Lab 0 – Introduction to JavaScript Part 2: Examples

Network Information System Technologies

Index

1. An erroneous example

- If we aren't used to JavaScript programming, our first programs may not run as we expect.
- Possible problems:
  - Some functions may provide an asynchronous result.
    - They may use ***callbacks***
      - i.e. arguments to a function A that are functions themselves and that will be invoked when A finishes its work.
      - If A uses any system call that sets the invoker into the "waiting" state, then A's end will arise after quite a long interval.
      - This breaks the regular (i.e. sequential) execution of our program!
        - *Callbacks* are not run immediately, but much later.
  - In asynchronous runs, some variables may hold unexpected values.

1. An erroneous example

- Let us assume a program that:
  - prints which is the longest file (and its size)…
  - …from a list of file names received as command-line arguments.
- A possible first version is:

```
const fs = require('fs')
var args = process.argv.slice(2)
var maxName = 'NONE'
var maxLength = 0
for (var i = 0; i<args.length; i++) {
//
  fs.readFile(args[i],'utf8',
    function(err,data) {
//
      if (!err) {
        console.log('Processing %s...',
          args[i])
          if (data.length>maxLength) {
            maxLength = data.length
            maxName = args[i]
          }
        } // if (!err)…
  }) // readFile()…
} // for
console.log('The longest file is '
  +'%s and its length is %d bytes.',
  maxName, maxLength)
```

- …but this first version doesn't work as we expected!

1. An erroneous example

- In order to check whether that program is correct, we should:
  1. Run it with some valid arguments.
  2. Check its output.
  3. If that output is not correct, follow a trace of its execution.
- Thus, in case of errors, we could detect where the program misbehaves.
- Let us assume that we have the following files:
  - A: 2300 bytes
  - B: 2812500 bytes
  - C: 180 bytes
  - D: 470 bytes
- Then, we run the program (whose name is "files.js") using the following command:

```
node files A B C D
```

1. An erroneous example

- However, its output is the following:

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

- Let us follow its trace in order to understand that output…

1. An erroneous example

```
 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: //
 7:   fs.readFile(args[i],'utf8',
 8:     function(err,data) {
 9: //
10:       if (!err) {
11:         console.log('Processing %s...',
12:           args[i])
13:           if (data.length>maxLength) {
14:             maxLength = data.length
15:             maxName = args[i]
16:           }
17:       } // if (!err)…
18:   }) // readFile()…
19: } // for
20: console.log('The longest file is '
21:   +'%s and its length is %d bytes.',
22:   maxName, maxLength)
```

- Its first four lines do the following:
  1. Import the "fs" module in constant **fs**.
  2. Place the command-line arguments in the **args** array:
     - ["A", "B", "C", "D"]
  3. Set value "NONE" on **maxName**.
  4. Set value 0 on **maxLength**.

1. An erroneous example

```
 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: //
 7:   fs.readFile(args[i],'utf8',
 8:     function(err,data) {
 9: //
10:       if (!err) {
11:         console.log('Processing %s...',
12:           args[i])
13:         if (data.length>maxLength) {
14:             maxLength = data.length
15:             maxName = args[i]
16:         }
17:       } // if (!err)…
18:   }) // readFile()…
19: } // for
20: console.log('The longest file is '
21:   +'%s and its length is %d bytes.',
22:   maxName, maxLength)
```

- Line 5 is a "for" loop with four iterations:
  - Thus, variable "i" gets the values from 0 to 3 in each iteration.
  - In each one, the "readFile()" function is called:
    - Using each command-line argument as the first argument in each call.
    - Note that the third parameter is a *callback*.

1. An erroneous example

```
 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: //
 7:   fs.readFile(args[i],'utf8',
 8:     function(err,data) {
 9: //
10:       if (!err) {
11:         console.log('Processing %s...',
12:           args[i])
```

That callback will be invoked once the corresponding file has been read.
Thus, the program reacts to the fact of such operation completion. At that time, the "Processing <filename>…" message is shown on the screen.

- Line 5 is a "for" loop with fo
  - Thus, variable "i" gets the values fro 4 in each iteration
  - In each one, the "readFile()" functio alled.
    - Using each command-line argument as e first argument in each call.
    - Note that the third parameter is a *callback*.

1. An erroneous example

- But "readFile()" has an asynchronous behaviour:
  - JavaScript processes behave as single-threaded programs.
  - readFile() needs to read a file.
    - Thus, the invoker remains "waiting" once the operating system (OS) receives such a call.
  - To avoid a complete process blocking, a new thread is created on each readFile() call.
    - That new internal thread invokes the OS, remains in the "waiting" state, and will prepare the *callback* invocation context once it is awakened.
    - Such thread is transparently managed by the "fs" module. It is hidden to the programmer.
  - In the meantime, the main execution thread goes on.

