install --save express
npm install express@2.1.1
```

POSIBLES RESPUESTAS:

- => comando npm que crea el fichero package.json inicial del proyecto
- => comando npm que instala el paquete del directorio de trabajo
- => Directorio que contiene los paquetes npm instalados en un proyecto
- => Directorio donde están los ficheros ejecutables de un proyecto
- => Directorio donde están los tests de prueba de un proyecto
- => Fichero con la información en formato JSON que permite crear paquetes npm
- => comando npm que solo instala la última versión del paquete express en node_modules
- => comando npm que instala la última versión de express y guarda la dependencia en package.json
- => comando npm que instala la versión 2.1.1 el paquete npm express en node modules

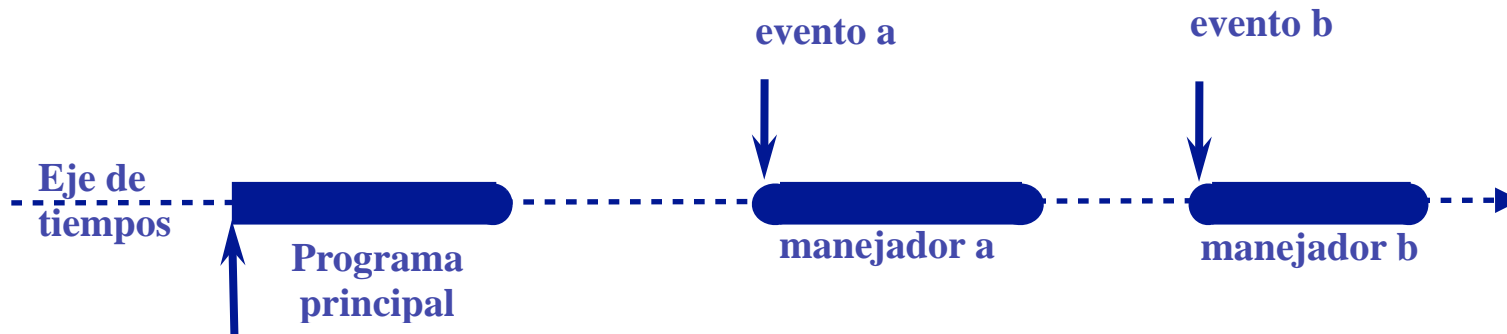


Timers, eventos, flujos (streams), stdin, stdout y stderr

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

Eventos y Manejadores

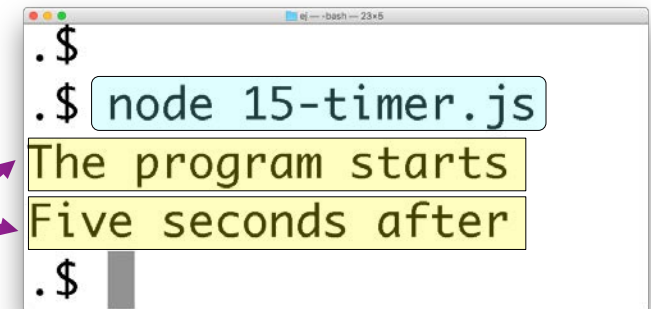
- ◆ El **entorno** interacciona con un programa JavaScript utilizando **eventos**
 - El evento indica al programa que ha ocurrido algo en su exterior
 - ◆ Tipos eventos: temporizadores, click de raton, llegada de datos, ...
- ◆ Un **evento** se atiende con un **manejador** (listener)
 - El manejador 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



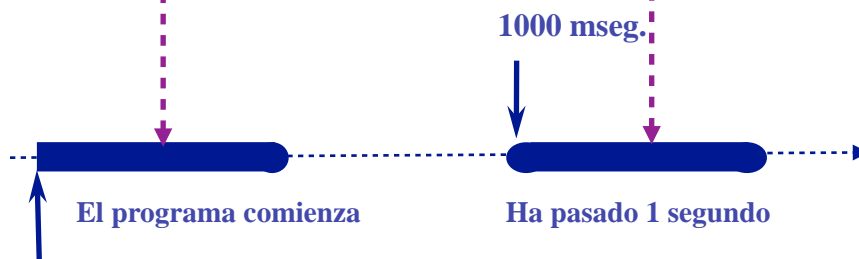
setTimeout de node.js

- ◆ 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");
```



```
. $  
.$ node 15-timer.js  
The program starts  
Five seconds after  
.$
```



`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>

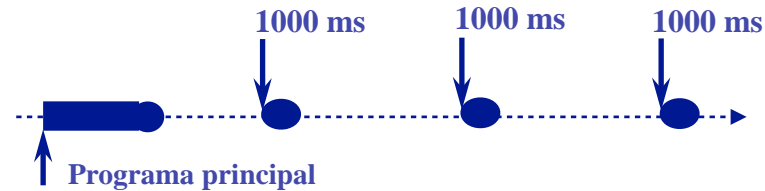
◆ El método **obj.emit(<evento>, <p1>, <p2>, ...)**

- 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: Creación de 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.

