



Introducción a node.js

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

node.js y sockets - Índice

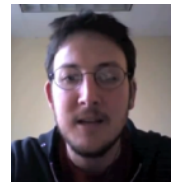
1.	<u>node.js, librería de módulos, entorno de ejecución y modo estricto</u>	<u>3</u>
2.	<u>Paquete npm, package.json, directorio del proyecto, node_modules, npx, registry</u>	<u>10</u>
3.	<u>Módulos node.js: module.exports, require, __dirname y __filename</u>	<u>17</u>
4.	<u>Timers, eventos, flujos (streams), stdin, stdout y stderr</u>	<u>24</u>
5.	<u>Acceso a Ficheros: readFile, writeFile, appendFile, readStream, writeStream,</u>	<u>33</u>
6.	<u>Acceso a Ficheros: uso de Excepciones y Promesas</u>	<u>39</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

<https://nodejs.org/en/>

◆ 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 v12.14.0 Documentation

[Index](#) | [View on single page](#) | [View as JSON](#) |
version ▼ | [Edit on GitHub](#)

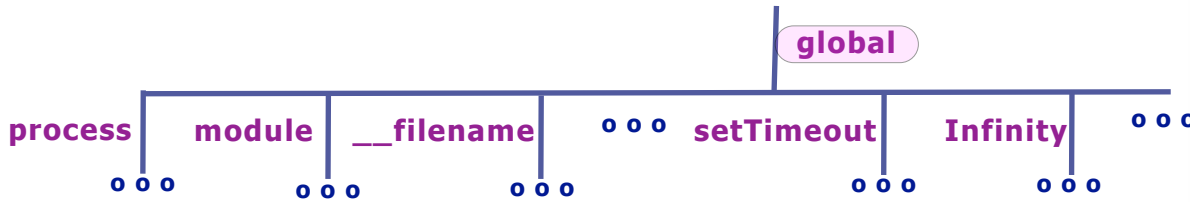
Table of Contents

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

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

- [HTTP](#)
- [HTTP/2](#)
- [HTTPS](#)
- [Inspector](#)
- [Internationalization](#)
- [Modules](#)
- [Net](#)
- [OS](#)
- [Path](#)
- [Performance Hooks](#)
- [Policies](#)
- [Process](#)
- [Punycode](#)
- [Query Strings](#)
- [Readline](#)
- [REPL](#)
- [Report](#)
- [Stream](#)
- [String Decoder](#)
- [Timers](#)
- [TLS/SSL](#)
- [Trace Events](#)
- [TTY](#)
- [UDP/Datagram](#)
- [URL](#)
- [Utilities](#)
- [V8](#)
- [VM](#)
- [Worker Threads](#)
- [Zlib](#)
- [GitHub Repo & Issue Tracker](#)

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

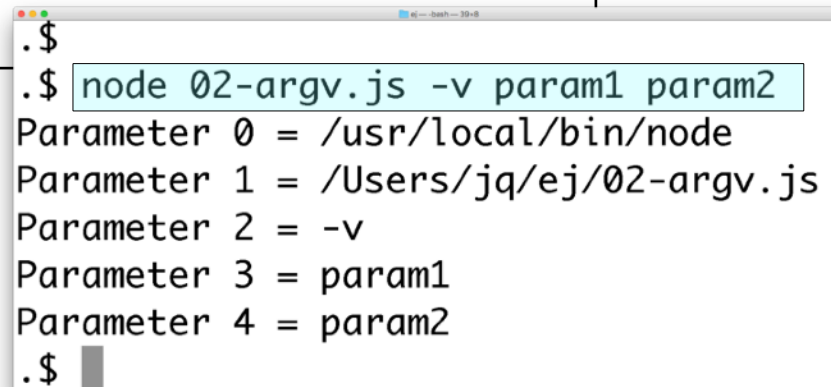
◆ Node se ejecuta en un proceso del S.O. (UNIX, ...)

- **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/api/process.html>

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

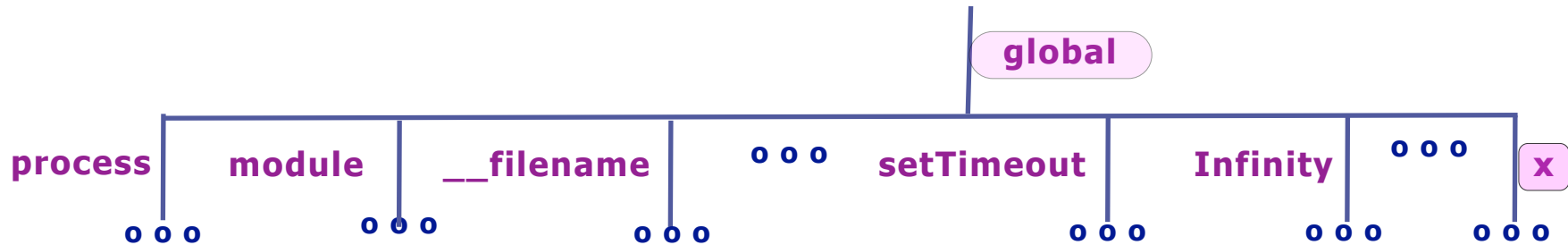
- Permite acceder a ellos desde el programa node.js

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



```
.$
.$ 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'

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
 - ◆ ...

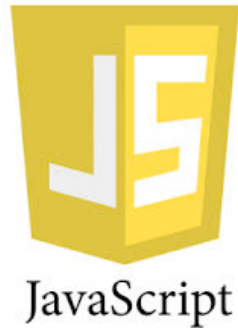
En **modo estricto** asignar una **variable no definida** provoca un error de ejecución.

