Práctica 0 – Introducción a JavaScript
Parte 2: Ejemplos

Tecnología de los Sistemas de Información en la Red

Índice

1. Un ejemplo erróneo

- Si no estamos acostumbrados a programar en JavaScript, nuestros primeros programas puede que no funcionen bien.
- Posibles problemas:
  - Algunas funciones podrán proporcionar sus resultados asincrónicamente
    - Al utilizar ***callbacks***
      - es decir, argumentos para una función A que también son funciones y que serán invocadas cuando A finalice su tarea.
      - Si A utiliza alguna llamada al sistema que deje al invocador en estado "suspendido", entonces A necesitará un largo intervalo para terminar.
      - Esto rompe la ejecución normal (es decir, secuencial) de nuestro proceso...
        - Pues los *callbacks* no se ejecutan inmediatamente, sino algo más tarde.
  - En ejecuciones asincrónicas, podremos encontrar valores inesperados en algunas variables.

1. Un ejemplo erróneo

- Supongamos un programa que...:
  - muestra qué fichero es el más grande (y su tamaño en bytes)...
  - ...de una lista de nombres de ficheros recibida como argumentos desde la línea de órdenes.
- Una posible primera versión es:

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:                 +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- …pero esta versión no funciona como esperamos.

1. Un ejemplo erróneo

- Para comprobar si este programa es correcto, deberíamos:
  1. Ejecutarlo con argumentos válidos.
  2. Comprobar su salida.
  3. Si la salida no fuera correcta, seguir una traza de su ejecución.
- Así, en caso de error, podríamos detectar dónde se comporta mal el programa.
- Supongamos que tenemos los siguientes ficheros:
  - A: 2300 bytes
  - B: 180 bytes
  - C: 4500 bytes
  - D: 470 bytes
- Y que ejecutamos el programa (cuyo nombre es "files.js") utilizando esta orden:

  node files A C D B

1. Un ejemplo erróneo

- Y con ello obtenemos los siguientes resultados:

```
$ node files A C D B
The longest file is NONE and its length is 0 bytes.
Processing undefined...
Processing undefined...
Processing undefined...
Processing undefined...
$
```

- Revisemos su traza para entender esa salida...

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:                 +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- Las cuatro primeras líneas hacen lo siguiente:
  1. Importar el módulo "fs" en la constante **fs**.
  2. Asignar los argumentos de la línea de órdenes al vector **args**:
     - ["A","C","D","B"]
  3. Asignar "NONE" a **maxName**.
  4. Asignar 0 a **maxLength**.

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:                 +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- La línea 5 es un bucle "for" con cuatro iteraciones:
  - La variable "i" recibe los valores de 0 a 3 en esas iteraciones.
  - En cada una se llama a la función "readFile()":
    - Utilizando cada argumento de la línea de órdenes como primer argumento en cada llamada.
    - Obsérvese que el tercer parámetro es un *callback*.

1. Un ejemplo erróneo

```
1:  const fs=require('fs')
2:  var args=process.argv.slice(2)
3:  var maxName='NONE'
4:  var maxLength=0
5:  for (var i=0; i<args.length; i++)
6:    fs.readFile(args[i],'utf8',
7:      function(err,data) {
8:        if (!err) {
9:          console.log('Processing'
10:               +' %s...',args[i])
```

Ese *callback* será invocado cuando el fichero correspondiente se haya leído. Así, el proceso reacciona a la finalización de esa operación. En ese momento, se muestra el mensaje "Processing <nombreFichero>…" en la pantalla.

- La línea 5 es un bucle "for" co
  - La variable "i" recibe los valores de
  - En cada una se llama a la función "read
    - Utilizando cada argumento de la línea de enes como primer argumento en cada llamada.
    - Obsérvese que el tercer parámetro es un *callback*.

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:               +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- Pero "readFile()" tiene un comportamiento asincrónico:
  - Los procesos JavaScript se comportan como procesos con un único hilo de ejecución.
  - readFile() se utiliza para leer un fichero.
    - Con ello, el invocador permanece suspendido cuando el sistema operativo (SO) recibe esa llamada.
  - Para evitar la suspensión del proceso, se crea un nuevo hilo en cada llamada a readFile().
    - Ese nuevo hilo interno invoca al SO, permanece suspendido y preparará el contexto de ejecución del *callback* una vez sea reactivado.
    - Ese hilo es gestionado de forma transparente por el módulo "fs". Queda oculto al programador.
  - Mientras tanto, el hilo principal continúa.