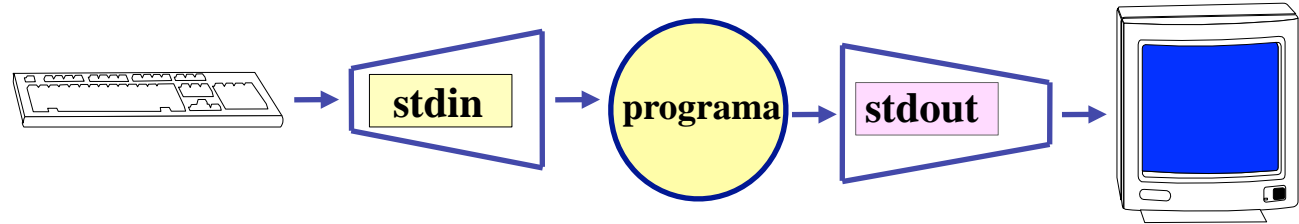
setInterval(..) programa un **evento interno periódico** que ejecuta la función y envía el **evento 'event'** cada 1000 ms.

A terminal window titled 'js_node_ej' showing the command 'node 15-timer_obj.js' and its output: 'an event occurred!' repeated five times. The terminal also shows the prompt '^C' and '\$'.

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

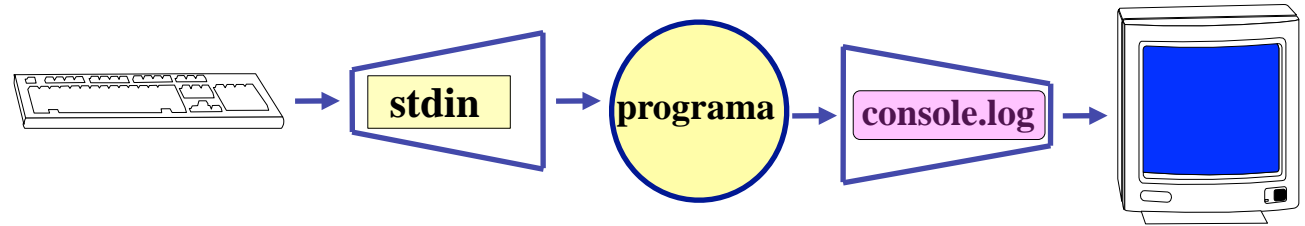
- **stdin**: entrada estándar (teclado), que recibe líneas tecleadas con el evento **data**
 - stdin se para con `pause()` y arranca con “`resume()`”
 - El flujo de entrada se cierra desde el teclado con `^D` o `^C` y desde programa con `close()`
- **stdout**: salida estándar asignada a pantalla
- **stderr**: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

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

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

```
ej --- bash --- 25x7
.$
.$ node 16-stdin_out.js
This year
This year
is very dry!
is very dry!
.$
```

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

The screenshot shows a terminal window with the following content:

```
. $
. $ node 17-stdin.js
This year
This year

is very dry!
is very dry!

. $
```

The output of the script is displayed in yellow boxes, showing two lines of input: "This year" and "is very dry!".

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

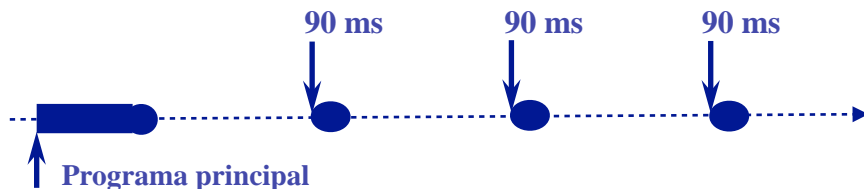
```
function time(){  
  let t = new Date();  
  return "Time: " + t.getHours() + "h " + t.getMinutes() + "m "  
    + t.getSeconds() + "s " + t.getMilliseconds();  
}
```

`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");
```

`\r` vuelve a principio de línea sobre-escribiendo la anterior



```
.$  
.$ node 18-clock.js
```

MY CLOCK

La función `time()` muestra el tiempo así.

```
Time: 18h 59m 34s 375
```

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() envía el "prompt" al flujo de salida (stdout).

line: indica nueva línea tecleada

close: indica cierre de stream.

Cuestionario

◆ Que respuesta describe mejor lo siguiente