```
'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



Paquete npm, package.json, directorio del proyecto, node_modules, npx, registry, ..

Juan Quemada, DIT - UPM

npm - Node Package Manager

◆ Sistema de gestión de paquetes

- Controla la instalación, actualización, configuración y eliminación de software

◆ npm es el sistema de gestión de paquetes de **node.js**

- Hoy se ha convertido en uno de los mayores ecosistemas de paquetes de software
 - Ver: <https://npmjs.com>
- **yarn**: alternativa mejorada y compatible con npm
 - Ver: <https://yarnpkg.com/>

◆ comando **npm** (o cli-npm)

- Comando de UNIX y de otros S.O. para gestión de paquetes
 - Documentación: <https://docs.npmjs.com/cli-documentation/>
 - Instalación (junto con node.js):
 - <https://docs.npmjs.com/getting-started/installing-node>
 - <http://blog.npmjs.org/post/85484771375/how-to-install-npm>

◆ comando **npx**

- Comando de UNIX y otros S.O. para ejecutar paquetes npm instalados en el ordenador (se instala con npm)
 - Documentación: <https://www.npmjs.com/package/npx>

```
venus:cal_2com jq$
venus:cal_2com jq$ npm --help

Usage: npm <command>

where <command> is one of:
  access, adduser, bin, bugs, c, cache, completi
  ddp, dedupe, deprecate, dist-tag, docs, edit,
  help, help-search, i, init, install, install-t
  list, ln, logout, ls, outdated, owner, pack, p
  prune, publish, rb, rebuild, repo, restart, ro
  run-script, s, se, search, set, shrinkwrap, st
  start, stop, t, tag, team, test, tst, un, unin
  unpublish, unstar, up, update, v, version, vie

npm <cmd> -h      quick help on <cmd>
npm -l            display full usage info
npm help <term>   search for help on <term>
npm help npm      involved overview

Specify configs in the ini-formatted file:
  /Users/jq/.npmrc
or on the command line via: npm <command> --key va
Config info can be viewed via: npm help config

npm@3.8.6 /usr/local/lib/node_modules/npm
venus:cal_2com jq$
```

Paquete y directorio de un proyecto

◆ Directorio de un proyecto

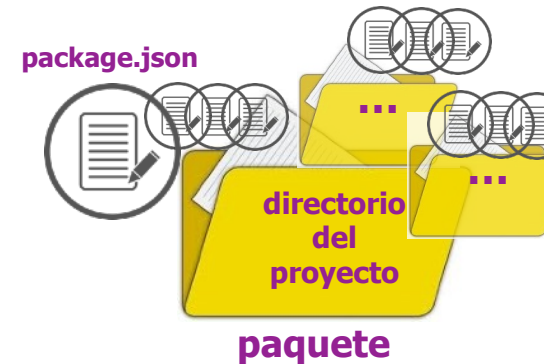
- Contiene **todo el software** del proyecto en sus **ficheros**, incluyendo sus **subdirectorios**

◆ Paquete

- Es un **directorio** de un proyecto que contiene un fichero **package.json**

◆ Algunos ficheros y directorio típicos de un proyecto

- **package.json**: fichero que empaqueta el el proyecto como un paquete npm
 - Define todos las características del paquete npm
- **node_modules**: directorio donde se instalan los paquetes que utiliza (dependencias)
 - Cada paquete puede utilizar así versiones diferentes de un mismo paquete
- **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**
- **test**: directorio con las pruebas de funcionamiento correcto
-



Registro central npm

◆ Registro central npm (registry)

- Repositorio de paquetes de la comunidad npm
 - Es uno de los mayores ecosistemas de software
- Modelo de negocio: paquetes públicos gratis, privados de pago
- Permite acceso **Web** o con comandos **npm**

◆ Publicar mi paquete en el registro central

- Cualquier persona registrada puede publicar paquetes
 - Doc: <https://docs.npmjs.com/cli-commands/publish.html>

```
$ npm publish <paquete> ## publica paquete en el registro central
```