1. An erroneous example

```
const fs = require('fs')
var args = process.argv.slice(2)
var maxName = 'NONE'
var maxLength = 0
for (var i = 0; i<args.length; i++) {
//
  fs.readFile(args[i],'utf8',
    function(err,data) {
//
      if (!err) {
        console.log('Processing %s...',
          args[i])
        if (data.length>maxLength) {
            maxLength = data.length
            maxName = args[i]
        }
      } // if (!err)…
  }) // readFile()…
} // for
console.log('The longest file is '
  +'%s and its length is %d bytes.',
  maxName, maxLength)
```

- Therefore, the main execution thread…
  - Has run all iterations without becoming blocked.
    - Its execution is not interrupted by other threads in its process.
  - And it continues once the "for" loop completes.

1. An erroneous example

```
 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: //
 7:   fs.readFile(args[i],'utf8',
 8:     function(err,data) {
 9: //
10:       if (!err) {
11:         console.log('Processing %s...',
12:           args[i])
13:         if (data.length>maxLength) {
14:           maxLength = data.length
15:           maxName = args[i]
16:         }
17:       } // if (!err)…
18:   }) // readFile()…
19: } // for
20: console.log('The longest file is '
21:   +'%s and its length is %d bytes.',
22:   maxName, maxLength)
```

- Thus, it reaches the instruction that follows the loop (lines 20-22).
  - Because of this, it prints:

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

    - Since no-one has changed the values of **maxName** or **maxLength**.
  - Once this is done, that thread has completed all its instructions. So, it looks for other available "execution turns":
    - There may be some: those corresponding to the prepared *callback* execution contexts.

1. An erroneous example

```
const fs = require('fs')
var args = process.argv.slice(2)
var maxName = 'NONE'
var maxLength = 0
for (var i = 0; i<args.length; i++) {
//
  fs.readFile(args[i],'utf8',
    function(err,data) {
//
      if (!err) {
        console.log('Processing %s...',
          args[i])
        if (data.length>maxLength) {
          maxLength = data.length
          maxName = args[i]
        }
      } // if (!err)…
    }) // readFile()…
} // for
console.log('The longest file is '
  +'%s and its length is %d bytes.',
  maxName, maxLength)
```

- Those callback execution contexts become ready to run as soon as their associated system calls complete.
  - Such completion time depends on each file size.
  - Because of this, their completion order may not be the same than the command-line argument order.

1. An erroneous example

```
const fs = require('fs')
var args = process.argv.slice(2)
var maxName = 'NONE'
var maxLength = 0
for (var i = 0; i<args.length; i++) {
//
  fs.readFile(args[i],'utf8',
    function(err,data) {
//
      if (!err) {
        console.log('Processing %s...',
          args[i])
        if (data.length>maxLength) {
          maxLength = data.length
          maxName = args[i]
        }
      } // if (!err)...
    }) // readFile()...
} // for
console.log('The longest file is '
  +'%s and its length is %d bytes.',
  maxName, maxLength)
```

- Indeed, when those callbacks run, the main thread has already completed all iterations in the "for" loop! (Slide 12)
  - So, what is the value of variable "i" at that time?
    - It is args.length, i.e. 4 in our example.
    - But process.argv[4] is "**undefined**" since it only holds the file names in slots 0 to 3!

1. An erroneous example

And this explains the process output:

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

- Indeed, whe cks run, the main thread has already completed all ite the "for" loop! (Slide 12)
  - So, what is the value c ble "i" at that time?
    - It is args.length, i.e. 4 in ou xample.
    - But process.argv[4] is "**undefined**" since it only holds the file names in slots 0 to 3!

1. An erroneous example

- There are two main problems in this program:
  1. The trace messages that printed the name of the file being processed did not show appropriate file names.
     - They were based on the value of the "i" variable, but that value could not be passed as an argument to the *callback*.
     - Because of this, the file name was wrong.
  2. The final output message reporting the name and size of the longest file is printed too early, when no valid data is ready yet.
     - That message cannot be printed in the code that follows the "for" loop.
       - Since those instructions are executed by the main thread before any *callback* is used.
     - The message should be printed by a sentence placed IN THE *CALLBACK*!!
       - Once all file names have been processed.
       - We need a counter to ensure this.

Index

2. Debugging

- While trying to correct an erroneous program, it is useful to...
  - add tracing messages into it
  - and/or use a debugger
- NodeJS includes a debugger activated through the **inspect** option at program execution
  - node **inspect** *program.js arguments*
- It is part of the NodeJS official documentation
  - https://nodejs.org/api/debugger.html
- Briefly:
  - Really basic
    - Step-by-step execution (**c**ont, **n**ext)
    - Show variables or expressions (**watch**('…'))
    - Breakpoints' setting (**debugger**)
  - It admits remote interaction (*Chrome, VSC, …*)

Debugger listening on ws://127.0.0.1:9229/6a8d75dd-756e-4265-add3-9a3f81596306

2. Debugging: getting ready

- Let's get some internal information in this program…
  - At the beginning of each iteration
  - At the beginning of each callback
- Let's add some breakpoints (***debugger*** sentence) to the initial program. This is the resulting new version:

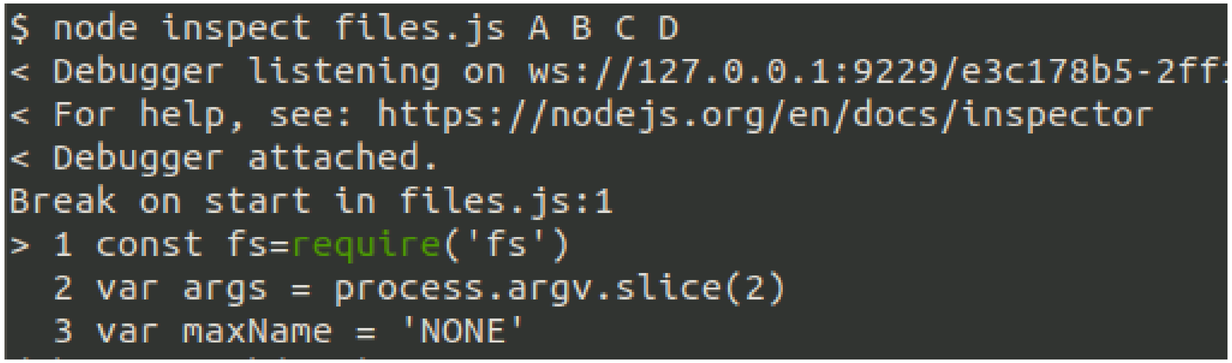
```
const fs = require('fs')
var args = process.argv.slice(2)
var maxName = 'NONE'
var maxLength = 0
for (var i = 0; i<args.length; i++) {
debugger
  fs.readFile(args[i],'utf8',
    function(err,data) {
debugger
      if (!err) {
        console.log('Processing %s...',
          args[i])
        if (data.length>maxLength) {
          maxLength = data.length
          maxName = args[i]
        }
      } // if (!err)…
  }) // readFile()…
} // for
console.log('The longest file is '
  +'%s and its length is %d bytes.',
  maxName, maxLength)
```

2. Debugging: getting ready

- It is useful to think previously:
  - which informations are meaningful
  - which are the expected values at each moment
- In this case we don't need to be selective…

```
const fs = require('fs')
var args = process.argv.slice(2)
var maxName = 'NONE'
var maxLength = 0
for (var i = 0; i<args.length; i++) {
debugger
  fs.readFile(args[i],'utf8',
    function(err,data) {
debugger
      if (!err) {
        console.log('Processing %s...',
          args[i])
        if (data.length>maxLength) {
          maxLength = data.length
          maxName = args[i]
        }
      } // if (!err)…
  }) // readFile()…
} // for
console.log('The longest file is '
  +'%s and its length is %d bytes.',
  maxName, maxLength)
```

2. Debugging: invocation from the command line



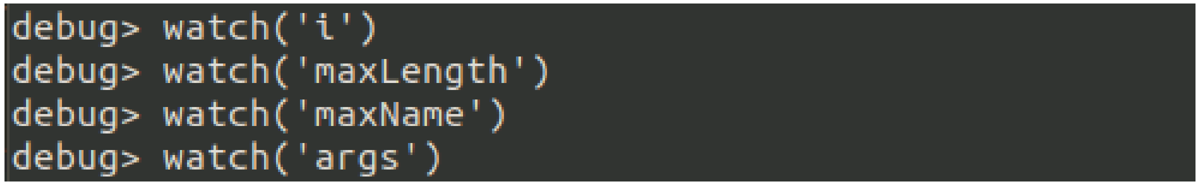
```
$ node inspect files.js A B C D
< Debugger listening on ws://127.0.0.1:9229/e3c178b5-2ff
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in files.js:1
> 1 const fs=require('fs')
  2 var args = process.argv.slice(2)
  3 var maxName = 'NONE'
```

- It will pause at the beginning, waiting for commands

Command	Description
cont (c)	Continue execution
next (n)	Run the next command
step (s)	Step into a function being called
out (o)	Step out of a function and return to the calling command
pause	Pause running code
watch('*expression*')	Show the expression value
.exit	End this debugging session