```
setTimeout(my_function, t)
setInterval(my_function, t)
events
readline
stream
stdin
stdout
stderr
obj1.addListener('event1', f2)
obj1.removeListener('event1', f2)
obj1.emit('event1', 'p1')
obj2.on('event2', f1)
```

POSIBLES RESPUESTAS:

- => Nombre del modulo de node.js para crear interfaces de entrada de líneas de más alto nivel que stdin
- => Formato de intercambio de datos en forma de flujo de octetos, usado en JavaScript y en otros lenguajes y S.O
- => Nombre de la propiedad de process que contiene el objeto de gestión de la entrada estándar
- => Función JavaScript que lanza un evento que ejecuta my_function() al cabo de t milisegundos
- => Función JavaScript que configura un evento periódico que ejecuta my_function() cada t milisegundos
- => Nombre del modulo de node.js que importa la clase con la interfaz de gestión de eventos
- => Expresión que elimina el manejador (función) f2 del evento 'event1' en el objeto obj1
- => Expresión que envía el evento 'event1' con el parámetro 'p1' al objeto obj1
- => Expresión que añade el manejador (función) f1 del evento 'event2' en el objeto obj2
- => Nombre de la propiedad de process que contiene el objeto de gestión de la salida estándar
- => Nombre de la propiedad de process que contiene el objeto de gestión de la salida de error
- => Expresión que añade el manejador (función) f2 del evento 'event1' en el objeto obj1

Ejercicio opcional

Añadir al ejemplo presentado en la transparencia "Ejemplo: Creación de evento" de este tema lo siguiente:

1. Añadir al objeto myEmitter un segundo manejador de eventos para un evento de nombre "event_2", que muestre por consola el mensaje "a 600ms event_2 occurred". Añadir además un emisor periódico de eventos, como el que ya esta incluido, que emita sobre myEmitter un evento "event_2" cada 600 milisegundos.
2. Añadir al objeto myEmitter un tercer manejador de eventos para un evento de nombre "event_3", que muestre por consola el mensaje "a stdin event_3 occurred: <línea tecleada>". Además habrá que añadir otro manejador asociado al evento "data" de stdin que emita un evento "event_3", que lleve la línea tecleada como parámetro, cada vez que llega una nueva línea por stdin.

El ejemplo de la transparencia "Ejemplo: Creación de evento" está incluido, con el nombre 15-timer_obj.js, en el fichero ZIP con todos los ejemplos de este tema que puede descargarse de la plataforma.



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.

35-file.js

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

```
fs.readFile('35-file.js',  
  'ascii',
```

```
  function(err, data){ console.log(data)}  
);
```

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 leído.

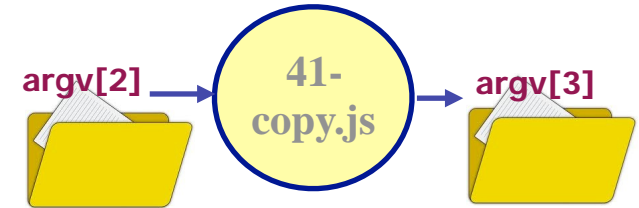
El manejador es una función que se asocia al evento de final de lectura. La función se ejecuta cuando ocurre el evento.

inicia
lectura

finaliza
lectura

```
$  
$ 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
```

El programa importa el paquete fs de node.

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

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

```
fs.readFile(
```

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 leído.

```
process.argv[2], // <orig> file
function(err, data) { // callback when read finishes
```

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**.

```
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.

```
);
}
);
```

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

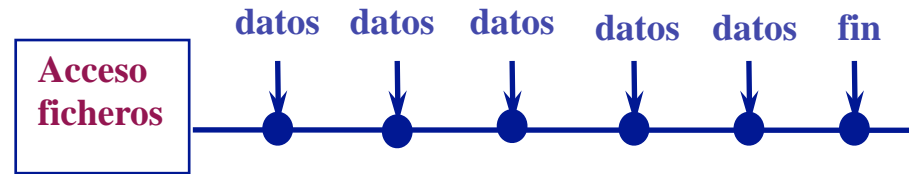
Ordenación de Callbacks en serie

- ◆ 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');
```

```
.$
.$ node 42-copy_pipe.js
    syntax: "node copy <orig> <dest>"
.$
.$ node 42-copy_pipe.js xx.txt ss.txt
    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.

```
.$
.$ node 43-append.js
  syntax: "node append <orig> <dest>"
.$
.$ node 43-append.js xx.txt ss.txt
file appended
.$
```

Cuestionario

◆ Que respuesta describe mejor lo siguiente

```
fs
manejador o callback
stream
pipe
fs.readFile('file1.txt', f1)
fs.writeFile('file1.txt', 'contenido', f2)
fs.appendFile('file1.txt', 'contenido', f2)
fs.createReadStream('file1.txt')
fs.createWriteStream('file1.txt')
```

POSIBLES RESPUESTAS:

- => Formato de intercambio de datos en forma de flujo de octetos, usado en JavaScript y en otros lenguajes y S.O
- => Método que permite conectar flujos de octetos (o streams) de salida con los de entrada
- => Método del módulo fs que lee los datos de un fichero
- => Modulo de node.js, instalado pero no importado por defecto, que permite acceder al sistema de ficheros
- => Función que se ejecuta para atender a un evento o a un cambio del exterior del programa
- => Método del módulo fs que abre un stream de entrada (o de lectura)
- => Método del módulo fs que abre un stream de salida (o de escritura)
- => Método del módulo fs que guarda datos en un fichero empezando desde el byte 0
- => Método del módulo fs que guarda datos en un fichero a partir del último byte

Ejercicio opcional

Diseñar un programa node.js que concatene varios ficheros en uno. El programa debe invocarse y responder de la siguiente forma:

```
$  
$ node concat <destination> <file1> <file2> ... <filen>  
-> files concatenated  
$
```

El comando concatenara un número variable de ficheros (<file1> <file2> ... <filen>) en el fichero <destination> añadiendo los contenidos en el orden de los parámetros.