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:               +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- Por tanto, el hilo de ejecución principal…
  - Ejecuta todas las iteraciones sin suspenderse.
    - Su ejecución no es interrumpida por los demás hilos de su proceso.
  - Y continúa una vez finalice el bucle "for".

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:                 +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- Así, alcanza la instrucción que sigue al bucle (líneas 18 a 20).
  - Por ello, imprime:

    `The longest file is NONE and its length is 0 bytes.`
    - Pues nadie ha modificado todavía los valores de **maxName** o **maxLength**.
  - Una vez hecho esto, ese hilo ha completado todas sus instrucciones. Por tanto, busca otros "turnos de ejecución":
    - Hay o habrá algunos: aquellos correspondientes a los contextos de ejecución de los *callbacks* que pasen a estar preparados.

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:                 +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- Esos contextos de ejecución de los *callbacks* pasan a preparados cuando finalizan sus llamadas al sistema correspondientes.
  - El tiempo necesario para ello dependerá del tamaño de cada fichero.
  - Así, el orden en que se mostrará cada fichero podrá diferir del orden en que esos nombres aparecían en la línea de órdenes.

1. Un ejemplo erróneo

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: for (var i=0; i<args.length; i++)
 6:   fs.readFile(args[i],'utf8',
 7:     function(err,data) {
 8:       if (!err) {
 9:         console.log('Processing'
10:                 +' %s...',args[i])
11:         if (data.length >
12:           maxLength) {
13:           maxLength = data.length
14:           maxName=args[i]
15:         }
16:       } // if (!err)…
17:     }) // readFile()…
18: console.log('The longest file is '
19: +'%s and its length is %d bytes.',
20:   maxName, maxLength)
```

- De hecho, cuando esos *callbacks* se ejecutan, el hilo principal ya ha finalizado todas las iteraciones del bucle "for" (Hoja 12)
  - Así, ¿cuál es el valor de la variable "i" en ese momento?
    - Es args.length; es decir, 4 en nuestro ejemplo.
    - ¡Pero args[4] es "**undefined**" pues únicamente mantiene nombres de fichero en sus componentes 0 a 3!

1. Un ejemplo erróneo

Y eso explica la salida obtenida:

```
The longest file is NONE and its length is 0 bytes.
Processing undefined...
Processing undefined...
Processing undefined...
Processing undefined...
```

- De hecho, c *allbacks* se ejecutan, el hilo principal ya ha finalizado to raciones del bucle "for" (Hoja 12)
  - Así, ¿cuál es el valo variable "i" en ese momento?
    - Es args.length; es decir, en nuestro ejemplo.
    - ¡Pero args[4] es "**undefined**" pues únicamente mantiene nombres de fichero en sus componentes 0 a 3!

1. Un ejemplo erróneo

- Hay dos problemas principales en este programa:
  1. Los mensajes de traza que muestran el nombre de cada fichero no ofrecen valores correctos:
     - Están basados en el valor de la variable "i", pero ese valor no puede darse como argumento en el *callback*.
     - Por ello, el nombre de fichero es incorrecto.
  2. El teórico mensaje final que reportaría el nombre y tamaño del mayor fichero se muestra antes de lo debido, cuando todavía no hay información válida.
     - Ese mensaje no debería imprimirse en el código que sigue al bucle "for".
       - Pues esas instrucciones se ejecutan antes de que se invoquen los *callbacks*.
     - ¡El mensaje debería mostrarse con alguna instrucción ubicada DENTRO DEL *CALLBACK*!
       - Cuando todos los nombres de fichero hayan sido procesados.
       - Necesitaremos un contador para asegurar esto.

Índice

1. Un ejemplo erróneo
2. El módulo "debug"
3. Una primera versión correcta
4. Segunda versión correcta

2. El módulo "debug"

- En lugar de intentar corregir un programa erróneo sin ayuda, es conveniente añadir mensajes de traza en él.
  - Sin embargo, tras detectar y corregir los errores, solemos eliminar esos mensajes de traza.
  - El módulo 'debug' es una buena ayuda:
    - Permite mostrar los mensajes de traza cuando se solicite.
    - Y ocultarlos por omisión, sin incurrir en un sobrecoste excesivo.
      - De esa manera, ya no es necesario eliminarlos manualmente.
    - Su documentación está disponible en:
    - https://www.npmjs.com/package/debug
  - En nuestro ejemplo, utilizaremos este módulo para averiguar las causas de los resultados incorrectos explicados en la sección anterior.

2. El módulo "debug"

- Para usar ese módulo:
  - Hay que instalarlo, con la orden:

    npm install debug
  - Debe importarse en el programa a depurar...
    - Con una o múltiples líneas de este estilo:

      const deb = require('debug')('label')
      - Donde:
        - El identificador de la constante ('deb' en este ejemplo) se usará como una función que reemplace a console.log() para mostrar los mensajes de traza.
        - El nombre de la etiqueta utilizada en los segundos paréntesis ('label' en este ejemplo) es el valor a utilizar para habilitar esos mensajes desde la línea de órdenes.
          - Al lanzar el proceso, debemos asignar esa etiqueta a la variable de entorno DEBUG.
    - Para definir múltiples partes en el programa a depurar, podremos utilizar múltiples líneas "require", con diferentes etiquetas.
      - Posteriormente, seleccionaremos qué partes integrar en la traza antes de iniciar el proceso.

2. El módulo "debug"

- Supongamos que se van a añadir mensajes de traza al programa descrito en la Sección 1.
- El programa resultante podría ser como este:

```
 1: const var_i=require('debug')('var_i')
 2: const names=require('debug')('names')
 3: const fs=require('fs')
 4: var args=process.argv.slice(2)
 5: var maxName='NONE'
 6: var maxLength=0
 7: for (var i=0; i<args.length; i++) {
 8:     var_i('in loop: %d',i)
 9:     fs.readFile(args[i],'utf8',
10:         function(err,data) {
11:             if (!err) {
12:                 names('Processing %s...', args[i])
13:                 var_i('in callback: %d', i)
14:                 if (data.length>maxLength) {
15:                     maxLength=data.length
16:                     maxName=args[i]
17:                 }
18:             }
19:         })
20: }
21: console.log('The longest file is %s and its'
22:     + ' length is %d bytes.', maxName,
23:     maxLength);
```

2. El módulo "debug"

- En este ejemplo, se han añadido dos etiquetas:
  - var_i: Muestra mensajes con el valor de la variable "i" en el cuerpo del bucle y en el cuerpo del *callback*.
  - names: Reemplaza a los mensajes originales de traza.

```
 1: const var_i=require('debug')('var_i')
 2: const names=require('debug')('names')
 3: const fs=require('fs')
 4: var args=process.argv.slice(2)
 5: var maxName='NONE'
 6: var maxLength=0
 7: for (var i=0; i<args.length; i++) {
 8:     var_i('in loop: %d',i)
 9:     fs.readFile(args[i],'utf8',
10:         function(err,data) {
11:             if (!err) {
12:                 names('Processing %s...', args[i])
13:                 var_i('in callback: %d', i)
14:                 if (data.length>maxLength) {
15:                     maxLength=data.length
16:                     maxName=args[i]
17:                 }
18:             }
19:         })
20: }
21: console.log('The longest file is %s and its'
22:     + ' length is %d bytes.', maxName,
23:     maxLength);
```

2. El módulo "debug"

- Ahora, cuando ejecutemos ese programa no obtendremos ningún mensaje de traza por omisión:
  - Por ello, si ejecutamos esta orden:

    node files A C D B
  - Obtendremos únicamente esta salida:

```
The longest file is NONE and its length is 0 bytes.
```

2. El módulo "debug"

- Pero podremos elegir cuáles son los mensajes de traza a mostrar.
- Para ello, asignaremos sus etiquetas a la variable de entorno DEBUG.
- Empecemos con los mensajes originales, mostrados en la primera versión del programa. Necesitaremos esta declaración en el *shell*:

  export DEBUG=names # En Windows, usaríamos: **set DEBUG=names**

- Tras ello, lanzaríamos el proceso:
  - Así, al dar la orden:

    node files A C D B

  - Obtendríamos esta salida:

```
The longest file is NONE and its length is 0 bytes.
  names Processing undefined... +0ms
  names Processing undefined... +2ms
  names Processing undefined... +0ms
  names Processing undefined... +0ms
```

2. El módulo "debug"

- Pero necesitaremos más mensajes para entender por qué falla.
  - Aquellos etiquetados con 'var_i'
  - Para ello, podremos usar o bien...

    export DEBUG=names,var_i
  - …o:

    export DEBUG=*
- Y ya podremos iniciar ese proceso:
  - Con la orden **node files A C D B** obtendremos ahora:

```
  var_i in loop: 0 +0ms
  var_i in loop: 1 +4ms
  var_i in loop: 2 +0ms
  var_i in loop: 3 +1ms
The longest file is NONE and its length is 0 bytes.
  names Processing undefined... +0ms
  var_i in callback: 4 +4ms
  names Processing undefined... +1ms
  var_i in callback: 4 +1ms
  names Processing undefined... +0ms
  var_i in callback: 4 +1ms
  names Processing undefined... +1ms
  var_i in callback: 4 +0ms
```

2. El módulo "debug"

Las primeras cuatro líneas de la salida muestran que el bucle "**for**" se ha ejecutado en primer lugar. En él la variable "**i**" se ha ido incrementando como se esperaba.

```
  var_i in loop: 0 +0ms
  var_i in loop: 1 +4ms
  var_i in loop: 2 +0ms
  var_i in loop: 3 +1ms
The longest file is NONE and its length is 0 bytes.
  names Processing undefined... +0ms
  var_i in callback: 4 +4ms
  names Processing undefined... +1ms
  var_i in callback: 4 +1ms
  names Processing undefined... +0ms
  var_i in callback: 4 +1ms
  names Processing undefined... +1ms
  var_i in callback: 4 +0ms
```

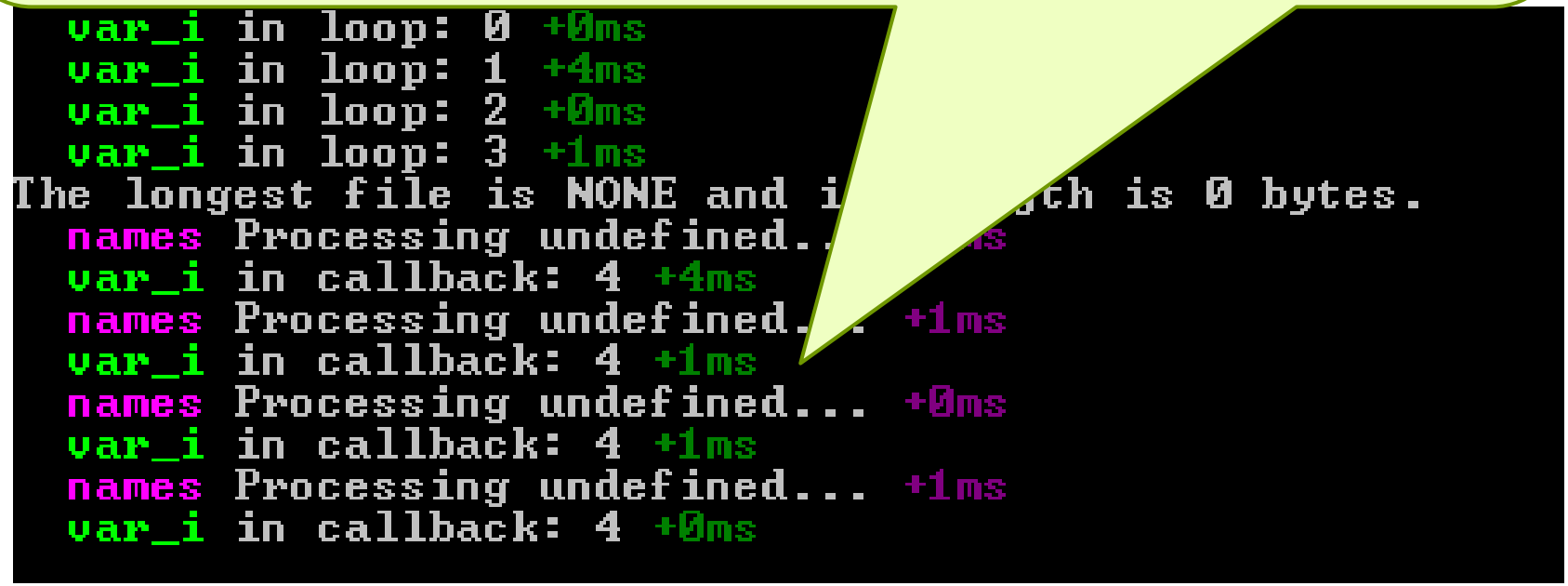
2. El módulo "debug"

Cuando el bucle finaliza, el proceso muestra el mensaje gestionado en las líneas 21 a 23 del programa.
Sin embargo, en ese momento ninguno de los *callbacks* se ha ejecutado todavía. Esto explica los resultados incorrectos del programa.