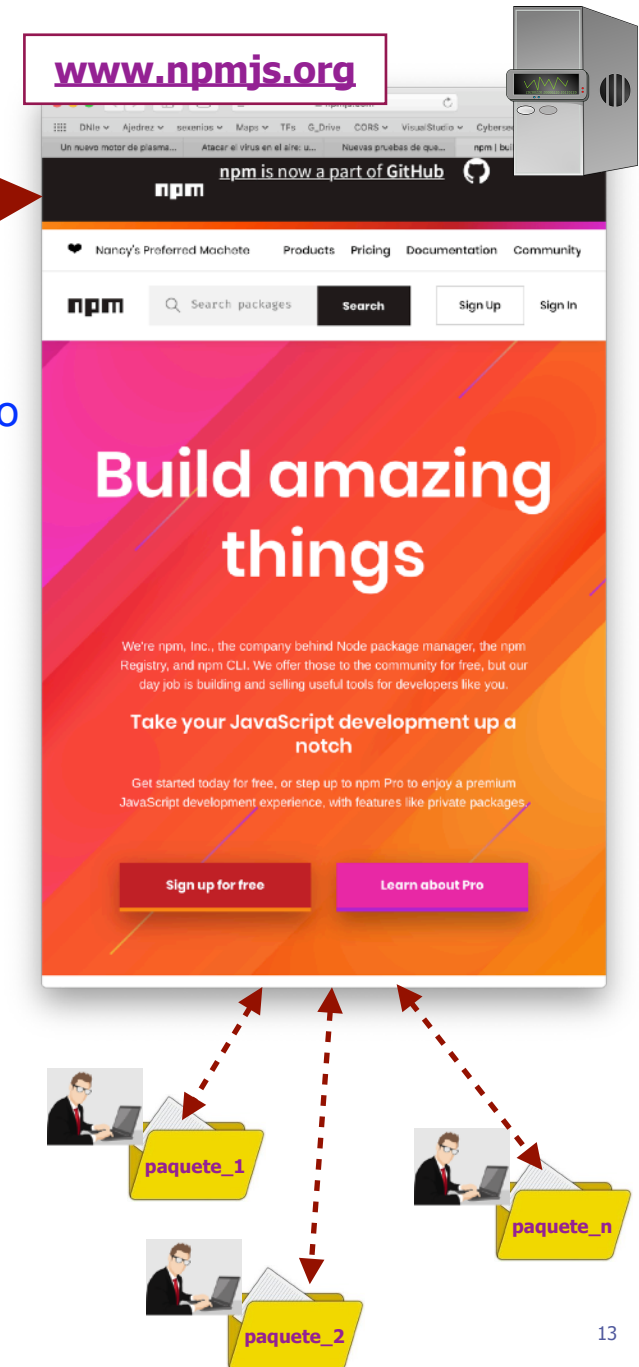
◆ Instalar paquete del registro central en mi proyecto

- Cualquier persona puede instalar paquetes publicado
 - Doc: <https://docs.npmjs.com/cli-commands/install.html>

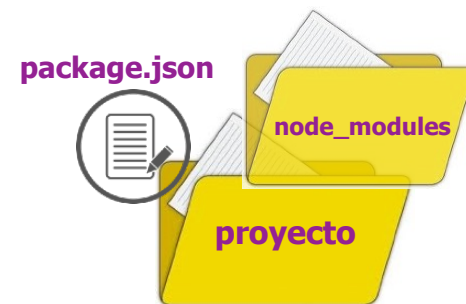
```
$ npm install <paquete> ## instala paquete del registro central en un proyecto  
$  
$ npm i <paquete>      ## es equivalente usar install o i
```

◆ Servicios de interés relacionados

- Comparar paquetes entre sí: <https://npmcompare.com>
- Comparar descargas de paquetes: <https://www.npmtrends.com>



Fichero package.json



◆ Fichero **package.json**

- Contiene la información de gestión del proyecto en formato JSON
 - Está en el directorio raíz del proyecto
- Doc: <https://docs.npmjs.com/files/package.json>

◆ Comando **init**

- Crea package.json: <https://docs.npmjs.com/cli-commands/init.html>

```
$ npm init <paquete> ## crea package.json y estructura del paquete
```

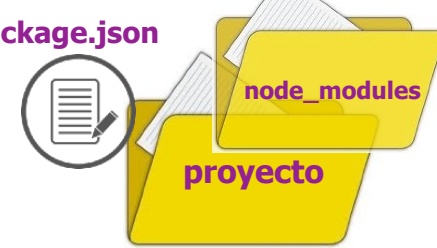
◆ **Metadatos** del programa: nombre, versión, tipo, ...

◆ **Scripts**: comandos de shell que realizan operaciones

- Se ejecutan con: **npm run <script>**
 - Algunos scripts muy habituales no necesitan run incluir run: start, test, ..
 - Doc: <https://docs.npmjs.com/cli-commands/run-script.html>

```
$ npm start      ## Ejecuta script start que arranca el paquete (no necesita run)
$
$ npm test       ## Ejecuta el script test que pasa batería de tests (no necesita run)
$
$ npm run super  ## Ejecuta el batería de tests al paquete (no necesita run)
.....
```

```
{
  "name": "quiz-2020",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www",
    "super": "supervisor ./bin/www",
    "test": "NODE_ENV=testing mocha"
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "ejs": "~2.6.1",
    "express": "~4.16.1",
    "express-partials": "^0.3.0",
    "express-session": "^1.17.0",
    "method-override": "^3.0.0",
    "morgan": "~1.9.1",
    "sequelize": "^5.21.5",
    "sequelize-cli": "^5.5.1",
    "serve-favicon": "^2.5.0",
    "sqlite3": "^4.1.1",
    "supervisor": "^0.12.0"
  },
  "devDependencies": {
    "mocha": "^7.1.0",
    "zombie": "^6.1.4"
  }
}
```



npm install: paquetes y dependencias

◆ Dependencias de un paquete: **dependencies**

- Conjunto de paquetes utilizados por el paquete
 - Un paquete no se puede ejecutar si sus dependencias no están instaladas