Gestión de la concurrencia, bucle de eventos y nextTick

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

El bucle de eventos

Cola de Eventos

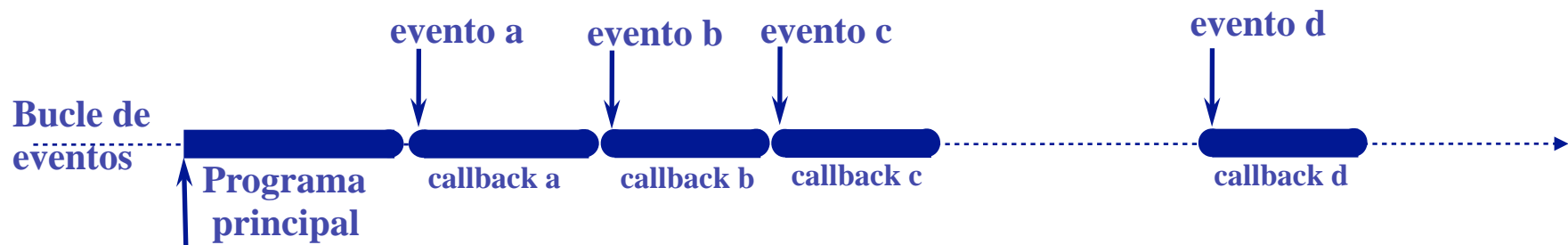


♦ node.js se ejecuta en un **único hilo (thread) de ejecución**

- 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>)

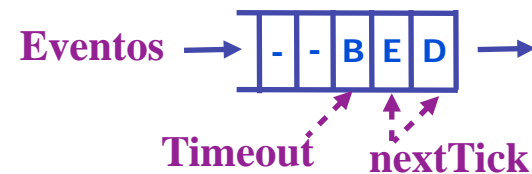
- Método de **process**
 - Introduce <callback> adelantando a los otros eventos
- **nextTick**: estrategia FIFO

◆ setTimeout() con retardo “0”

- entra en cola inmediatamente

```
.$  
.$ 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



```
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.js garantiza exclusion mutua

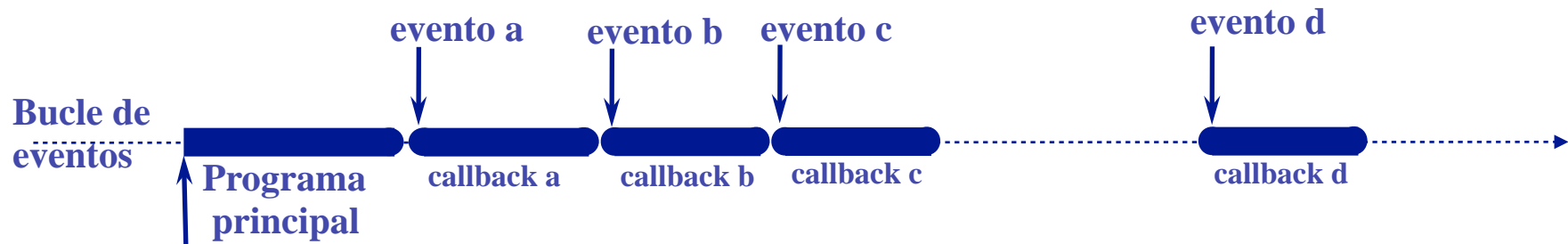
♦ node es muy sencillo de programar

- 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, ...

Cola de Eventos



Bloqueo en node.js

Cola de Eventos

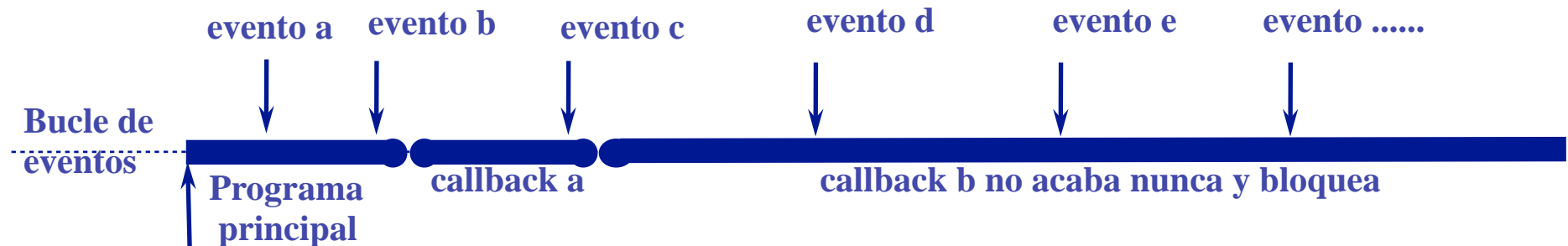


◆ Bloqueo

- **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**



Cuestionario

◆ Que respuesta describe mejor lo siguiente

bucle de eventos de node.js

cola de eventos

nextTick(f1)

exclusión mutua

bloqueo por inanición (starvation)

manejador o callback

POSIBLES RESPUESTAS:

- => Función node.js que añade eventos que adelantan a los eventos convencionales
- => Bucle con el que node.js atiende los eventos pendientes de la cola de eventos
- => Cola que almacena los eventos lanzados pero no atendidos todavía
- => Función que se ejecuta para atender a un evento o a un cambio del exterior del programa
- => Garantía de que la función de un evento no será interrumpida por la de otro hasta que finalice
- => Bloqueo de un programa, porque una función no acaba nunca y bloquea la ejecución de las demás



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 interrumpe 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 excution" 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

```
. $  
. $ 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  
  
. $
```

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

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.