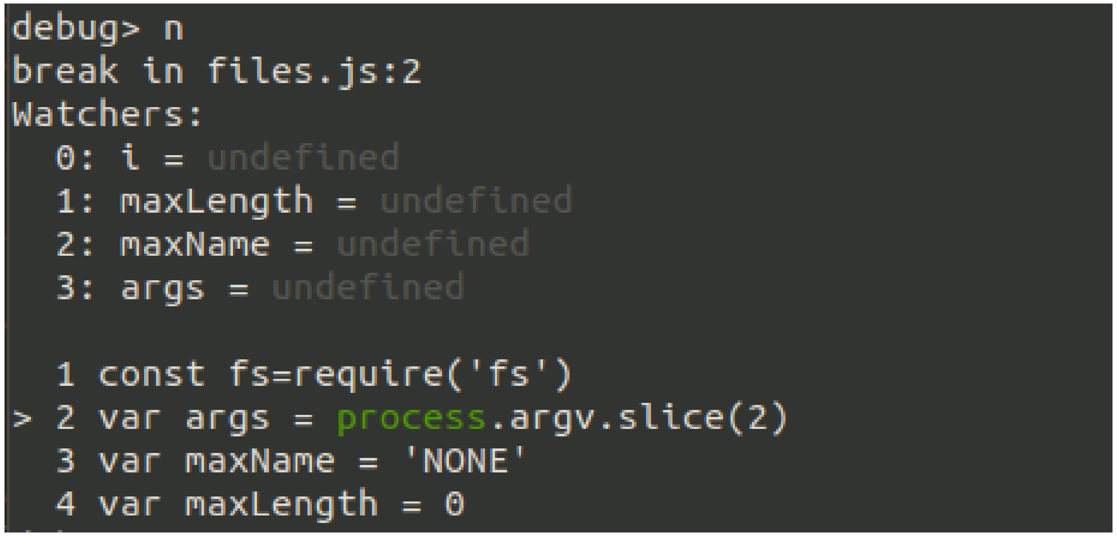
2. Debugging: tracing

- Let's add **watch** expressions for some interesting variables (**i**, **maxLength**, **maxName** and **args**)



```
debug> watch('i')
debug> watch('maxLength')
debug> watch('maxName')
debug> watch('args')
```

- Next instruction is executed (**<u>n</u>ext**)



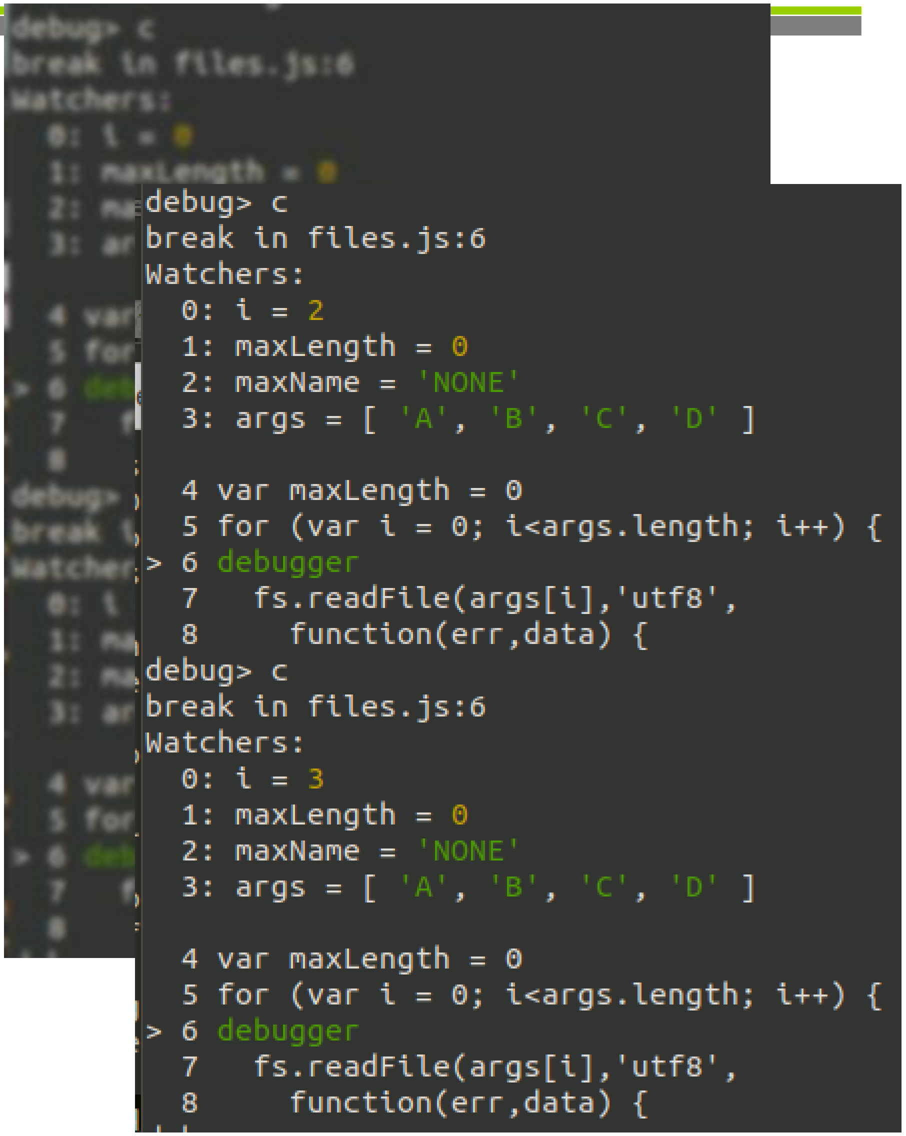
```
debug> n
break in files.js:2
Watchers:
  0: i = undefined
  1: maxLength = undefined
  2: maxName = undefined
  3: args = undefined

  1 const fs=require('fs')
> 2 var args = process.argv.slice(2)
  3 var maxName = 'NONE'
  4 var maxLength = 0
```

It is important to check if observed values do match expected ones

2. Debugging: tracing

- **c** command will show information at next breakpoint
- You must remember that line 9 instruction belongs to a *callback* that it is not effective by the moment
- Illustration at the right shows 2 last iterations (from the total 4)
  - **i** values match what we expected
  - **maxName** and **maxLength** remain unchanged