```
  var_i in loop: 0 +0ms
  var_i in loop: 1 +4ms
  var_i in loop: 2 +0ms
  var_i in loop: 3 +1ms
The longest file is NONE and its length is 0 bytes.
  names Processing undefined... +0ms
  var_i in callback: 4 +4ms
  names Processing undefined... +1ms
  var_i in callback: 4 +1ms
  names Processing undefined... +0ms
  var_i in callback: 4 +1ms
  names Processing undefined... +1ms
  var_i in callback: 4 +0ms
```

2. El módulo "debug"

Finalmente, se ejecutan los *callbacks*, pero cuando esto ocurre la variable "**i**" tiene un valor inesperado, 4, pues el bucle "**for**" ya ha finalizado.
Además, en ese momento el mensaje que debía mostrar los resultados ya ha sido impreso, ofreciendo valores erróneos.



2. El módulo "debug"

- Esta sección ha mostrado cómo:
  - Los mensajes de traza pueden ser gestionados fácilmente con el módulo “debug”.
    - Así es más fácil identificar los errores en nuestros programas.
- Una vez los errores han sido localizados y corregidos, los mensajes de traza pueden seguir integrados en el código...
  - Pues tarde o temprano se ampliará o modificará el programa y otros errores podrían introducirse en esas extensiones.
- Para ocultar los mensajes de traza debemos asegurar que la variable de entorno DEBUG no tenga valor.
  - DEBUG=

Índice

1. Un ejemplo erróneo
2. El módulo "debug"
3. Una primera versión correcta
4. Segunda versión correcta

3. Una primera versión correcta

- Recordemos los problemas identificados en la Sección 1:
  - El segundo problema es fácil de resolver.
    - Ya se dio una guía en la Hoja 16.
  - El primer problema se debe a que el *callback* no puede acceder a "i" en la iteración del bucle, sino después.
    - Los *callbacks* de las bibliotecas tienen una signatura ya definida.
      - No tiene sentido añadir parámetros o argumentos.
      - Por tanto, necesitamos algún mecanismo para pasar el nombre de fichero apropiado al código del *callback*.
        - La solución se basa en el ámbito de declaración de las variables.
          - Una función puede acceder a toda variable o parámetro declarado en un ámbito más externo.
          - Así, convendrá escribir una función que mantenga el nombre de fichero en alguno de sus parámetros y devuelva como resultado el *callback* a utilizar.
            - Con ello, el código del *callback* podrá conocer el nombre de fichero.
            - Eso es una CLAUSURA.

3. Una primera versión correcta

- Este programa resuelve los problemas que hemos visto. Así:
  - Escribe cuál es el fichero más grande (y su tamaño)...
  - …de una lista de nombres de fichero recibidos como argumentos.
- Su código es:

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

- Ahora, la función generator() resuelve el segundo problema:
  - Recibe el nombre de fichero en su parámetro.
    - Por tanto, todo su código puede utilizarlo.
  - Y devuelve una función cuya signatura coincide con la de los *callbacks* de readFile(). Esa función procesará el resultado de esa lectura.
- Además, en su línea 16 comprueba si se han procesado todos los ficheros.
  - De ser así, muestra el mensaje con los resultados y el proceso termina ahí.

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

- En la línea 24, cuando se especifica el tercer argumento de readFile():
  - No pasamos un puntero a la función "generator" sino que LA LLAMAMOS pasando el nombre de fichero apropiado como argumento.
  - Eso genera la función a utilizar como *callback*.

3. Una primera versión correcta

- Utilicemos esta nueva versión del programa, con los mismos argumentos explicados en la Hoja 5:

```
$ node files A C D B
Processing D...
Processing B...
Processing A...
Processing C...
The longest file is C and its length is 4500 bytes.
$
```

- Sigamos una traza para entender por qué ahora la salida proporcionada es correcta…

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

- Sus primeras 5 líneas hacen lo siguiente:
  1. Importar el módulo "fs" en la constante **fs**.
  2. Asignar los argumentos de la línea de órdenes al vector **args**.
  3. Asignar "NONE" a **maxName**.
  4. Asignar 0 a **maxLength**.
  5. Inicializar el contador de nombres procesados a cero.

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

- Las líneas 6 a 21 definen la función generator().
  - Pero esa función no se invoca todavía.
  - Cuando se invoque, retornará como resultado una función anónima.
    - Esa función tiene dos parámetros (*err* y *data*) que casan con la signatura exigida a los *callbacks* de fs.readFile().
- De momento, la ejecución proseguirá en la línea 22.

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:           maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

- Las líneas 22 a 24 definen un bucle con tantas iteraciones como nombres de fichero. En cada iteración:
  - Se llama a la función readFile(). Así, el proceso inicia la lectura del fichero correspondiente.
  - Su tercer parámetro es una llamada a la función generator().
    - Con ella, el nombre del fichero procesado en cada iteración es recibido en el parámetro 'name', accesible al código del *callback*.
    - Por tanto, esto resuelve el primer problema citado en la Hoja 16.

3. Una primera versión correcta

- Cuando todas las iteraciones han terminado, el hilo inicial ya no tiene más instrucciones a ejecutar.
  - Aparentemente, ha terminado todo el programa.
  - Pero el proceso en ejecución todavía no ha terminado...
    - …¡porque hay cuatro llamadas a readFile() que todavía no han finalizado!
- Esas llamadas a readFile() terminarán en algún momento.
  - Cada vez que termine alguna, su *callback* se ejecutará.
  - Se ha llamado a readFile() con esta secuencia de nombres: "A", "C", "D" y "B".
    - Pero "C" es el fichero mayor y "B" el más pequeño.
      - ¡Por lo que es impredecible su orden de finalización!
- Continuemos con la traza…

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

Según la salida mostrada en la Hoja 34, el primer fichero que finaliza su readFile() es D. Su tamaño es 470 bytes.
El *callback* muestra esto en la pantalla:
Processing D…
…y después modifica el valor de maxLength (dejándolo a 470) y maxName (“D”). La variable “counter” se incrementa.

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

Posteriormente finaliza B. Su tamaño es 180 bytes. Su *callback* muestra esto en la pantalla:

Processing B…

…pero ahora maxLength y maxName no se modifican. La variable "counter" se incrementa. Tendrá valor 2. No se muestra nada más.

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<args.length; i++)
23:   fs.readFile(args[i], 'utf8',
24:     generator(args[i]))
```

Después finaliza A. Su tamaño es 2300 bytes. Su *callback* muestra:

Processing A…

…y se modifican maxLength (2300) y maxName ("A"). La variable "counter" se incrementa (valor 3). No se muestra nada más.