Errores de ejecución

◆ El interprete JavaScript **analiza** las instrucciones del programa **al ejecutarlo**

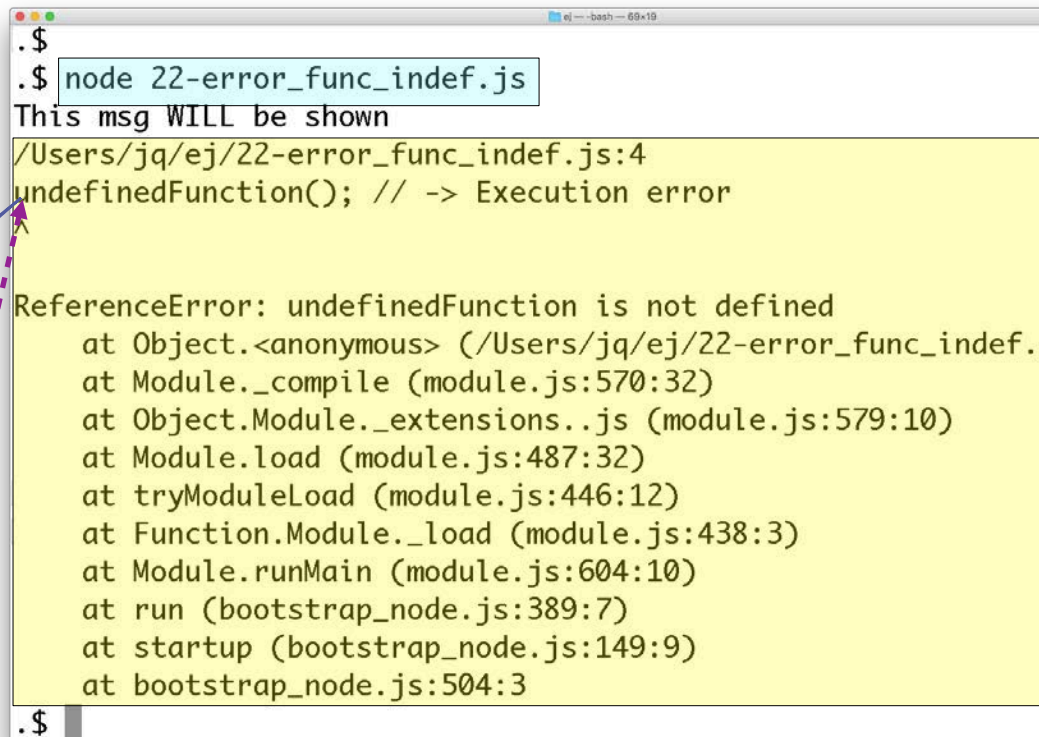
- Si encuentra algún tipo de **error** lanza una excepción con un valor de la **clase Error**
 - ◆ El ejemplo invoca una **función no definida**, que lanza un **ReferenceError** (clase derivada de Error)
 - http://www-db.deis.unibo.it/courses/TW/DOCS/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError.html
 - ◆ 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");
```



```
. $  
. $ node 22-error_func_indef.js  
This msg WILL be shown  
/Users/jq/ej/22-error_func_indef.js:4  
undefinedFunction(); // -> Execution error  
  
ReferenceError: undefinedFunction is not defined  
    at Object.<anonymous> (/Users/jq/ej/22-error_func_indef.  
    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

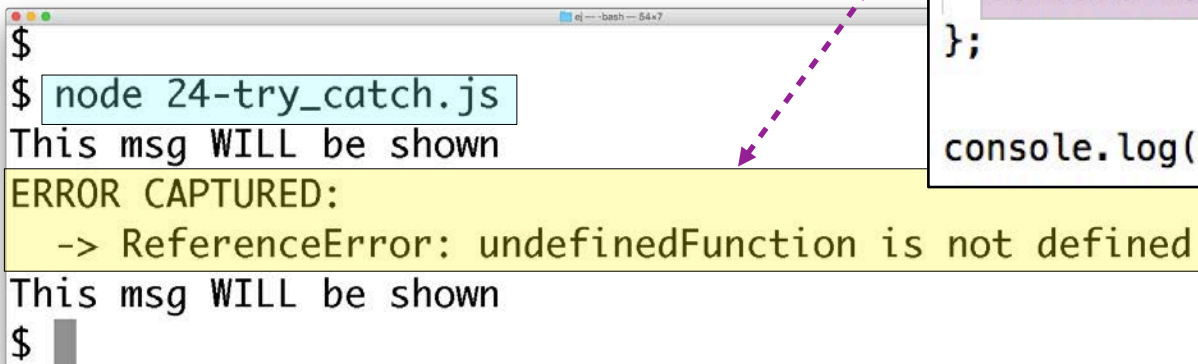
- Excepciones o errores del **programa**
 - ♦ p.e., **throw new Error(..)** lanza un error
- Errores lanzados por el interprete

```
.....
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.

