

T2 – JavaScript y NodeJS

Tecnologías de los Sistemas de Información en la Red



Objetivos

- ▶ Utilizar NodeJS (JavaScript) como herramienta base para el desarrollo de componentes en una aplicación distribuida.
- ▶ Identificar las características principales de JavaScript/NodeJS y sus ventajas en el desarrollo de aplicaciones distribuidas: orientación a eventos, posibilidad de interacción asincrónica...
 - ▶ Recordar servidores asincrónicos
- ▶ Describir algunas bibliotecas de NodeJS a utilizar durante el desarrollo de la asignatura.



1. Introducción

- ▶ El resto del texto introduce
 - ▶ El lenguaje JavaScript, concretamente ECMAScript 6
 - ▶ El entorno NodeJS
- ▶ Este texto no es una referencia sobre JavaScript ni NodeJS: se limita a los aspectos relevantes para TSR.
 - ▶ Algunos conceptos son importantes a nivel general, pero de uso limitado en la asignatura: no los detallaremos.
 - Orientación a objetos.
 - ▶ Los aspectos fuera de nuestro ámbito no llegan a abordarse



1.1 Introducción. JavaScript

- ▶ Es un lenguaje de scripting, interpretado, dinámico y portable
 - ▶ Alto nivel de abstracción
 - Programas más sencillos
 - Desarrollo más rápido
- ▶ Inicialmente diseñado para dotar de comportamiento dinámico a las páginas web.
 - ▶ Los navegadores web incluyen un intérprete de JavaScript
- ▶ Orientado a eventos e interacciones asincrónicas (*callbacks*)
 - ▶ Mejora el rendimiento y la escalabilidad de las aplicaciones
- ▶ No soporta distintos hilos de ejecución.
 - ▶ No hay objetos compartidos. No existen secciones críticas ni se necesitan herramientas de sincronización.
 - ▶ Pero debe vigilarse cuándo una variable obtendrá el valor esperado.
 - Lógica de los "callbacks".
- ▶ Soporta programación funcional y Orientada a Objetos



1.1 Introducción. NodeJS

- ▶ NodeJS:
 - ▶ Entorno orientado a la creación de aplicaciones para Internet en el contexto del servidor
 - ▶ Facilita el desarrollo de aplicaciones distribuidas
 - ▶ Construido sobre el entorno de ejecución V8 (JavaScript para Chrome)
 - ▶ Define
 - ▶ Interfaces y entorno
 - ▶ Utilidades comunes
 - ▶ Intérprete
 - ▶ Gestión de paquetes
 - ▶ ...
 - ▶ La mayor parte de las tecnologías que se consideraron al establecer las competencias y resultados de aprendizaje de TSR encajan perfectamente en NodeJS



2. JavaScript. Guion para un curso completo

- ▶ Características principales
- ▶ Alternativas para la ejecución de código
- ▶ Sintaxis
- ▶ Valores
 - ▶ Primitivos
 - ▶ Compuestos (objetos)
- ▶ Variables
 - ▶ Tipo dinámico
 - ▶ Propiedades y métodos sobre valores
 - ▶ Ámbito declaración
- ▶ Operadores
- ▶ Sentencias
- ▶ Funciones
 - ▶ Aridad
- ▶ Arrays
- ▶ Programación funcional
- ▶ Orientación a Objetos
- ▶ JSON
- ▶ Eventos
- ▶ Callbacks
 - ▶ Limitaciones asincronía con callbacks
 - ▶ Asincronía mediante promesas

La guía describe los apartados no incluidos en esta presentación



2. JavaScript. Guion para TSR

1. Características principales
2. Alternativas para la ejecución de código
3. Variables
 1. Tipo dinámico
 2. Ámbito declaración
4. Funciones
 1. Aridad
 2. Funciones y ámbito variables
 3. Clausuras
5. Eventos
6. Callbacks
 1. Limitaciones asincronía con callbacks
 2. Asincronía mediante promesas



2.1 JavaScript (JS). Características principales

- ▶ Imperativo y estructurado
 - ▶ Sintaxis similar a Java
- ▶ Multiparadigma
 - ▶ Programación funcional:
 - Las funciones son "objetos":
 - pueden pasarse como argumentos a otras funciones
 - pueden ser devueltas como resultado de una función
 - ▶ Programación orientada a objetos
 - No convencional: basada en prototipos en vez de clases y herencia
- ▶ Influencias sobre JS
 - ▶ Java sintaxis, valores primitivos vs. objetos
 - ▶ Scheme programación funcional
 - ▶ Self herencia basada en prototipos
 - ▶ Perl y Python string, array y expresiones regulares



2.2 JS. Alternativas para la ejecución de código

1. Intérprete integrado en los navegadores web
 - ▶ Empleando elementos "script" en el código HTML de una página web
 - ▶ O abrir la consola JavaScript y ejecutar de forma interactiva

2. Intérprete externo invocado desde la línea de órdenes
 - ▶ Por ejemplo NodeJS (**node programa.js**)
 - ▶ Tomaremos esta alternativa en la asignatura
 - ▶ Software y documentación disponibles en <http://nodejs.org>

2.3.1 JS. Variables (tipo dinámico)

- ▶ JavaScript no es un lenguaje fuertemente "tipado"
 - ▶ Las variables se declaran antes de su uso, pero sin tipo asociado (salvo si se inicializa en la declaración).
 - ▶ Durante su existencia una variable puede mantener valores de varios tipos (tipo dinámico)

```
let x = "texto"           // x representa una cadena (string)
x = {color:'rojo', marca:'seat', año:2018} // x ahora es un objeto
x = [1, 2, 3, "prueba", 6] // ahora un vector (array)
x = function() {return "Ejemplo"}         // ahora una función
let y = x()                               // ¿qué tipo de valor mantiene y?
```

- ▶ Gestión de tipos débil. Cuidado con las conversiones implícitas de tipos.

```
let x = '7'               // x vale "7"
x == 7                    // true (por conversión implícita de tipos)
x === 7                   // false (comparación estricta)
x + 23                    // genera "723" (+ como concatenación de string)
x + "2"                   // genera "72"
x * 2                     // genera 14 (previa conversión del valor de x a número)
```



2.3.2 JS. Ámbito declaración variables

- ▶ **Ámbito léxico**
 - ▶ El ámbito de una variable es...
 - Local al bloque en que se declara (con **let**)
 - Local a la función donde se declara (con **var**)
 - Global (accesible desde todo el fichero)
 - Si no se declara dentro de una función
 - ▶ Equivale a considerar una función implícita que abarca todo el fichero
 - O se declara en una función pero sin incluir **let** ni **var**. Ej `x=3`. NO recomendado
 - ▶ Desde un punto...
 - Se tiene acceso a (son visibles) las variables definidas en los ámbitos que incluyen ese punto
 - Busca de más interno a más externo (evita colisión de nombres)



2.4 JS. Funciones

- ▶ Literal (función anónima)

- ▶ `function (args) {...}`

```
const doble = function (x) {return 2*x}  
let x = doble(28)
```

- ▶ Es un valor que puede asignarse, pasarse como argumento, etc.