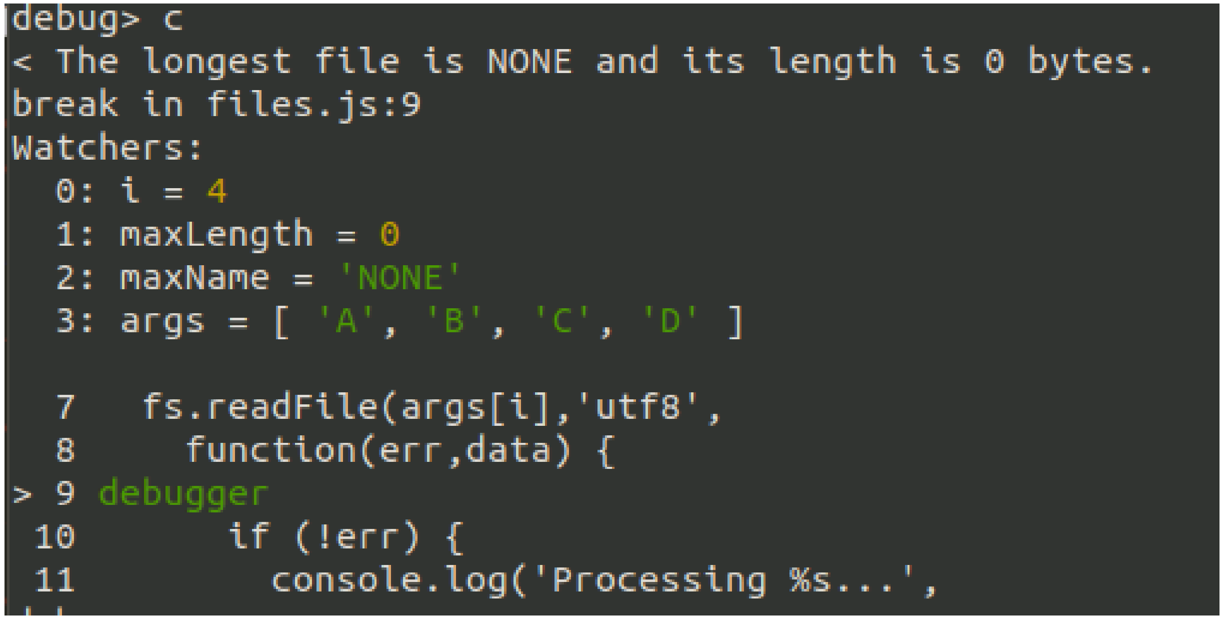
```
debug> c
break in files.js:6
Watchers:
  0: i = 2
  1: maxLength = 0
  2: maxName = 'NONE'
  3: args = [ 'A', 'B', 'C', 'D' ]

  4 var maxLength = 0
  5 for (var i = 0; i<args.length; i++) {
> 6 debugger
  7   fs.readFile(args[i],'utf8',
  8     function(err,data) {
debug> c
break in files.js:6
Watchers:
  0: i = 3
  1: maxLength = 0
  2: maxName = 'NONE'
  3: args = [ 'A', 'B', 'C', 'D' ]

  4 var maxLength = 0
  5 for (var i = 0; i<args.length; i++) {
> 6 debugger
  7   fs.readFile(args[i],'utf8',
  8     function(err,data) {
```

2. Debugging: tracing

- But many events occur after pressing <u>**c**</u> …



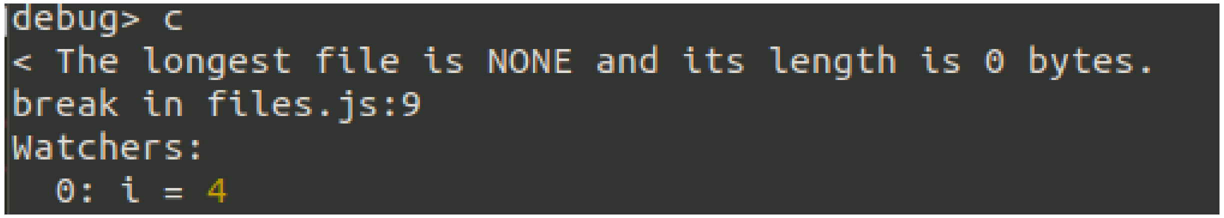
```
debug> c
< The longest file is NONE and its length is 0 bytes.
break in files.js:9
Watchers:
  0: i = 4
  1: maxLength = 0
  2: maxName = 'NONE'
  3: args = [ 'A', 'B', 'C', 'D' ]

  7   fs.readFile(args[i],'utf8',
  8     function(err,data) {
> 9 debugger
 10       if (!err) {
 11         console.log('Processing %s...',
```