```
.$  
.$ 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' 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**.

```
.$  
.$ node 25-age_error.js John  
Age of John is 30  
.$  
.$ node 25-age_error.js Jo  
'Jo' not in DB  
.$  
.$ node 25-age_error.js  
'undefined' not in DB  
.$
```

Lanza **Error**.

uncaught-Exception

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

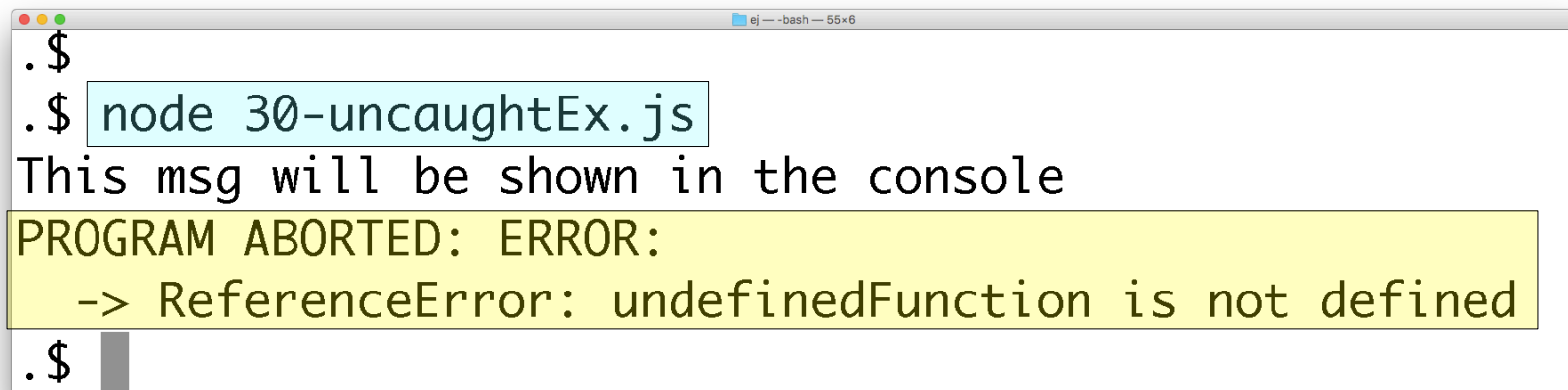
console.log('This msg will be shown in the console');

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



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


Cuestionario

◆ Que respuesta describe mejor lo siguiente

Excepción

Error

Error de ejecución

`throw "out_of_range_detected"`

`throw new Error("error_detected")`

`'uncaughtException'`

`try {...} catch (x) {...}`

`try {...} catch (x) {...} finally {...}`

POSIBLES RESPUESTAS:

- => Error lanzado por el interprete de JavaScript que interrumpe la ejecución si no es capturado
- => Señal enviada por la sentencia `throw` que interrumpe la ejecución del programa si no es capturada
- => Excepción que lleva un objeto de error como valor asociado
- => Sentencia que lanza una excepción con un valor asociado
- => Sentencia que lanza una excepción de tipo error con un objeto de error asociado
- => Sentencia que permite capturar con un `catch` las excepciones lanzadas en la parte `try`
- => Sentencia que ejecuta siempre la parte `finally`, tanto si en `try` se lanzan excepciones, como si no
- => Evento lanzado antes de interrumpir la ejecución si una excepción llega al entorno global

ES6



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
- ◆ 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>)**

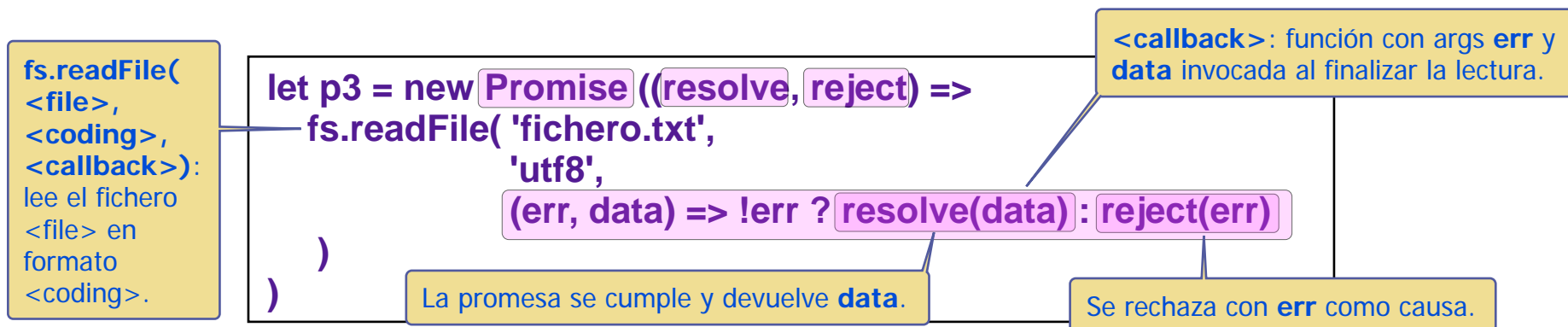
- El **<ejecutor>** es una función (**((resolve, reject) => <sentencias>)**, donde
 - ♦ **resolve** y **reject** son funciones que deben ser invocadas para cumplir o rechazar la promesa