- ▶ Notación alternativa: `(args) => {...}`

```
let triple = (x) => {return 3*x}
```

- ▶ Declaración

- ▶ `function nombre(args) {...}`

```
function doble(x) {return 2*x}
```

- ▶ Equivale a: `let nombre = function (args) {...}`

- ▶ Declarables en cualquier punto, incluso dentro de otra función (anidables)

- ▶ Constituyen el ámbito de definición de las variables

- ▶ Cuando estas se definen utilizando `var`

- ▶ Paso de parámetros por valor

- ▶ Pero si trabajamos con objetos realmente pasamos referencias

- ▶ Las funciones son objetos

- ▶ Tienen propiedades y métodos, y podemos añadir otros

- ▶ Un único valor de retorno, aunque puede ser compuesto



2.4.1 JS. Funciones (aridad)

- ▶ Aridad (nº de argumentos). Una función con ***n*** argumentos se puede invocar con ***m*** valores:
 - ▶ ***m* = *n***, los valores se asocian a los argumentos por posición
 - ▶ ***m* < *n***, argumentos restantes a la derecha se inicializan a **undefined**
 - ▶ ***m* > *n***, valores sobrantes a la derecha se ignoran
- ▶ Argumentos accesibles:
 - ▶ por nombre
 - ▶ con el pseudo-array **arguments**

```
function saluda() {  
    for (let i=0; i<arguments.length; i++) {  
        console.log("Hola, " + arguments[i])  
    }  
}  
saluda("Ana", "Juan", "Isabel")
```

```
Hola, Ana  
Hola, Juan  
Hola Isabel
```



2.4.1 JS. Funciones (aridad)

- ▶ Aridad. Otras posibilidades:

- ▶ Reforzar aridad:

```
function f(x,y) {if (arguments.length !== 2) ... }
```

- ▶ Asignar valores por defecto:

```
function f(x = defaultX, y = defaultY) { ... }
```

```
function saluda(x = "Ana", y = "Juan") {  
  console.log("Hola, " + x); console.log("Hola, " + y)  
}  
saluda("Isabel"); console.log("\n-----")  
saluda(undefined, "Jordi", "Pepe")
```

```
Hola, Isabel  
Hola, Juan  
-----  
Hola, Ana  
Hola, Jordi
```

- ▶ No se permiten funciones con el mismo nombre, aunque tengan distinta aridad



2.4.2 JS. Ámbito variables (¡los comentarios son *spoilers*!)

```
function alert(x) { console.log(x) }
let global = 'this is global'

function scopeFunction() {
  alsoGlobal = 'This is also global!'
  let notGlobal = 'This is private to scopeFunction!'

  function subFunction() {
    alert(notGlobal) // We can still access notGlobal in this child function.
    stillGlobal = 'No let keyword so this is global!'
    let isPrivate = 'This is private to subFunction!'
  }

  alert(stillGlobal) // This is an error since we haven't executed subfunction
  subFunction()      // Execute subfunction
  alert(stillGlobal) // This will output 'No let keyword so this is global!'
  alert(isPrivate)   // It generates an error since isPrivate is private to
subfunction()
  alert(global)      // It outputs: 'this is global'
}

alert(global)        // It outputs: 'this is global'
alert(alsoGlobal)    // It generates an error since we haven't run
scopeFunction yet.
scopeFunction()
alert(alsoGlobal)    // It outputs: 'This is also global!';
alert(notGlobal)     // This generates an error.
```

Ejemplo tomado de:

https://www.evl.uic.edu/luc/bvis546/Essential_Javascript_-_A_Javascript_Tutorial.pdf



2.4.3 JS. Clausuras

- ▶ Clausura = función + conexión a las variables en ámbitos que la incluyen
 - ▶ Una función recuerda el entorno en que se ha creado
- ▶ Para más detalles, consultar [1]

```
function creaFunc() {  
  let nombre = "Mozilla"  
  return () => {console.log(nombre)}  
}  
let miFunc = creaFunc()  
miFunc()
```

Mozilla

```
function multiplicarPor(x) {  
  return (y) => {return x*y}  
}  
let triplicar = multiplicarPor(3)  
y = triplicar(21)  
console.log("y = " + y)
```

y = 63



2.4.3 JS. Clausuras

```
function writing(x) {  
  console.log("---\nWriting after " + x + " seconds")  
}  
  
function writingClosure(x) {  
  return function() {  
    console.log("---\nWriting after " + x + " seconds")  
  }  
}  
  
setTimeout(function() {writing(6) }, 6000)  
setTimeout(writing, 3000)  
setTimeout(writingClosure(4) , 4000)  
console.log("root(2) =", Math.sqrt(2))
```

```
root(2) = 1.4142135623730951  
---  
Writing after undefined  
seconds  
---  
Writing after 4 seconds  
---  
Writing after 6 seconds
```



2.5 JS. Eventos

- ▶ JS asume un único hilo de ejecución
 - ▶ Pero podemos ejecutar múltiples actividades
 - ▶ Estableciéndolas como eventos
- ▶ Existe una cola de eventos que
 - ▶ Recoge interacciones externas
 - ▶ Representa actividades pendientes de gestionar
 - ▶ Organiza turnos
- ▶ Podemos asociar una respuesta a cada tipo de evento
 - ▶ Pero las respuestas a eventos se ejecutan en el único hilo
 - ▶ Por lo tanto en secuencia (no se procesa un nuevo evento hasta finalizar el actual)



2.5 JS. Eventos

- ▶ Gestión de la ejecución en JavaScript
 - ▶ Pila de llamadas
 - ▶ Cada llamada a una función añade un contexto de ejecución (argumentos, variables locales) a la pila.
 - ▶ Cuando finaliza la función, se desapila el contexto.
 - ▶ Si la pila se vacía, se explora la cola de eventos.
 - ▶ Cola de eventos
 - ▶ Encola un contexto de ejecución por cada evento recibido.
 - ▶ Se atienden en orden, cuando la pila de llamadas se haya vaciado.