◆ Dependencias de desarrollo: **devDependencies**

- Dependencias adicionales del desarrollo como depuradores, tests, ..
 - Se omiten con la opción **--production** o la variable de entorno **Node_ENV**

◆ Instalar un **paquete**

- Traer sus dependencias del registro y las instala en **node_modules**
 - **node_modules** es un directorio de la raíz de mi proyecto/paquete

\$ npm i ## Se ejecuta con el terminal de comandos en la raíz del paquete

◆ Añadir una nueva **dependencia** a un paquete

- Trae el paquete del **registro central**, lo instala en **node_modules** y añade la dependencia en **package.json**
 - la opción por defecto es **--save** (guardar en **dependencies**) y no hace falta incluirla
 - la opción **--save-dev** o **-D** añade el paquete instalado a **devDependencies**
 - La opción **--no-save** no lo guarda en package.json

\$ npm i <paquete> ## Se ejecuta con el terminal de comandos en la raíz del paquete

```
{
  "name": "quiz-2020",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www",
    "super": "supervisor ./bin/www",
    "test": "NODE_ENV=testing mocha -r",
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "ejs": "~2.6.1",
    "express": "~4.16.1",
    "express-partials": "^0.3.0",
    "express-session": "^1.17.0",
    "method-override": "^3.0.0",
    "morgan": "~1.9.1",
    "sequelize": "^5.21.5",
    "sequelize-cli": "^5.5.1",
    "serve-favicon": "^2.5.0",
    "sqlite3": "^4.1.1",
    "supervisor": "^0.12.0"
  },
  "devDependencies": {
    "mocha": "^7.1.0",
    "zombie": "^6.1.4"
  }
}
```

package.json

Versiones de un paquete

◆ <paquete>@<version>.<update>.<patch>

- <version> - las versiones mayores no suelen mantener compatibilidad
 - 0 es la versión de desarrollo y a partir de 1 comienzan las oficiales
- <update> - versión menor que mantiene compatibilidad
 - normalmente no hay problema al actualizar al último update
- <patch> - parche que corrige un error
 - Se debe actualizar siempre al último parche, que suele corregir los errores detectados

◆ Symbolos ^ y ~

- ^ indica que se debe instalar el mejor **update**
- ~ indica que se debe instalar el mejor **patch**

◆ Comando npm outdated

- npm outdated - indica las dependencias que necesitan actualización
- npm update - actualiza las dependencias no actualizadas

```
$ npm i express@4.16.1  ## instala la versión 4.16.1 del paquete
$
$ npm i express@^4.16.1  ## instala el patch recomendado de la versión 4.16.1
$
$ npm outdated           ## indica que dependencias están desactualizadas
.....
$ npm update             ## actualiza las dependencias desactualizadas
```

package.json



node_modules

proyecto

```
{
  "name": "quiz-2020",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www",
    "super": "supervisor ./bin/www",
    "test": "NODE_ENV=testing mocha -r",
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "ejs": "~2.6.1",
    "express": "~4.16.1",
    "express-partial": "^0.3.0",
    "express-session": "^1.17.0",
    "method-override": "^3.0.0",
    "morgan": "~1.9.1",
    "sequelize": "^5.21.5",
    "sequelize-cli": "^5.5.1",
    "serve-favicon": "^2.5.0",
    "sqlite3": "^4.1.1",
    "supervisor": "^0.12.0"
  },
  "devDependencies": {
    "mocha": "^7.1.0",
    "zombie": "^6.1.4"
  }
}
```

package.json



Módulos node.js: `module.exports`,
`require`, `__dirname` y `__filename`

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

Módulos node.js

◆ **node.js** incorpora un sistema de **módulos** propietario

- Soluciona la carencia de modulos ES5
 - ◆ Documentación: <http://nodejs.org/api/modules.html>

◆ Un **módulo node.js** es un **fichero diferente** con dos partes

- **Interfaz**
- **Implementación**

◆ **ES6** añade en 2015 otro sistema de módulos diferente

- Se basa en las sentencias **export** e **import**
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export>
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>
- Los módulos **ES6** siempre están en modo estricto
- Los módulos **ES6** son todavía experimentales (nivel 1) en node.js

Interfaz e implementación de módulos node.js

◆ Interfaz

- Parte **pública** que permite que otros módulos y el programa principal lo usen
- El interfaz se define de dos formas (excluyentes entre si)
 - ◆ **exports.nombre = <nombre>** exporta una colección de propiedades y métodos
 - ◆ **module.exports = <objeto_interfaz>** exporta solo un objeto principal o constructor

◆ Implementación

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

◆ El entorno de ejecución (objeto global) de JavaScript

- está accesible en todos los módulos

◆ 'use strict' al comienzo del módulo activa el modo estricto (seguro)

Importar módulos: método require

◆ El método **require(<module>)** importa todo lo exportado por **<module>**, donde