◆ 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 resultado de esta segunda promesa cuando se cumpla o rechace
 - 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



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
- 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

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
- 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

◆ **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>)**

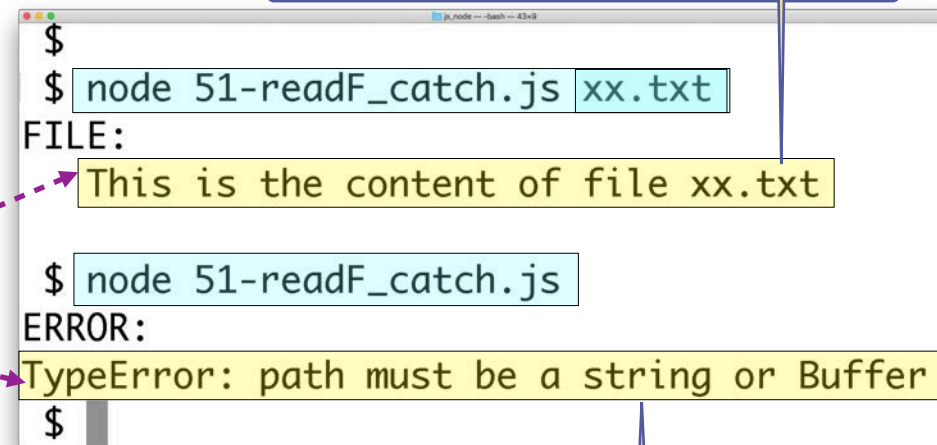
- **catch(<manejador de rechazo>)** equivale a **then(undefined, <manejador de rechazo>)**
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch

```
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-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
$
```

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

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

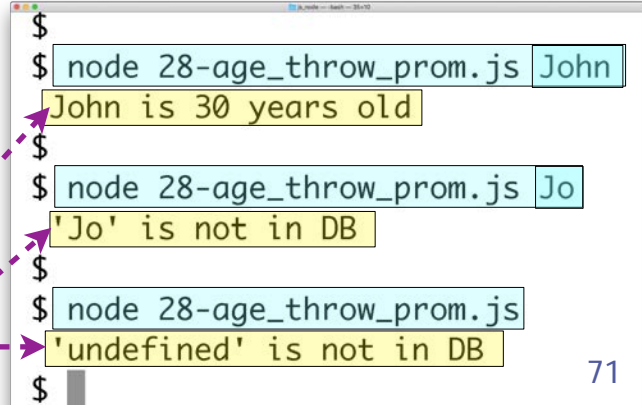
◆ La **promesa** se **rechaza** lanzando **errores o excepciones** con **throw <code>**

- **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.

La sentencia **return person** lleva a finalizar con **éxito** la promesa y ejecutar el siguiente **then** pasando el valor **person**.



```
$ 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
$
```

Cuestionario

◆ Dada la siguiente promesa

```
function my_promise (x) {  
  return new Promise( (resolve, reject) =>  
    (x > 5) ? resolve(x) : reject("err2")  
  );  
}
```

◆ Que respuesta describe mejor el estado en que finaliza esta composición de promesas

```
Promise.resolve(7)  
.then((x) => {if (x>5) {return x;} else {throw("err2");}});
```

```
Promise.resolve(4)  
.then((x) => {if (x>5) {return x;} else {throw("err2");}});
```

POSIBLES RESPUESTAS:

- => Promesa cumplida con valor 7
- => Promesa cumplida con valor "err2"
- => Promesa cumplida con valor "Error: 7"
- => Promesa cumplida con valor "Error: err2"
- => Promesa rechazada con mensaje "err2"
- => Promesa rechazada con mensaje "Error: err2"

```
Promise.resolve(7)  
.then((x) => "Error: " + x)  
.catch((x) =>  
  Promise.reject("Error: " + x));
```

```
Promise.reject("err2")  
.then((x) => "Error: " + x)  
.catch((x) =>  
  Promise.reject("Error: " + x));
```

```
my_promise (7)  
.then((x) => x);
```

```
my_promise (4)  
.then((x) => x);
```

```
my_promise (7)  
.then((x) => x)  
.catch((x) => x);
```

```
my_promise (4)  
.then((x) => x)  
.catch((x) => x);
```

```
my_promise (7)  
.then((x) => x)  
.catch((x) => {throw(x)})  
.then((x) => "Error: " + x );
```

```
my_promise (4)  
.then((x) => x)  
.catch((x) => {throw(x)})  
.then((x) => "Error: " + x );
```

Ejercicio opcional

Dado el programa de copia de ficheros que se vio en el tema correspondiente, definir tres funciones que generan promesas*: una de comprobación de parámetros (checkParams), otra de lectura (readFilePromise) y una tercera de escritura (writeFilePromise) de ficheros. Realizar con dichas funciones un programa de copia de ficheros similar al ya visto, componiéndolas con then y catch, y añadiendo lo que haga falta para que den exactamente los mismos mensajes. Debe invocarse con

```
$ node copy_2 <orig> <dest>
```

*Nota: las funciones que generan promesas deben devolver una promesa como valor de retorno. Con la sintaxis tradicional de funciones la promesa debe devolverse con la sentencia return. return se omite al utilizar la notación flecha de ES6 (sin corchetes) que retorna el valor resultante de evaluar una expresión.