2.5 JS. Eventos

```
function fibo(n) {  
    return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
    console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
    console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
    showMessage("M3", j)  
}, 1)
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

Pila de
llamadas

Cola de eventos

```
C:\> node turnQueue.js
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

Pila de
llamadas

console
.log

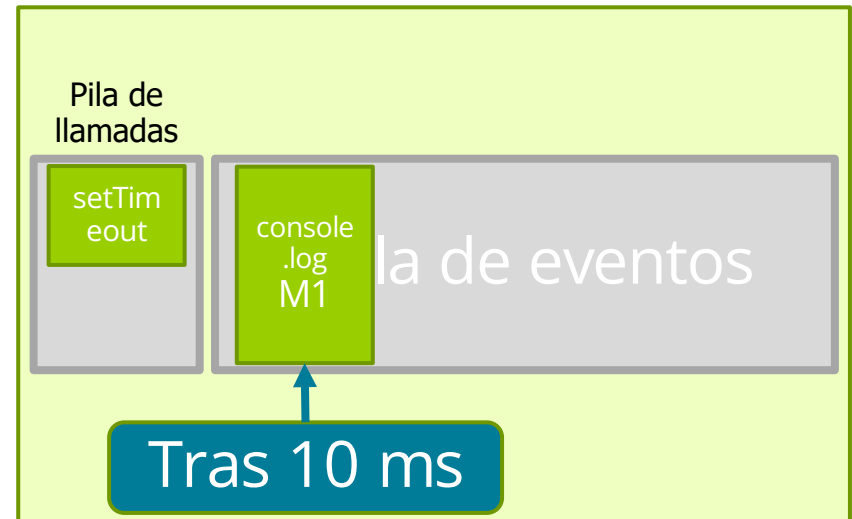
Cola de eventos

```
C:\> node turnQueue.js  
Starting the process...
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

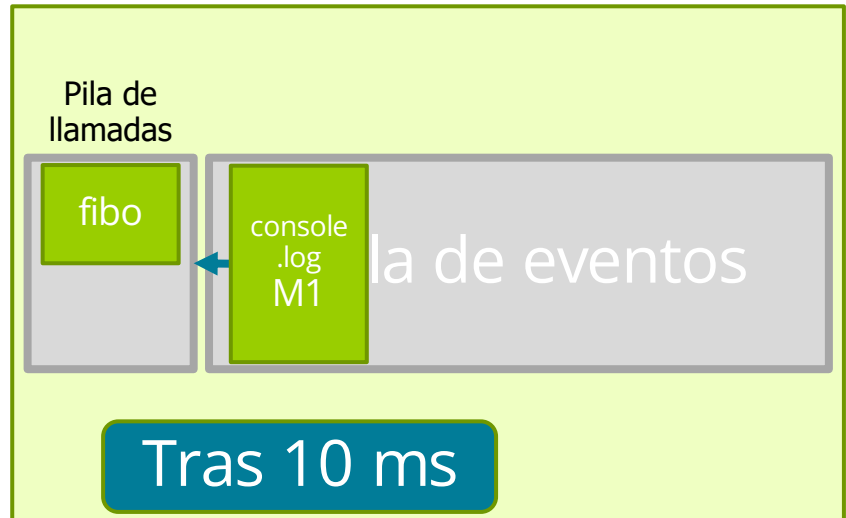


```
C:\> node turnQueue.js  
Starting the process...
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

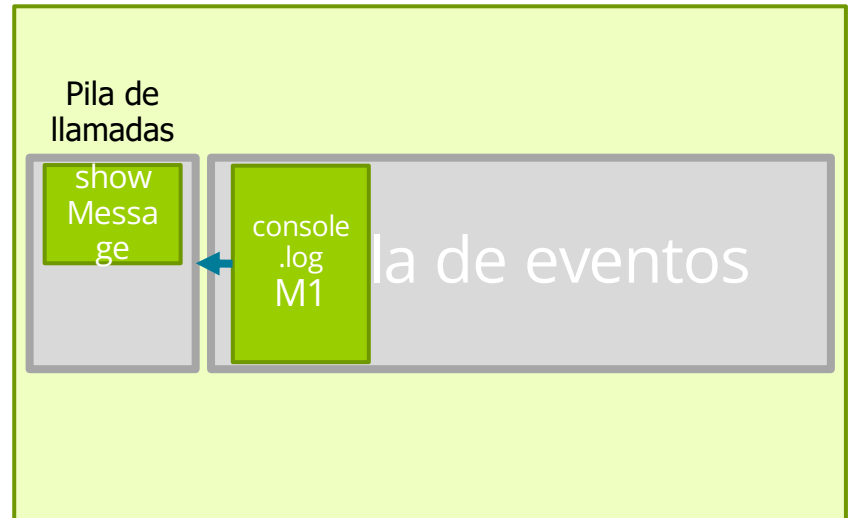


```
C:\> node turnQueue.js  
Starting the process...
```




2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

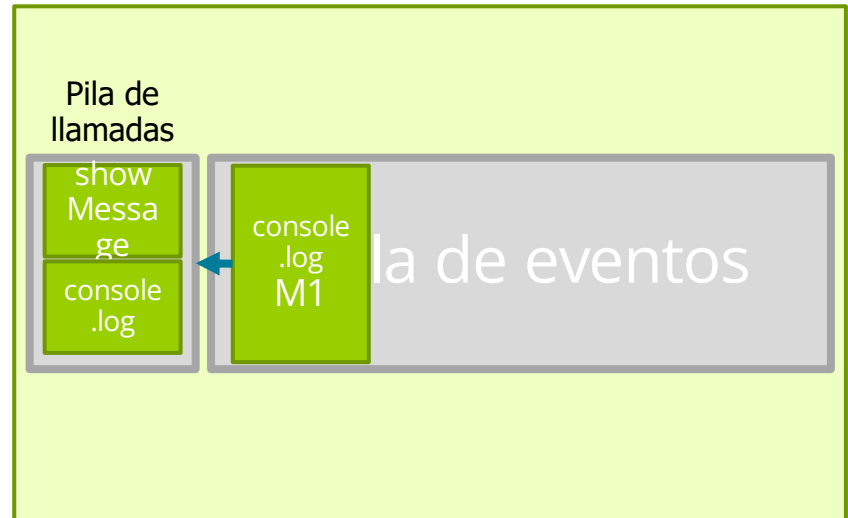


```
C:\> node turnQueue.js  
Starting the process...
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

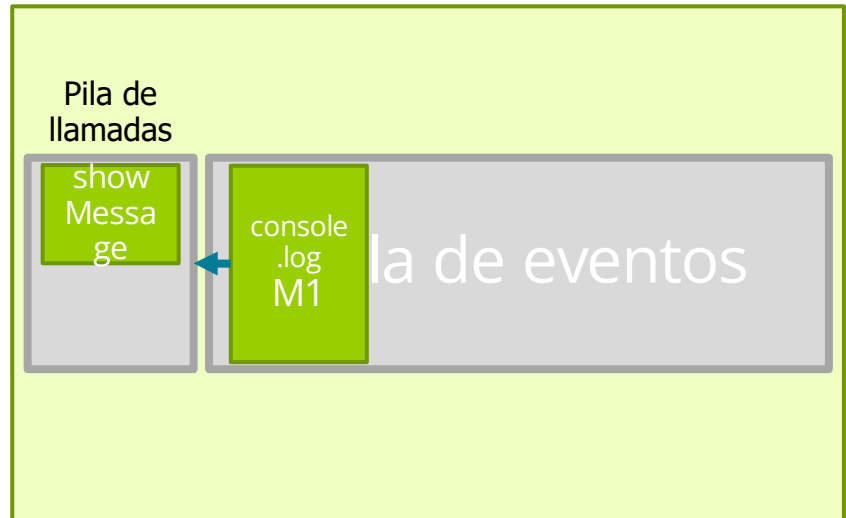