- **<module>** puede ser el nombre de algún paquete 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 otros (la casuística es compleja)
 - ◆ Ver: <https://nodejs.org/api/modules.html>

◆ 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

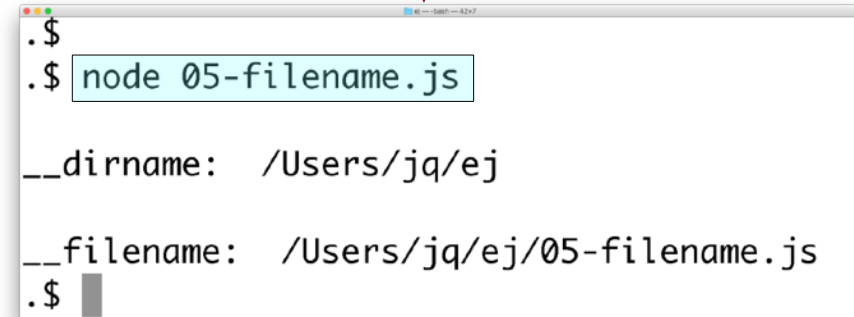
◆ Todos los módulos tienen definidos 5 propiedades locales

- **__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

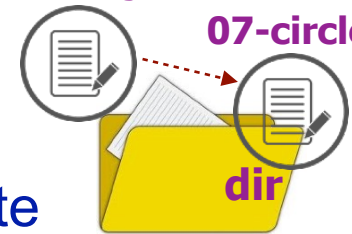
- **module.exports** o a **exports.xx**

◆ Doc: <https://nodejs.org/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
```

```
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
```

07-circle.js

```

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.

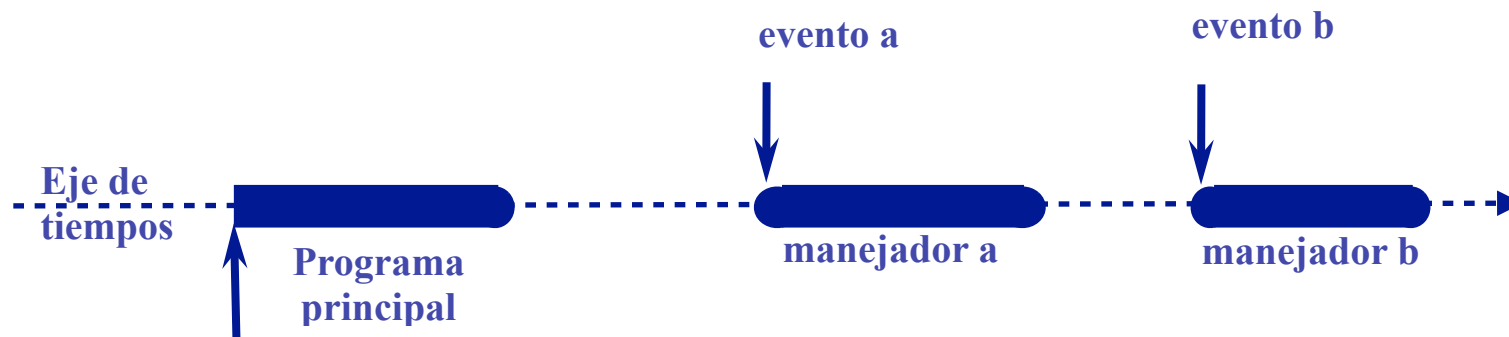


Timers, eventos, flujos (streams), stdin, stdout y stderr

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



Eventos: Clase EventEmitter

◆ Todas las clases que emiten eventos derivan de **EventEmitter**

- Heredan métodos: **addListener(...)**, **removeListener(...)**, **on(...)**, **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 **addListener(<event>, <listener>)** es equivalente a **on(<event>, <listener>)**
 - **removeListener(<event>, <listener>)** desinstala el manejador <listener>

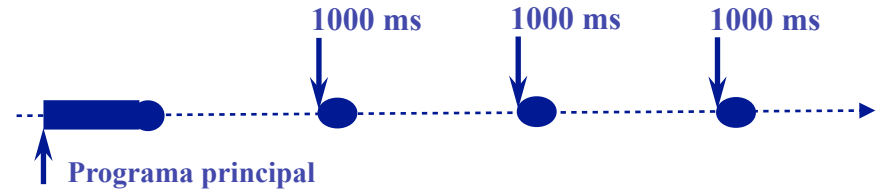
◆ El método **obj.emit(<evento>, <p1>, <p2>, ...)**

- 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 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.