… and we need a detailed explanation …

2. Debugging: tracing

- First event…



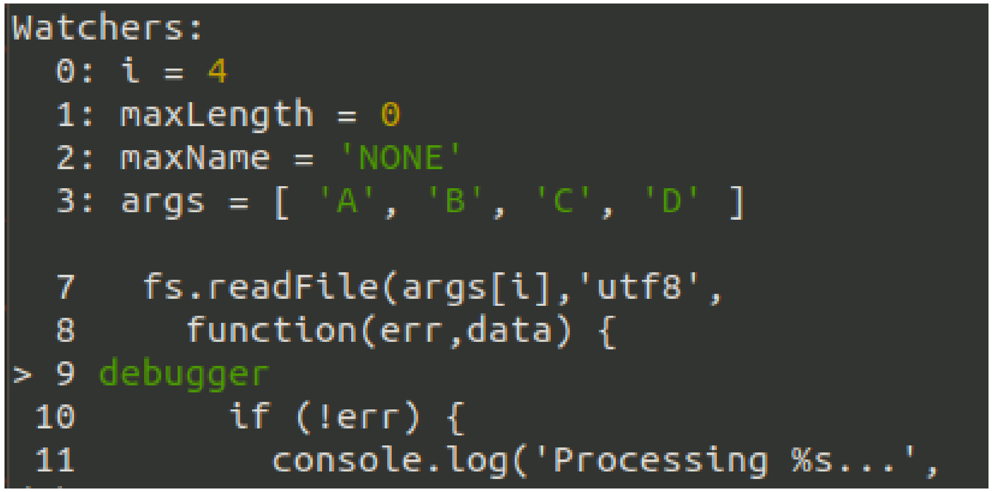
```
debug> c
< The longest file is NONE and its length is 0 bytes.
break in files.js:9
Watchers:
  0: i = 4
```

- Program outputs "**The longest file is NONE and its length is 0 bytes**"
  - **i** stores 4, **for** loop ends, **console.log()** (lines 20 to 22) is executed
  - Result is wrong because neither *callback* has been executed

    (this behaviour has been described at page 12)

2. Debugging: tracing

- This thread ends, but there remains (or not) some more waiting for execution
  - It depends on the termination of **readFile()** instruction execution that started at each loop iteration
  - Debugger does not show the whole **console.log()** instruction:

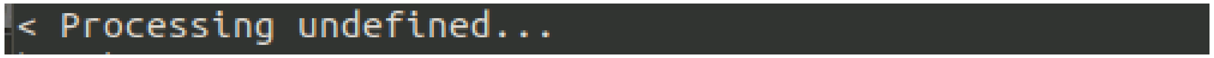


```
Watchers:
  0: i = 4
  1: maxLength = 0
  2: maxName = 'NONE'
  3: args = [ 'A', 'B', 'C', 'D' ]

   fs.readFile(args[i],'utf8',
     function(err,data) {
>  debugger
       if (!err) {
         console.log('Processing %s...',
```

console.log('Processing %s...', args[i])

- but **i** stores **4**!, so **args[4]** is **undefined**

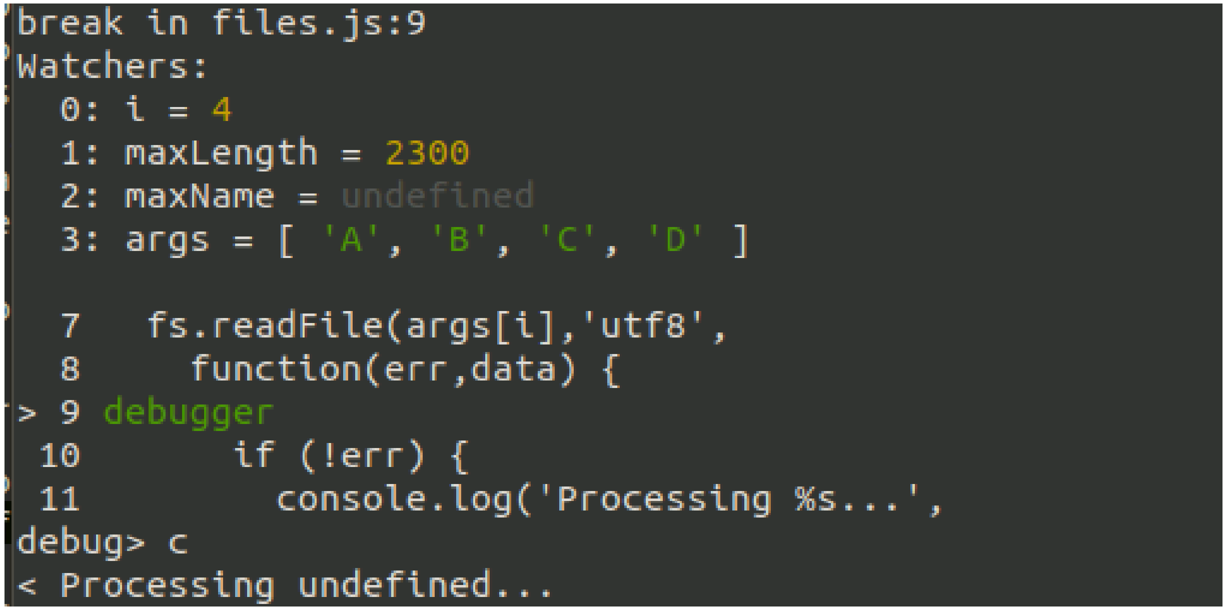


```
< Processing undefined...
```

- It may be confusing watching values *at the beginning* of the callback while reading a **console.log** message executed *at the end* of the same callback
  - Specially if those values (**maxLength**, **maxName**) may be modified through callback code execution