```
C:\> node turnQueue.js  
Starting the process...  
M2: The result is: 165580141
```



2.5 JS. Eventos

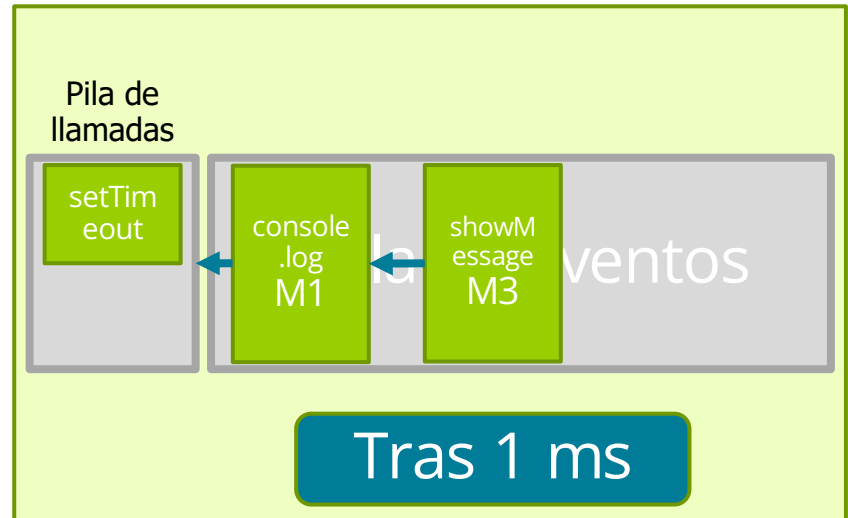
```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```



```
C:\> node turnQueue.js  
Starting the process...  
M2: The result is: 165580141
```

2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

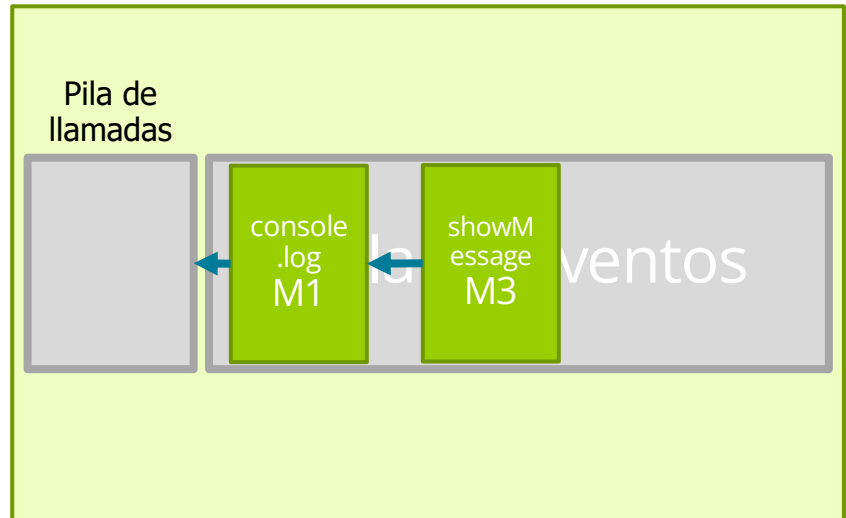


```
C:\> node turnQueue.js  
Starting the process...  
M2: The result is: 165580141
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

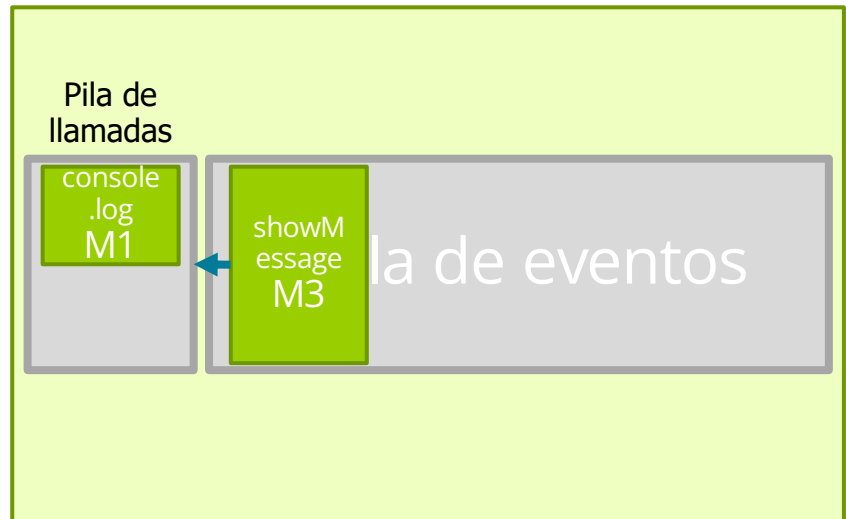


```
C:\> node turnQueue.js  
Starting the process...  
M2: The result is: 165580141
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```



```
C:\> node turnQueue.js  
Starting the process...  
M2: The result is: 165580141  
M1: My first message...
```



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

Pila de
llamadas

showM
essage
M3

Cola de eventos

```
C:\> node turnQueue.js
```

Starting the process...

M2: The result is: 165580141

M1: My first message...



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

Pila de
llamadas

showM
essage
M3

console
.log

Cola de eventos

```
C:\> node turnQueue.js
```

Starting the process...

M2: The result is: 165580141

M1: My first message...

M3: The result is: 165580141



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

Pila de
llamadas

showM
essage
M3

Cola de eventos

```
C:\> node turnQueue.js
```

Starting the process...

M2: The result is: 165580141

M1: My first message...

M3: The result is: 165580141



2.5 JS. Eventos

```
function fibo(n) {  
  return (n<2) ? 1 : fibo(n-2) + fibo(n-1)  
}  
  
function showMessage(m, u) {  
  console.log(m + ": The result is: " + u)  
}  
  
console.log("Starting the process...")  
  
// Wait for 10 ms in order to show the  
message...  
setTimeout( function() {  
  console.log("M1: My first message...")  
}, 10)  
  
// Several seconds are needed in order to  
// complete this call: fibo(40)  
let j = fibo(40)  
  
// M2 is written before M1 is shown. Why?  
showMessage("M2", j)  
  
// M3 is written after M1. Why?  
setTimeout( function() {  
  showMessage("M3", j)  
}, 1)
```

Pila de
llamadas

Cola de eventos

```
C:\> node turnQueue.js  
Starting the process...  
M2: The result is: 165580141  
M1: My first message...  
M3: The result is: 165580141  
C:\>
```