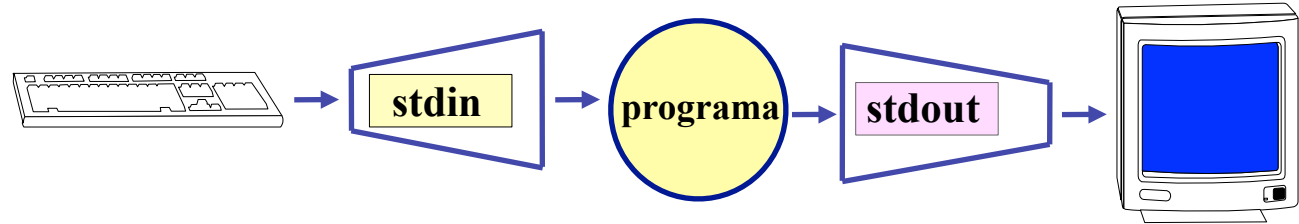
setInterval(..) programa un **evento interno periódico** que ejecuta la función y envía el **evento 'event'** cada 1000 ms.

```
$  
$ node 15-timer_obj.js  
an event occurred!  
an event occurred!  
an event occurred!  
an event occurred!  
an event occurred!  
^C  
$
```

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

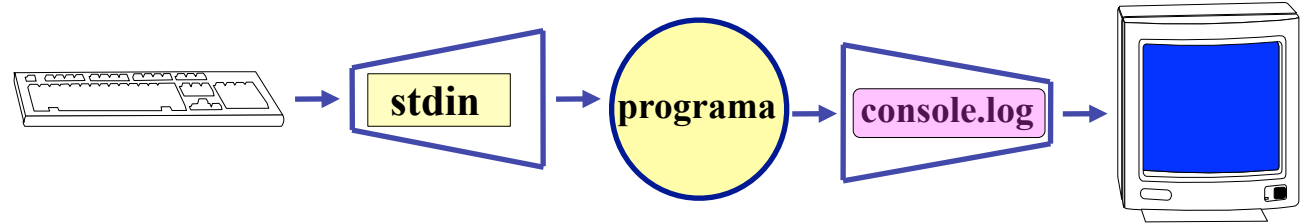
- **stdin**: entrada estándar (teclado), que recibe líneas tecleadas con el evento **data**
 - stdin se para con `pause()` y arranca con “`resume()`”
 - El flujo de entrada se cierra desde el teclado con `^D` o `^C` y desde programa con `close()`
- **stdout**: salida estándar asignada a pantalla
- **stderr**: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

```
$
$ 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 terminal shows the execution of a Node.js script named '17-stdin.js'. The user enters 'node 17-stdin.js' at the prompt. The script outputs 'This year' twice, followed by a blank line, then 'is very dry!' twice, followed by another blank line. The prompt '.\$' is visible at the bottom of the terminal.

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

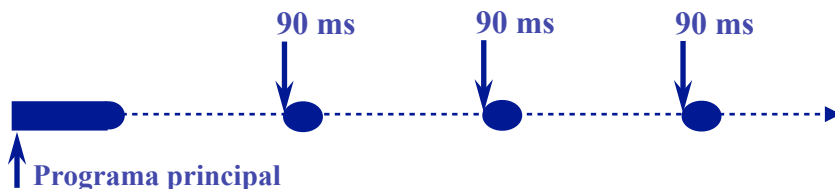
```
function time(){
  let t = new Date();
  return "Time: " + t.getHours() + "h " + t.getMinutes() + "m "
    + t.getSeconds() + "s " + t.getMilliseconds();
}
```

`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.

```
$
$ node 18-readline.js
Type something: Hi
Did you type 'Hi'
Type something: Hello
Did you type 'Hello'
Type something: Have a great day!
Have a great day!
$
```

rl.prompt()



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



◆ La anidación de callbacks garantiza el orden de ejecución

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

El programa importa el paquete fs de node.

```
if (process.argv.length !== 4){  
  console.log('  syntax: "node 40-copy <orig> <dest>"');  
  process.exit()  
}
```

Se comprueba si se han incluido los nombres de los ficheros origen y destino, y se da mensaje de error si no se han incluido.

Se copian con multiasignación (ES6) los parámetro 3 y 4 a las variables **orig** y **dest**.

```
const [, , orig, dest] = process.argv;
```

fs.readFile(<orig>, <callback>) da la orden de lectura del fichero **<file>** e instala el manejador **<callback>** para que procese el fichero al finalizar la lectura.

```
fs.readFile(  
  orig,
```

El primer **<callback>** se invoca al finalizar la lectura. Si no hay errores, la lectura habrá sido exitosa y el contenido estará en **data**.

```
  function(err, data) {  
    if (err) throw err;
```

Si no hay error de lectura, se invoca **fs.writeFile(<dest>, <data>, <callback>)** que ordena la escritura de los datos leídos (**<data>**) en el fichero **<dest>** e instala **<callback>** que se ejecuta al finalizar y muestra el mensaje si no hay error.

```
    fs.writeFile(  
      dest,
```

El segundo **<callback>** envía mensaje final a consola, si la escritura de **<data>** en **dest** se realiza sin error.

```
      data,  
      function (err) {  
        if (err) throw err;  
        console.log('  file copied');      }  
    );  
  }  
);
```

40-copy.js

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

Ejemplo: Append

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

if (process.argv.length !== 4){
  console.log('  syntax: "node 41-append <orig> <dest>"');
  process.exit()
}

const [, , orig, dest] = process.argv;

fs.readFile(
  orig,
  function(err, data) {
    if (err) throw err;
    fs.appendFile(
      dest,
      data,
      function (err) {
        if (err) throw err;
        console.log('  file appended');
      }
    );
  }
);
```



El mensaje de error de invocación se modifica.