Sugerencias de código:

```
function checkParams () { .... código que genera promesa .... };
```

```
function readFilePromise (file_name, format) {  
  return new Promise( (resolve, reject) =>  
    fs.readFile ( file_name, format, (err, data) => (!err) ? resolve(data) : reject(err) )  
  );  
};
```

```
function writeFilePromise (file_name, data) { .... código que genera promesa .... };
```

ES6



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

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

Promesas: async/await

- ◆ ES8 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 la **promesa** se **cumpla** si esta **retorna un valor** o se **rechace** si esta genera una **excepción**
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/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));
```

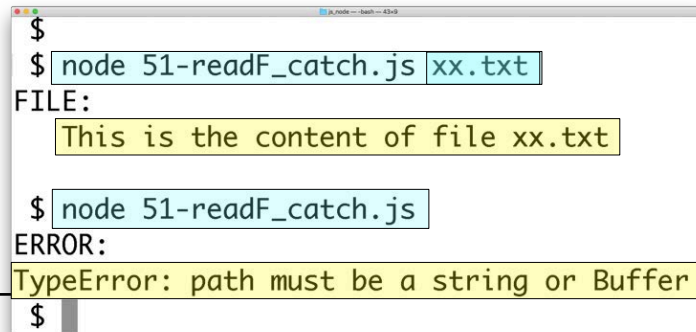
```
$  
$ node 60-msg_prom.js  
Msg:Hi Peter  
$
```

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



```
$  
$ 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  
$
```

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))
```

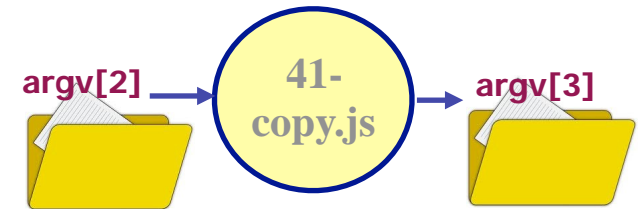
52-readF_async_catch.js

```
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 try_catch_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);  
  }  
}  
  
try_catch_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
}
  
```

El programa importa el paquete fs de node.

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

```

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 leído.

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
$
  
```

- ◆ 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

Ejemplo: Copy II

```
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));
```

```
$
$ 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)
  })

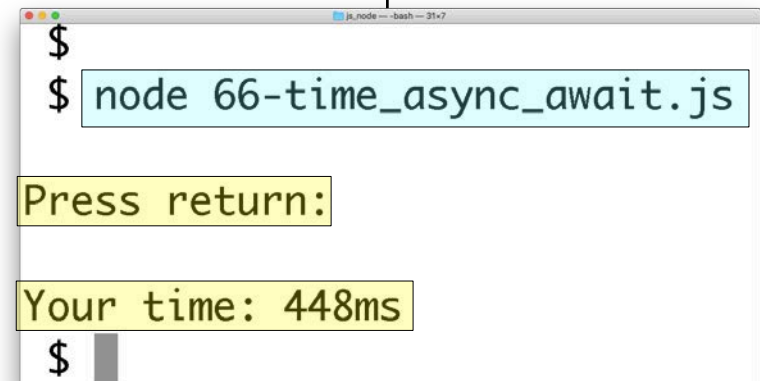
  console.log("\nPress return:");           // saca mensaje por consola
  let start = new Date().getTime();         // guarda el instante actual

  await new Promise((resolve, reject) => {   // espera a se pulse "return"
    process.stdin.setEncoding('utf8');      // configura código de entrada a UTF-8

    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
```



```
$
$ node 66-time_async_await.js
Press return:
Your time: 448ms
$
```

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

◆ **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 está
 - ◆ 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

```
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
$
```

Cuestionario

◆ Dado el programa

```
async function prom1 (x) {  
  if ( x > 9) { return x; }  
  else      { throw("Error: 4")}  
}
```

```
async function prom2 () {
```

```
  let x = await  <promesa> ; // <-- sustituir <promesa> por prom1(..)  
  if (x < 20) { return x;}  
  else      { throw("err2");};  
}
```

```
prom2();
```

◆ Que respuesta describe mejor el estado en que finaliza la invocación de prom2(), si hemos sustituido <promesa> por:

```
prom1(7);  
prom1(45);  
prom1(11);  
prom1(9);  
prom1(15);
```

POSIBLES RESPUESTAS:

- => Promesa cumplida con valor 7
- => Promesa cumplida con valor 9
- => Promesa cumplida con valor 11
- => Promesa cumplida con valor 15
- => Promesa cumplida con valor 45
- => Promesa rechazada con mensaje "err2"
- => Promesa rechazada con mensaje "Error: 4"

Ejercicio opcional

Reutilizar las funciones de comprobación de parámetros (checkParams), de lectura (readFilePromise) y escritura (writeFilePromise) del ejercicio del capítulo anterior para realizar un programa de copia de ficheros similar al anterior, pero componiendo dichas promesas con async/await. Debe invocarse con

```
$  
$ node copy_3 <orig> <dest>  
$
```



Final del tema
Muchas gracias!