2.6 JS. Callbacks

- ▶ callback function =
 - ▶ *"Referencia a una función que se pasa como argumento a otra para que esta última la invoque cuando finalice su ejecución"*
 - ▶ Ejemplo. Sea un método *fadeOut* para hacer desaparecer progresivamente un elemento de pantalla.
 - Se invoca como *elemento.fadeIn(velocidad, function() {...})*
 - La función que se pasa como segundo argumento es una función *callback* que se invoca cuando se completa la desaparición
 - ▶ Otro ejemplo. La función *writingClosure(4)* es el *callback* de la función *setTimeout* en:

```
setTimeout(writingClosure(4), 4000)
```
- ▶ La función *callback* permite invocación asincrónica:
 - ▶ desde **A** se invoca **B(args, C)**, siendo **C** una función *callback*
 - ▶ cuando **B** termina, invoca **C** (sirve para avisar de la finalización y comunicar el resultado)
- ▶ Como excepción *setTimeout* coloca el *callback* como primer argumento



2.6 JS. Callbacks

```
const fs = require('fs')
fs.writeFileSync('data1.txt', 'Hello Node.js')
fs.writeFileSync('data2.txt', 'Hello everybody!')

function callback(err, data) {
  if (err) console.error('---\n' + err.stack)
  else console.log('---\nFile content is:\n' + data.toString())
}

setTimeout(function(){fs.readFile('data1.txt', callback)}, 3000)
fs.readFile('data2.txt', callback)
fs.readFile('data3.txt', callback)
console.log("root(2) =", Math.sqrt(2))
```

- ▶ Los argumentos para **callback** son suministrados por la función invocada (en ese caso **readFile**)
 - ▶ Consultar documentación de NodeJS sobre File System -> [readFile](#)

```
root(2) = 1.4142135623730951
---
Error: ENOENT: no such file or
directory, open '... data3.txt'
    at Error (native)
---
File content is:
Hello everybody!
---
File content is:
Hello Node.js
```



2.6.1 JS. Callbacks. Limitaciones

- ▶ No se restringe el anidamiento de *callbacks*. Pero hay limitaciones en la práctica:
 - ▶ Excepciones en *callbacks* anidados. Cuando no se trata una excepción, se propaga a la operación invocadora.
 - ▶ Si no garantizamos un tratamiento uniforme en todas las operaciones,

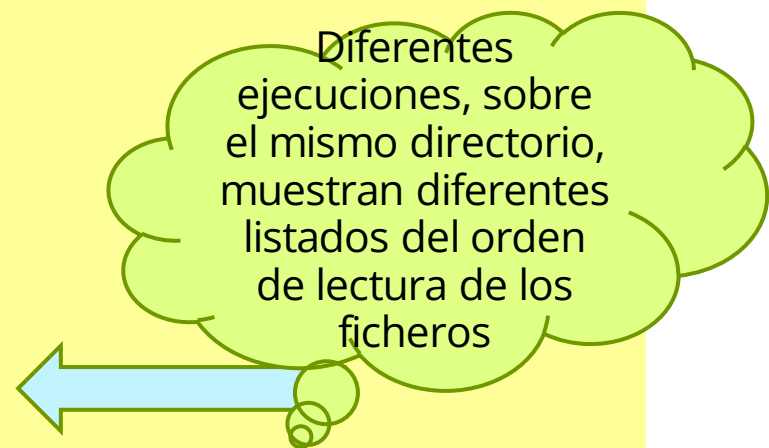
```
fs.access(fileName, function(err) {  
  if (err) {  
    console.log(fileName + " does not exist!")  
  } else {  
    fs.stat(fileName, function(error, stats) {  
      fs.open(fileName, "r", function(error, fd) {  
        let buffer = Buffer.alloc(stats.size)  
        fs.read(fd, buffer, 0, buffer.length, null,  
function(error, bytesRead, buffer) {  
          let data = buffer.toString("utf8", 0, buffer.length)  
          console.log(data)  
          // fs.closeSync(fd)  
          fs.close(fd, function(er) {if (er) console.error(er)})  
        })  
      })  
    })  
  }  
})
```

2.6.1 JS. Callbacks. Limitaciones

▶ Otras limitaciones

- ▶ Dificultad para seguir el código. Orden de ejecución no siempre resulta intuitivo.
- ▶ Incertidumbre sobre el turno en que se ejecutará el *callback*. No podemos confiar en la ejecución en un turno concreto.

```
fs.readdir('.', function(err, files) {  
  let count = files.length  
  let results = {}  
  files.forEach(function(filename) {  
    fs.readFile(filename, function(err, data) {  
      console.log(filename, 'has been read')  
      results[filename] = data  
      count--  
      if (count <= 0) {  
        console.log('\nTOTAL:', files.length, 'files have been read')  
      }  
    })  
  })  
})
```



Diferentes
ejecuciones, sobre
el mismo directorio,
muestran diferentes
listados del orden
de lectura de los
ficheros



2.6.2 JS. Asincronía mediante promesas

- ▶ Se puede modelar la ejecución asincrónica utilizando promesas en lugar de *callbacks*
 - ▶ Invocación de las operaciones sigue el formato tradicional (fácil de leer)
 - ▶ No hay un argumento *callback*
 - ▶ El resultado es un objeto promesa (en inglés '*promise*'). Una promesa...
 - ▶ Representa un valor futuro sobre el que podemos asociar operaciones y gestionar errores
 - ▶ Está en uno de los siguientes estados
 - **pendiente**.- es el estado inicial. La operación todavía no ha concluido (resultado desconocido)
 - **resuelta**.- la operación ha concluido y podemos acceder al resultado. Es un estado final que no puede cambiar
 - **rechazada**.- la operación ha terminado con error. Se acompaña de una razón
 - **satisfecha**.- la operación ha terminado con éxito. Se acompaña un valor
 - ▶ Asociamos a cada estado final una función que se ejecuta cuando el hilo 'principal' finaliza la acción actual
 - ▶ Queda pendiente como evento futuro para un turno posterior

2.6.2 JS. Asincronía mediante promesas

- ▶ **Creación:** Mediante el constructor `Promise()`
 - ▶ `Promise((resolutionFunc, rejectionFunc) => {...})`
 - ▶ `resolutionFunc(param)`
 - *Callback* a utilizar cuando la promesa finalice con éxito. Recibe un argumento.
 - ▶ `rejectionFunc(param)`
 - *Callback* a utilizar cuando se rechace. Recibe un argumento.

```
const fs=require('fs') // This program is stored in file "readfile.js"
```

```
function readFilePromise(filename) {
```

```
  return new Promise( (resolve, reject) => {
```

```
    fs.readFile(filename, (err, data) => {
```

```
      if (err) reject(err+')
```

```
      else resolve(data+')
```

```
    })
```

```
  })
```

```
}
```

```
readFilePromise("readfile.js").then(console.log, console.error)
```

```
readFilePromise("doesntExist.js").then(console.log, console.error)
```

Función asincrónica a convertir en promesa.

En caso de error, se rechaza la promesa.

Si no hay errores, la promesa se resuelve.