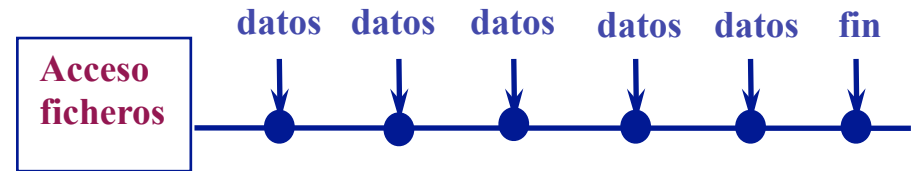
Programa similar a copy que utiliza el método **appendFile (...)** de node en vez de **writeFile (...)**

El mensaje final también se modifica.

A terminal window titled 'CORE_08_MOD_8_ej -- -bash -- 42x7' showing the following commands and output:

```
$
$ node 41-append
syntax: "node 41-append <orig> <dest>"
$
$ node 41-append xx.txt yy.txt
file appended
$
```

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)

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

if (process.argv.length !== 4){
  console.log('    syntax: "node 42-copy <orig> <dest>"');
  process.exit()
}
```

```
const [, , orig, dest] = process.argv;
```

```
const readStream = fs.createReadStream(orig);
const writeStream = fs.createWriteStream(dest);
```

```
readStream.pipe(writeStream);
```

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

```
CORE_08_MOD_8.ej — -bash — 43x7
$
$ node 42-copy
  syntax: "node 42-copy <orig> <dest>"
$
$ node 42-copy xx.txt yy.txt
  file copied
$
```

Se define el fichero origen como un stream de salida y el destino como un stream de entrada de datos.

Se conectan con un pipe y se realiza la copia.

42-copy.js



Acceso a Ficheros: uso de Excepciones y Promesas

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

Excepciones, errores y sentencia try-catch-finally

◆ Excepción

- Señal que interrumpe un programa al lanzarla con **throw <valor>**
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw>
- Los errores son excepciones donde **<valor>** pertenece a la clase **Error**

◆ La sentencia **try-catch-finally** permite **capturar** excepciones o errores

- Captura solo las **excepciones** que ocurren dentro del **bloque try**
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch>

◆ Si las **excepciones** ocurren dentro de **try**

- Interrumpen la ejecución de try y continúan en **catch**
 - ◆ La variable **exception** recibe **<valor>** (de throw **<valor>**)

◆ El bloque **finally** se ejecuta siempre al final

- Tanto si ocurren excepciones, como si no

```
.....  
try {  
    .....  
    throw "<valor>"  
    // or throw new Error(<valor>)  
    .....  
} catch (exception) {  
    .....  
} finally {  
    .....  
}  
.....
```


Promesas ES8

◆ Promesa

- Se crean con **new Promise((resolve, reject) =>)**
 - ♦ La promesa ejecuta el **callback** (parámetro) al crearla
 - **resolve(<data>)**: función que resuelve la promesa con valor de éxito <data>
 - **reject(<err>)**: función que rechaza la promesa con valor de rechazo <err>
- Info: <https://javascript.info/async>

◆ async define una función que retorna una promesa

- Por ejemplo, **async function suma (x, y) {...}** o **async (x, y) => {...}**
 - ♦ Al invocar **return <data>** la promesa finaliza con éxito
 - ♦ Al invocar **throw <err>** la promesa se rechaza

◆ await <promesa>: espera resolución y devuelve valor de éxito

- **let x = await promesa()** asigna a **x** el valor de éxito (solo si lo hay)
 - ♦ **await** espera resolución de la promesa y devuelve éxito o lanza excepción de rechazo
- **await promesa()** solo puede usarse dentro de una función async

◆ El rechazo de una promesa se captura con try-catch

- La promesa se **invoca** en el **bloque try** y el **rechazo** se captura en el **bloque catch**

Las dos formas de definir la promesa **prom** son equivalentes y se comportan igual, el 50% de las veces **prom** se resuelve con éxito y **main** sacará por consola el msj "Exito". El otro 50% la promesa se rechaza y main sacará por consola el msj "Error". Las promesas definidas con **async** no se pueden resolver en un callback, para poder hacerlo hay que construir la promesa con el constructor de la clase.

```
const prom = () =>
  new Promise((resolve, reject) => {
    if (Math.random() < 0.5) {
      resolve("Exito");
    } else {
      reject("Error");
    }
  })
```

```
const prom = async () => {
  if (Math.random() < 0.5) {
    return "Exito";
  } else {
    throw "Error";
  }
}

const main = async () => {
  try {
    let msg = await prom();
    console.log(msg);
  } catch (err) {
    console.log(err);
  }
}

main();
```

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

```
function readFileP (file) {  
  return new Promise(  
    (resolve, reject) => fs.readFile(  
      file,  
      (err, data) => err ? reject(err) : resolve(data)  
    )  
  );  
}
```