3. Una primera versión correcta

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: function generator(name) {
 7:   return function(err,data) {
 8:     if (!err) {
 9:       console.log('Processing %s...',
10:         name)
11:       if (data.length>maxLength) {
12:         maxLength=data.length
13:         maxName=name
14:       }
15:     }
16:     if (++counter==args.length)
17:       console.log('The longest file is %s '
18:       +'and its length is %d bytes.',
19:       maxName, maxLength)
20:   }
21: }
22: for (var i=0; i<a                    -)
23:   fs.readFile(
24:     gen
```

Por último finaliza C (4500 bytes). Su *callback* escribe:

Processing C…

…y se modifican maxLength y maxName. La variable "counter" se incrementa (valor 4). Por tanto, se muestra el mensaje con los resultados, indicando que el mayor fichero es C.

Como ya no hay otros ficheros, el proceso finaliza aquí.

Índice

1. Un ejemplo erróneo
2. El módulo "debug"
3. Una primera versión correcta
4. Segunda versión correcta

4. Segunda versión correcta

- Los programas vistos hasta ahora han usado "**var**" para declarar variables.
  - ¡El primer problema discutido en la Sección 1 estaba causado parcialmente por esas declaraciones!
  - Una solución más compacta a ese problema consiste en utilizar "**let**" en lugar de "**var**".
    - "**let**" define variable en el ámbito del bloque actual.
      - Un bloque es un grupo de instrucciones encerradas entre un par de llaves {}
      - Así, esas variables pueden utilizarse en ese bloque y otros contenidos en él.
    - Cuando "**let**" se use en el ámbito global, esas variables podrán usarse a partir de ese punto.
      - No definen propiedades del objeto "**global**".
  - Además, la instrucción "**for**" define un bloque implícito que incluye todas las instrucciones del bucle.
    - Así, todas las instrucciones del bucle "recuerdan" cuál ha sido el valor actual de la variable iteradora en esa sentencia "**for**".

4. Segunda versión correcta

- Por tanto, este programa es también correcto y facilita la misma salida que aquel mostrado en la Sección 3:

```
 1: const fs=require('fs')
 2: var args=process.argv.slice(2)
 3: var maxName='NONE'
 4: var maxLength=0
 5: var counter=0
 6: for (let i=0; i<args.length; i++)
 7:   fs.readFile(args[i],'utf8',
 8:     function(err,data) {
 9:       if (!err) {
10:         console.log('Processing %s...',
11:             args[i])
12:         if (data.length>maxLength) {
13:           maxLength=data.length
14:           maxName=args[i]
15:         }
16:       }
17:       if (++counter==args.length)
18:         console.log('The longest file is %s'
19:           +' and its length is %d bytes.',
20:           maxName, maxLength)
21:   })
22:
23:
24:
```

4. Segunda versión correcta

Como "i" se define con "**let**", su ámbito es solo el conjunto de instrucciones de ese bucle.
Cada iteración utiliza una "nueva" definición de "i", con un valor diferente.

```
 1: const fs=
 2: var args=                    lice(2)
 3: var max                      ONE'
 4: var ma                       n=0
 5: var co                       ter=0
 6: for (let i=0; i<args.length; i++)
 7:   fs.readFile(args[i],'utf8',
 8:     function(err,data) {
 9:       if (!err) {
10:         console.log('Processing %s...',
11:             args[i])
12:         if (data.length>maxLength) {
13:           maxLength=data.length
14:           maxName=args[i]
15:         }
16:       }
17:       if (++counter==args.length)
18:         console.log('The longest file is %s'
19:           +' and its length is %d bytes.',
20:           maxName, maxLength)
21:   })
22:
23:
24:
```

4. Segunda versión correcta

Por ello, el *callback* utilizado en cada iteración "recuerda" qué valor de "i" fue utilizado en ella.
Así, escribe el nombre de fichero correcto en las líneas 10 y 11 y lo asigna en la línea 14. Obsérvese que el bucle "**for**" ya habrá terminado cuando se ejecuten estos *callbacks*.

```
 1: const fs=
 2: var args                    e(2)
 3: var max                     E'
 4: var ma
 5: var co                      0
 6: for (l                      0; i<args.length; i++)
 7:   fs.r                      dFile(args[i],'utf8',
 8:     function(err,data) {
 9:       if (!err) {
10:         console.log('Processing %s...',
11:             args[i])
12:         if (data.length>maxLength) {
13:           maxLength=data.length
14:           maxName=args[i]
15:         }
16:       }
17:       if (++counter==args.length)
18:         console.log('The longest file is %s'
19:           +' and its length is %d bytes.',
20:           maxName, maxLength)
21:   })
22:
23:
24:
```