2.6.2 JS. Asincronía mediante promesas

- ▶ **Gestión:** Operaciones `then()` y `catch()`
 - ▶ `then(onFulfilled [, onRejected])`:
 - ▶ Permite especificar la gestión a realizar en la resolución de la promesa.
 - El *callback* `onFulfilled` se invoca cuando la promesa se satisfaga.
 - El *callback* `onRejected` se invoca cuando la promesa se rechace.
 - ▶ `catch(onRejected)`:
 - ▶ El *callback* `onRejected` se invocará cuando la promesa se rechace.
 - ▶ Tanto `then()` como `catch()` devuelven una promesa como resultado.
 - ▶ De esa manera, resulta sencillo "encadenar" diferentes etapas en la gestión.
 - Puede dejarse un solo `catch()` al final de la cadena para gestionar todos los rechazos.



2.6.2 JS. Asincronía mediante promesas

- ▶ **Gestión de múltiples promesas:** Operaciones `all()` y `any()`
 - ▶ `all(promiseArray)` devuelve una única promesa que:
 - ▶ Se satisfará cuando todas las promesas en `promiseArray` se hayan resuelto correctamente.
 - ▶ Se rechazará tan pronto como alguna de las promesas en `promiseArray` haya sido rechazada.
 - ▶ `any(promiseArray)` devuelve una única promesa que:
 - ▶ Se satisfará tan pronto como alguna de las promesas en `promiseArray` se haya resuelto correctamente.
 - ▶ Se rechazará cuando todas las promesas en `promiseArray` hayan sido rechazadas.



2.6.2 JS. Asincronía mediante promesas

```
// Prints the length of multiple files on screen. All file names are given as arguments.
const fsp = require("fs").promises
// Check for the names of the files to be read.
if (process.argv.length < 3) {console.log("Introduce several file names as arguments, please!")
  return}
let myFiles=[], names=process.argv.slice(2) // Array of promises, Array of file names.
let numFiles=names.length, allLength=0      // Number of files to be processed, accumulated length
for (let i=0; i<numFiles; i++) {
  myFiles[i]=fsp.readFile(names[i],"utf8") // Generate the promises
}

function showAll( ) {console.log( "Total: " + allLength )} // Print the total length.

function showLength( name ) {// Print the length of the file contents.
  return function( contents ) {
    allLength += contents.length
    console.log( name + ": " + contents.length ) // Print the name of the corresponding file
    and its length
  }}

function showError( err ) {console.log(err.message)} // Print information about errors

function showFinalError( err ) // The same, but for the ".all()" case
  {console.log( "Errors accessing some files. Unable to compute total length.")}

Promise.all(myFiles).then(showAll, showFinalError) // Show the summary
// Show the results.
for (let i=0; i<numFiles; i++) {myFiles[i].then(showLength( names[i] ), showError)}
```

2.6.2 JS. Asincronía mediante promesas

▶ **async/await**

- ▶ **Objetivo:** Controlar cuándo gestionar la resolución de cada promesa.
- ▶ **async:** Este calificador precede a la declaración de una función e indica que retornará una promesa.
 - ▶ Si se intenta retornar un valor, se convertirá en promesa satisfecha.
- ▶ **await:** Esta palabra reservada precede a la invocación de una función que devuelve una promesa y **espera** a que esta se resuelva.
 - ▶ Si se satisface, retorna su resultado.
 - ▶ Si se rechaza, genera una excepción, que puede ser gestionada con:
 - Un **try {...} catch(ex) {...}**
 - La operación **catch()**
 - ▶ **await** solo puede utilizarse dentro de funciones **async**



2.6.2 JS. Asincronía mediante promesas

```
// Variation of program in slide 40...
const fs=require('fs')

function readFilePromise(filename) {
  return new Promise( (resolve,reject) => {
    fs.readFile(filename, (err,data) => {
      if (err) reject(err+'')
      else resolve(data+'')
    })
  })
}

async function readTwoFiles() {
  try {
    console.log(await readFilePromise("readfile.js"))
    console.log(await readFilePromise("doesntExist.js"))
  } catch (err) {
    console.error(err+'')
  }
}

readTwoFiles()
```



2.6.2 JS. Asincronía mediante promesas

```
// Variation of program in previous slide...
const fs=require('fs')

function readFilePromise(filename) {
  return new Promise( (resolve,reject) => {
    fs.readFile(filename, (err,data) => {
      if (err) reject(err+'')
      else resolve(data+'')
    })
  })
}

async function readTwoFiles() {
  console.log(await readFilePromise("readfile.js"))
  console.log(await readFilePromise("doesntExist.js"))
}

// We may use catch() here, instead of try/catch in readTwoFiles().
readTwoFiles().catch( console.error )
```

2.6.3 JS. Ejemplo callbacks vs promesas

- ▶ Lectura asincrónica de un fichero
 - ▶ La versión basada en promesas necesita que la función asincrónica (en este caso `fsp.readFile`) devuelva una promesa.

| Callbacks | Promesas |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>fs.readFile('jsonFILE', function (error, text) { if (error) { console.error(error) } else { try { const obj = JSON.parse(text) console.log(JSON.stringify(obj)) } catch(e) { console.error(error) } } })</pre> | <pre>const fsp = require('fs').promises fsp.readFile('jsonFILE') .then(function(text) { const obj = JSON.parse(text) console.log(JSON.stringify(obj)) }) .catch(function(error) { console.error(error) })</pre> |



3.1 NodeJS. Introducción

Es un entorno orientado a la creación de aplicaciones para internet en el contexto del servidor

- ▶ Define interfaces y entorno, utilidades comunes, intérprete, gestión de paquetes, ...
 - ▶ Construido sobre el entorno de ejecución V8 (JavaScript para Chrome)
 - ▶ Pero independiente (no está incrustado en ningún navegador web)
- ▶ Diseñado para desarrollo rápido de aplicaciones escalables
 - ▶ Modelo de E/S no bloqueante dirigido por eventos
 - ▶ bucle de procesamiento de eventos, ejecutado por un único hilo
 - ▶ Modelo de concurrencia basado en eventos y *callbacks* (mismo modelo que JavaScript)



3.1 NodeJS. Introducción (cont.)

- ▶ Útil para
 - ▶ Desarrollar componentes en la parte servidora
 - ▶ Aplicaciones no críticas que toleran ciertos retardos eventuales
 - ▶ Aplicaciones que requieren interfaces ligeras REST/JSON
 - ▶ Aplicaciones monopágina (interactúan con el servidor mediante AJAX)
 - ▶ Datos por streaming
 - ▶ Comunicación.- aplicaciones de mensajería instantánea, juegos multijugador, etc.