2. Debugging: the end

- This second thread ends, but there are some more waiting their turn as **readFile()**… callbacks become executed…
  - Messages are identical for these 3 cases



```
break in files.js:9
Watchers:
  0: i = 4
  1: maxLength = 2300
  2: maxName = undefined
  3: args = [ 'A', 'B', 'C', 'D' ]

  7   fs.readFile(args[i],'utf8',
  8     function(err,data) {
> 9 debugger
 10       if (!err) {
 11         console.log('Processing %s...',
debug> c
< Processing undefined...
```

- **maxLength** value at line 9 (callback beginning), does not change (**2300**) after first write
  - But its resulting value will actually be **2812500** at callback end
- **maxName** gets **undefined**

Index

3. Its correct counterpart

- Let us remember the problems identified in Section 1:
  - The second problem is easy to solve.
    - Some guidelines have been given in Slide 16.
  - The first problem arises because the *callback* has no access to the value of "i" in that iteration of the loop.
    - *Callbacks* to be used in library functions have a static signature.
      - We cannot add other parameters to them.
      - Therefore, we need a mechanism for passing the appropriate file name to our *callback* code.
        - The solution is based on the variable declaration scope.
          - Recall that a function is able to access every variable or parameter declared in any outer scope.
          - Thus, we should write a function that has the needed file name as one of its formal parameters and that returns as a result the *callback* to be used.
            - In this way, the *callback* code will have access to that file name.
            - This is a CLOSURE.

3. Its correct counterpart

- The resulting program fixes the problems shown before. So, it:
  - prints which is the longest file (and its size)…
  - …from a list of file names received as command-line arguments.
- Its code is:

```
const fs=require('fs')
var args=process.argv.slice(2)
var maxName='NONE'
var maxLength=0
var counter=0
function generator(name) {
  return function(err,data) {
    if (!err) {
      console.log('Processing %s...',
        name)
      if (data.length>maxLength) {
        maxLength=data.length
        maxName=name
      }
    }
    if (++counter==args.length)
      console.log('The longest file is %s '
      +'and its length is %d bytes.',
      maxName, maxLength)
  }
}
for (var i=0; i<args.length; i++)
  fs.readFile(args[i], 'utf8',
    generator(args[i]))
```

3. Its correct counterpart

```
 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]))
```

- Now, the function generator() solves the second problem:
  - It receives the file name in its formal parameter.
  - Then, all its code has access to that name.
  - And it returns a function that has the signature of the readFile() *callback* and that processes the completion of each reading operation.
- Besides, in line 16 it checks whether all files have been managed.
  - If so, it prints the adequate output message and the process ends there.

3. Its correct counterpart

```
 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]))
```

- Note also that in line 24, when the third argument to readFile() is stated…
  - We do not pass a pointer to the "generator" function, but WE CALL IT specifying the appropriate file name as its argument.
  - This returns a function that is interpreted as the intended *callback* to be used once the given file is read.

3. Its correct counterpart

- Let us try this second version of the program, using the same arguments explained in Slide 5:

```
$ node files A B C D
Processing A...
Processing C...
Processing D...
Processing B...
The longest file is B and its length is 2812500 bytes.
$
```

- Let us follow its trace in order to understand its output…

3. Its correct counterpart

```
 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]))
```

- Its first 5 lines do the following:
  1. Import the ‘fs’ module into the fs constant.
  2. Get the command line arguments into the ‘args’ array.
  3. Initialise the name of the longest file to ‘NONE’.
  4. Initialise the length of the longest file to zero.
  5. Initialise the counter of processed file names to zero.

3. Its correct counterpart

```
 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]))
```

- Lines 6 to 21 define the generator() function.
  - But such function is not called yet.
  - On being called, it will return an anonymous function as its result.
    - That function has two formal parameters (err and data) that match the signature of the fs.readFile() *callback*.
- At the moment, the execution proceeds till line 22.

3. Its correct counterpart

```
 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]))
```

- Lines 22 to 24 define a loop with as many iterations as file names. In each iteration:
  - The readFile() function is called. Thus, the process starts to read the corresponding file.
  - The third actual parameter in this call is a call to the generator() function.
    - With it, the file name managed in that iteration is maintained in the 'name' implicit variable held in the resulting *callback* code.
    - Therefore, this solves the first problem mentioned in Slide 16.

3. Its correct counterpart

- Once all iterations have ended, the initial thread has no other instruction to run.
  - Apparently it has completed the entire program.
  - But the running process does not end yet…
    - …since there are four started readFile() calls that haven't been completed!
- Those readFile() calls will eventually terminate.
  - Each time any of those calls finishes, its corresponding *callback* is run.
  - We have called readFile() using this sequence of file names: "A", "B", "C" and "D".
    - But "B" is the longest file and "C" is the shortest one.
      - So the read completion order is unpredictable!
- Let us continue with our trace…

3. Its correct counterpart

```
const fs=require('fs')
var args=process.argv.slice(2)
var maxName='NONE'
var maxLength=0
var counter=0
function generator(name) {
  return function(err,data) {
    if (!err) {
      console.log('Processing %s...',
        name)
      if (data.length>maxLength) {
        maxLength=data.length
        maxName=name
      }
    }
    if (++counter==args.length)
      console.log('The longest file is %s '
      +'and its length is %d bytes.',
      maxName, maxLength)
  }
}
for (var i=0; i<args.length; i++)
  fs.readFile(args[i], 'utf8',
    generator(args[i]))
```

According to the output shown in Slide 34, the first file that completes its readFile() operation is A. Its size is 2300 bytes. The callback prints this line on screen:

Processing A…

…and later it updates the value of maxLength (setting it to 2300) and maxName (“A”). Variable “counter” is also increased.