```
function writeFileP (file, data) {  
  return new Promise(  
    (resolve, reject) => fs.writeFile(  
      file,  
      data,  
      (err) => err ? reject(err) : resolve()  
    )  
  );  
}
```

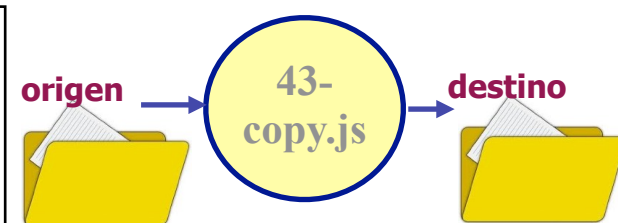
```
if (process.argv.length !== 4){  
  console.log(' syntax: "node 43-copy <orig> <dest>"');  
  process.exit();  
}
```

```
const [, , orig, dest] = process.argv;
```

```
async function copy() {  
  try {  
    let data = await readFileP(orig);  
    await writeFileP(dest, data);  
    console.log(' file copied');  
  } catch (err) { console.log(err); }  
}  
copy();
```

43-copy.js

Ejemplo Copy con promesas ES8



readFileP(<file>) y **writeFileP(<file>, <data>)** son funciones que transforman **readFile(..)** y **writeFile(..)** en promesas equivalentes. Estas promesas se resuelven dentro del callback, por lo que se deben crear con el constructor.

Si node.js se hubiese creado ahora, las funciones de sus módulos utilizarían promesas, tal y como hacemos en este ejemplo.

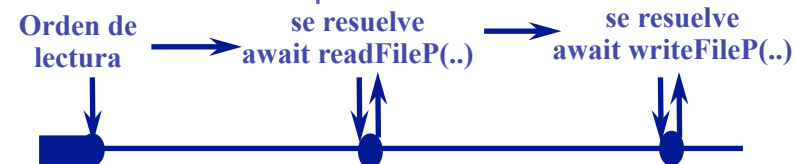
```
CORE_08_MOD_8_ej -- -bash -- 43x7  
$  
$ node 43-copy  
syntax: "node 43-copy <orig> <dest>"  
$  
$ node 43-copy xx.txt yy.txt  
file copied  
$
```

La expresión **await readFileP(orig)** invoca la promesa **readFileP(orig)** inmediatamente, pero la asignación a la **variable data** del valor de retorno no se realiza hasta que la promesa se resuelve.

await writeFileP(dest, data) invoca la promesa inmediatamente, pero la ejecución no continúa hasta que la promesa se resuelve.

Esta instrucción no se ejecutará hasta, que se resuelva la promesa, es decir hasta que la escritura finalice.

Las promesas garantizan el mismo orden de ejecución que los callbacks pero su código es mucho más sencillo de entender y programar. La resolución de una expresión **await** es equivalente a la invocación de un callback.



Ejemplo Copy con promesas ES6



El programa es similar al anterior salvo que usa **then** y **catch** de ES6. **readFileP(<file>)**, **writeFileP(<file>, <data>)** y todo el código no resaltado es igual. **async/await** no llega hasta ES8 2 años después.

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

```
function readFileP (file) {  
  return new Promise(  
    (resolve, reject) => fs.readFile(  
      file,  
      (err, data) => err ? reject(err) : resolve(data)  
    )  
  );  
}
```

```
function writeFileP (file, data) {  
  return new Promise(  
    (resolve, reject) => fs.writeFile(  
      file,  
      data,  
      (err) => err ? reject(err) : resolve()  
    )  
  );  
}
```

```
if (process.argv.length !== 4){  
  console.log('    syntax: "node 44-copy <orig> <dest>"');  
  process.exit();  
}
```

```
const [, , orig, dest] = process.argv;
```

```
readFileP(orig)  
  .then( (data) => writeFileP(dest, data))  
  .then( () => console.log('    file copied'))  
  .catch( (err) => console.log(err));
```

44-copy.js

◆ then(..) y catch(..)

- Métodos de instancia de la clase **Promise**, que devuelven **promesas** y pueden encadenarse

◆ p.then(<callback>)

- **<callback>** que atiende el **éxito** de la promesa **p**
 - ♦ Propaga el **rechazo** al siguiente then(..) o catch(..)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

◆ p.catch(<callback>)

- **<callback>** que atiende el **rechazo** de la promesa **p**
 - ♦ Propaga el **éxito** al siguiente then(..) o catch(..)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch

◆ <callback> de then(..) o catch(..)

- **throw <msg>** rechaza la promesa con mensaje de error **<msg>**
- **return <valor>** resuelve la promesa con valor de éxito **<valor>**
 - ♦ Si **<valor>** es una **promesa**, se espera a su resolución y se devuelve su **éxito** o **rechazo**

La promesa **readFileP(orig)** lee **orig** y pasa su contenido como valor de éxito.

El **primer then** instala el callback de éxito, que escribe los datos recibidos en **dest**, al ejecutar **writeFileP(dest, data)**.

El segundo **then** instala el callback de éxito que muestra el mensaje **'file copied'** por consola.

catch instala el callback de rechazo que atenderá cualquier rechazo en las promesas anteriores.

```
$  
$ node 44-copy  
syntax: "node 44-copy <orig> <dest>"  
$ node 44-copy xx.txt yy.txt  
file copied  
$
```





Final del tema