- ▶ Referencias
 - ▶ Intérprete: <http://nodejs.org/download/>
 - ▶ Documentación: <http://nodejs.org/api/>
 - ▶ Repositorio en GitHub (<https://github.com/nodejs>)
 - ▶ Conferencias: <http://www.nodeconf.com/> y <http://jsconf.com>
 - ▶ Apadrinado por la compañía Joyent, <http://joyent.com>



3.1 NodeJS. Introducción (cont.)

- ▶ Modularidad
 - ▶ Usando `require()` se permite incluir otros módulos en un programa
- ▶ Los métodos ofrecidos en los módulos de NodeJS permiten interacción asincrónica
 - ▶ El método retorna el control de inmediato, y el resultado se comunica mediante "*callback*"
 - ▶ Un único hilo de ejecución ...
 - ▶ No hay objetos compartidos ni secciones críticas
 - Elimina los peligros de la programación concurrente tradicional
 - Pero para evitar sorpresas en la actualización de las variables hay que controlar en qué turno llega cada *callback*
 - ▶ Que no se bloquee
 - Incluso al utilizar operaciones de Entrada/Salida u otros servicios del sistema que normalmente acarrearán suspensión
 - ▶ Pero la mayoría de los métodos también tienen una versión sincrónica (sin "*callbacks*")
 - ▶ Ej `fs.readFile()` es asincrónico, pero también existe `fs.readFileSync()`



3.2 NodeJS. Implantación de la asincronía

Los programadores perciben un único hilo, pero internamente...

- ▶ Mantiene una cola de eventos ordenados temporalmente
 - ▶ Contiene contextos de función preparadas para ejecución
 - ▶ En cada momento el soporte en ejecución extrae y ejecuta el primer elemento de la cola
 - ▶ Eso define un turno
 - ▶ NOTA.- `setTimeout(f, retardo)` añade `f` en la cola de eventos (cuando corresponda según el retardo)
 - ▶ `setTimeout(f,0)` añade `f` en ese momento al final de la cola
- ▶ Basa la gestión de los módulos 'asincrónicos' en la biblioteca libuv [7], que mantiene un *pool* de hilos...



3.2 NodeJS. Implantación de la asincronía

Los programadores perciben un único hilo, pero internamente...

- ▶ Mantiene una cola de eventos ordenados temporalmente
- ▶ Basa la gestión de los módulos 'asincrónicos' en la biblioteca libuv [7], que mantiene un pool de hilos
 - ▶ Si se invoca una operación potencialmente bloqueante
 - ▶ Se toma un hilo H del pool, y se deja preparado (con acceso a los argumentos de invocación y el contexto del callback)
 - ▶ El hilo invocante retorna y el programa continúa
 - ▶ Durante su ejecución, H puede que se suspenda en alguna operación, pero no afecta al resto
 - ▶ Cuando H termina
 - crea un contexto para el callback original, con los argumentos necesarios
 - deposita el contexto en la cola de eventos y vuelve al pool
 - ▶ El callback se ejecutará en un turno posterior (cuando pase a ser el primero de la cola de eventos)

3.3 NodeJS. Gestión de módulos

▶ Exports

- ▶ Módulo = fichero que exporta objetos y/o funciones
 - ▶ Debemos indicar qué objetos y funciones se exportan
 - ▶ Los elementos a exportar se asignan como propiedades del objeto "module.exports"
 - o simplemente "exports"

```
// Module Circle.js
exports.area = function(r) {
  return Math.PI * r * r;
}
exports.circumference = function(r) {
  return 2 * Math.PI * r;
}
```

▶ Require

- ▶ Para utilizar funciones/objetos de otro módulo JavaScript
 - ▶ Representamos dicho módulo mediante una constante o una variable
 - ▶ Para ello se invoca 'require()'

```
const cir = require('./Circle.js')

console.log("Área círculo de radio 5: "
  + cir.area(5));
```



3.4 NodeJS. Módulo **events**

- ▶ El módulo **events** permite instanciar generadores de eventos.
 - ▶ Los generadores son instancias de la clase **EventEmitter**.
 - ▶ El generador dispara los eventos mediante su método **emit(event [,arg1][,arg2][...])**.
 - ▶ **emit()** ejecutará los manejadores en este turno.
 - ▶ Si no queremos que ocurra eso:
 - `setTimeout(function(){emit(event,...);},0)`
- ▶ En un generador se podrán registrar "*listeners*":
 - ▶ Mediante la operación **on(event,listener)** del generador.
 - ▶ **addListener(event, listener)** hace lo mismo.
 - ▶ Ante un evento se invocan los **callbacks** de los oyentes registrados
 - ▶ Puede haber múltiples "*listeners*" para un mismo evento.
- ▶ Este módulo está documentado en: <http://nodejs.org/api/events.html>



3.4 NodeJS. Módulo events (ej.)

```
const ev = require('events')
const emitter = new ev.EventEmitter()
const e1 = "print", e2 = "read"

function createListener( eventName ) {
  let num = 0
  return function () {
    console.log("Event " + eventName + " has " +
      "happened " + ++num + " times.")
  }
}

// Listener functions are registered in the event emitter.
emitter.on(e1, createListener(e1))
emitter.on(e2, createListener(e2))
// There might be more than one listener for the same event.
emitter.on(e1, function() {
  console.log( "Something has been printed!!" )
})

// Generate the events periodically...
setInterval(function() {emitter.emit(e1)}, 2000) // First event emitted every 2s
setInterval(function() {emitter.emit(e2)}, 3000) // Second event emitted every 3s
```



3.5 NodeJS. Módulo stream

- ▶ Los objetos Stream permiten acceder a un "canal" de datos.
- ▶ Cuatro variantes:
 1. Readable: Solo lectura.
 2. Writable: Solo escritura.
 3. Duplex: Permite leer y escribir datos.
 4. Transform: Como Duplex, pero sus escrituras dependen de la información leída.
- ▶ Todos son EventEmitter. Eventos manejados:
 - ▶ Readable: readable, data, end, close, error.
 - ▶ Writable: drain, finish, pipe, unpipe.
- ▶ Ejemplos:
 - ▶ Readable: process.stdin, ficheros, peticiones HTTP en servidor, respuestas HTTP en cliente...
 - ▶ Writable: process.stdout, process.stderr, ficheros, peticiones HTTP en cliente, respuestas HTTP en servidor...
 - ▶ Duplex: sockets TCP, ficheros...
- ▶ Documentación accesible en: <http://nodejs.org/api/stream.html>



3.5 NodeJS. Módulo **stream**. Ejemplo

- ▶ Versión interactiva del cálculo de la circunferencia a partir del radio.
- ▶ `process.stdin` es un *stream* "Readable".

```
const st = require('./Circle.js')
const os = require('os')
process.stdout.write("Radius of the circle: ")
process.stdin.resume() // Needed for initiating the reads from stdin.
process.stdin.setEncoding("utf8") // ... for reading strings
// Endless loop. Every time we read a radius its circumference is printed and a new
// radius is requested
process.stdin.on("data", function(str) {
  // The string "str" has been read. Remove its trailing newline.
  let rd = str.slice(0, str.length - os.EOL.length)
  console.log("Circumference for radius " + rd + " is " +
    st.circumference(rd))
  console.log(" ")
  process.stdout.write("Radius of the circle: ")
})
// The "end" event is generated when STDIN is closed. [Ctrl]+[D] in UNIX.
process.stdin.on("end", function() {console.log("Terminating...")})
```