3. Its correct counterpart

```
const fs=require('fs')
var args=process.argv.slice(2)
var maxName='NONE'
var maxLength=0
var counter=0
function generator(name) {
  return function(err,data) {
    if (!err) {
     console.log('Processing %s...',
       name)
     if (data.length>maxLength) {
        maxLength=data.length
         maxName=name
      }
    }
    if (++counter==args.length)
      console.log('The longest file is %s '
      +'and its length is %d bytes.',
      maxName, maxLength)
  }
}
for (var i=0; i<args.length; i++)
  fs.readFile(args[i], 'utf8',
    generator(args[i]))
```

Subsequently, C completion arises. Its size is 180 bytes.
The callback prints this line on screen:

Processing C…

…but now maxLength and maxName do not need any update.
Variable "counter" is increased. It reaches value 2. No other message is printed.

3. Its correct counterpart

```
const fs=require('fs')
var args=process.argv.slice(2)
var maxName='NONE'
var maxLength=0
var counter=0
function generator(name) {
  return function(err,data) {
    if (!err) {
     console.log('Processing %s...',
       name)
     if (data.length>maxLength) {
        maxLength=data.length
          maxName=name
      }
    }
    if (++counter==args.length)
      console.log('The longest file is %s '
      +'and its length is %d bytes.',
      maxName, maxLength)
  }
}
for (var i=0; i<args.length; i++)
  fs.readFile(args[i], 'utf8',
    generator(args[i]))
```

Later, D completion arises. Its size is 470 bytes.
The callback prints this line on screen:

Processing D…

… maxLength and maxName do not need any update. Variable "counter" is increased. It reaches value 3. No other message is printed.

3. Its correct counterpart

```
const fs=require('fs')
var args=process.argv.slice(2)
var maxName='NONE'
var maxLength=0
var counter=0
function generator(name) {
  return function(err,data) {
    if (!err) {
      console.log('Processing %s...',
        name)
      if (data.length>maxLength) {
        maxLength=data.length
          maxName=name
      }
    }
    if (++counter==args.length)
      console.log('The longest file is %s '
      +'and its length is %d bytes.',
      maxName, maxLength)
  }
}
for (var i=0; i<a                 )
  fs.readFil
    gen
```

Finally, B completes its read operation. Its size is 2812500 bytes. The callback prints this line on screen:

Processing B…

…and maxLength and maxName are updated. Variable "counter" is increased. It reaches value 4. Therefore, the last reporting message is printed, telling the user that the longest file is B. Since there is no other pending file, the process ends here.

Index

4. Another correct version

- The two programs seen up to now have used "**var**" in order to declare variables.
  - The first problem discussed in Section 1 was partially caused by those declarations!
  - A more compact solution to that problem consists in using "**let**" instead of "**var**".
    - "**let**" defines variables in the scope of its current block.
      - A block is a group of statements encompassed by a pair of curly braces: {}
      - Thus, those variables may be accessed in that block and others nested in it.
    - When "**let**" is used in the global scope, those variables are known from that point onwards.
      - They do not define properties of the "**global**" object.
  - Besides, the "**for**" statement defines an implicit block that encompasses all statements contained in the loop.
    - Thus, all instructions in the loop body "remember" which was the current value of the iterator variable used in that "**for**" statement.

4. Another correct version

- So, the following program is also correct and provides exactly the same output than that shown in Section 2:

```
const fs=require('fs')
var args=process.argv.slice(2)
var maxName='NONE'
var maxLength=0
var counter=0
for (let i=0; i<args.length; i++)
  fs.readFile(args[i],'utf8',
    function(err,data) {
      if (!err) {
        console.log('Processing %s...',
            args[i])
        if (data.length>maxLength) {
          maxLength=data.length
          maxName=args[i]
        }
      }
      if (++counter==args.length)
        console.log('The longest file is %s'
          +' and its length is %d bytes.',
          maxName, maxLength)
  })
```

4. Another correct version

Since "i" is now defined using "**let**", its scope is only the set of instructions contained in the body of that "**for**" loop.
Each iteration uses a new definition of "i", with a different value.

```
const fs=
var args=              ice(2)
var max              ONE'
var ma         n=0
var co    ter=0
for (let i=0; i<args.length; i++)
  fs.readFile(args[i],'utf8',
    function(err,data) {
      if (!err) {
        console.log('Processing %s...',
          args[i])
        if (data.length>maxLength) {
          maxLength=data.length
          maxName=args[i]
        }
      }
      if (++counter==args.length)
        console.log('The longest file is %s'
          +' and its length is %d bytes.',
          maxName, maxLength)
})
```

4. Another correct version

Because of this, now the *callback* used in each iteration "remembers" which value of "i" was used there.
So, it prints the correct file name in lines 10 and 11 and assigns it correctly in line 14. Note that this loop has already finished when those *callbacks* run.

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