3.6 NodeJS. Módulo net

- ▶ Facilita algunas clases para trabajar con sockets TCP:
 - ▶ **net.Server:** Es un servidor TCP.
 - ▶ Debe ser generado utilizando `net.createServer([options],[connectionListener])`.
 - "connectionListener", en caso de utilizarse, recibirá como argumento un Socket sobre el que se habrá establecido ya la conexión.
 - ▶ Gestiona los eventos: listening, connection, close, error.
 - ▶ **net.Socket:** Socket TCP.
 - ▶ A generar mediante "new net.Socket()" o con "net.connect(options [,listener])" o "net.connect(port [,host][,listener])"
 - ▶ Implanta un Duplex Stream.
 - ▶ Gestiona los eventos: connect, data, end, timeout, drain, error, close.
- ▶ Documentación disponible en: <http://nodejs.org/api/net.html>



3.6 NodeJS. Módulo net. Ejemplo 1

Servidor

```
const net = require('net')

let server = net.createServer(
  function(c) { // 'connection' listener
    console.log('server connected')
    c.on('end', function() {
      console.log('server disconnected')
    })
    // Send "Hello" to the client.
    c.write('Hello\r\n')
    // With pipe() we write to Socket 'c'
    // what is read from 'c'.
    c.pipe(c)
  }) // End of net.createServer()

server.listen(9000,
  function() { // 'listening' listener
    console.log('server bound')
  })
```

Cliente

```
const net = require('net')

// The server is in our same machine.
let client = net.connect({port: 9000},
  function() { // 'connect' listener
    console.log('client connected')
    // This will be echoed by the server.
    client.write('world!\r\n')
  })

client.on('data', function(data) {
  // Write the received data to stdout.
  console.log(data.toString())
  // This says that no more data will be
  // written to the Socket.
  client.end()
})

client.on('end', function() {
  console.log('client disconnected')
})
```



3.6 NodeJS. Módulo net. Ejemplo 2

Servidor

```
const net = require('net')
let server = net.createServer(
  function(c) {
    console.log('server connected')
    c.on('end', function() {
      console.log('server disconnected')
    })
    c.on('error', function() {
      console.log('some connect. error')
    })
    c.on('data', function(data) {
      console.log('data from client: '
        + data.toString())
      c.write(data)
    })
  }) // End of net.createServer()
server.listen(9000,
  function() {
    console.log('server bound')
  })
```

Cliente

```
const net = require('net')
let cont = 0
// The server is in our same machine.
let client = net.connect({port: 9000},
  function() {
    console.log('client connected')
    client.write(cont + ' world!')
  })

client.on('data', function(data) {
  console.log(data.toString())
  if (cont > 1000) client.end()
  else client.write(++cont + ' world!')
})

client.on('end', function() {
  console.log('client disconnected')
})

client.on('error', function() {
  console.log('some connect. error')
})
```



3.6 NodeJS. Módulo net. Ejemplo 3

Servidor

```
const net = require('net')
let myF = require('./myFunctions')
let end_listener = function() {...}
let error_listener = function() {...}
let bound_listener = function() {...}

let server = net.createServer(function(c) {
  c.on('end', end_listener)
  c.on('error', error_listener)
  c.on('data', function(data) {
    let p = JSON.parse(data)
    let q
    if (typeof(p.num) !== 'number') q = NaN
    else { switch (p.fun) {
      case 'fibo': q = myF.fibo(p.num); break
      case 'fact': q = myF.fact(p.num); break
      default: q = NaN
    } }
    c.write(p.fun+'('+p.num+') = '+q)
  })
})
server.listen(9000, bound_listener)
```

Cliente

```
const net = require('net')
if (process.argv.length !== 4) {...}
let fun = process.argv[2]
let num = Math.abs(parseInt(process.argv[3]))
// The server is in our same machine.
let client = net.connect({port: 9000},
  function() {
    console.log('client connected')
    let request = {"fun":fun, "num":num}
    client.write(JSON.stringify(request))
  })
client.on('data', function(data) {
  console.log(data.toString())
  client.end()
})
client.on('end', function() {
  console.log('client disconnected')
})
client.on('error', function() {
  console.log('some connection error')
})
```



3.7 NodeJS. Módulo http

- ▶ Permite implantar servidores web (y sus clientes).
- ▶ Define las siguientes clases:
 - ▶ http.Server: Es un EventEmitter que modela al servidor web.
 - ▶ http.ClientRequest: Modela una petición al servidor.
 - Es un Writable Stream y un EventEmitter.
 - Eventos: response, socket, connect, upgrade, continue.
 - ▶ http.ServerResponse: Modela una respuesta del servidor.
 - Es un Writable Stream y un EventEmitter.
 - Eventos: close.
 - ▶ http.IncomingMessage: Modela las peticiones que lleguen al servidor y las respuestas asociadas a los ClientRequest.
 - Es un Readable Stream y puede gestionar el evento "close".
- ▶ Documentación disponible en: <http://nodejs.org/api/http.html>



3.7 NodeJS. Módulo http

- ▶ Un servidor web mínimo. Propuesto como ejemplo en: <http://nodejs.org/about/>


```
const http = require('http')
const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  // res is a ServerResponse.
  // Its setHeader() method sets the response header.
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  // The end() method is needed to communicate that both the header
  // and body of the response have already been sent. As a result, the
  // response can be considered complete. Its optional argument may be
  // used for including the last part of the body section.
  res.end('Hello World\n')
})
// listen() is used in an http.Server in order to start listening for
// new connections. It sets the port and (optionally) the IP address.
server.listen(port, hostname, () => {
  console.log('Server running at http://'+hostname+':'+port+'/')
})
```




4. Bibliografía

Básica (Recomendada)

1. Tim Caswell: "Learning JavaScript with Object Graphs". Disponible en: <https://howtonode.org/object-graphs>, 2011.
 - ▶ Aunque en parte hace referencia a versiones anteriores de JavaScript, su descripción de las clausuras es muy interesante.
2.  Tutorials Point: "ES6 (ECMAScript 6) Quick Guide". Disponible en: https://www.tutorialspoint.com/es6/es6_quick_guide.htm, julio 2018
3. Joyent, Inc.: "Node.js v16.17.0 Documentation", disponible en: <https://www.nodejs.org/dist/latest-v16.x/docs/api/>, septiembre 2022.
4. Lydia Hallie: "JavaScript Visualized: Event Loop", disponible en: <https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif/>, enero 2020

Avanzada (No exigible)

5. David Flanagan: "JavaScript: The Definitive Guide", 7ª ed., O'Reilly Media, mayo 2020. ISBN: 978-1491952023.
6.  Marijn Haverbeke: "Eloquent JavaScript", 3ª ed., No Starch Press, 460 pgs., octubre 2018. Disponible en: <https://eloquentjavascript.net/> (mayo 2018)
7. Nikhil Marathe: "An Introduction to libuv (Release 1.0.0)", julio 2013. Disponible en: <http://nikhilm.github.io/uvbook/index.html>
8. Tutorials Point: "ES6 (ECMAScript 6) Tutorial". Disponible en: <https://www.tutorialspoint.com/es6/index.htm>, julio 2018