Unit 2 – JavaScript and NodeJS

Network Information System Technologies

Objectives

- To use NodeJS (JavaScript) as a basic tool for developing distributed system components.
- To identify the main JavaScript/NodeJS characteristics and its advantages for application development: event-driven, asynchronous actions,...
- To describe some of the NodeJS modules to be used in this course.

1. Introduction

- The rest of this presentation introduces…
  - The JavaScript programming language, concretely ECMAScript 6
  - The NodeJS interpreter
- This is not a JavaScript or NodeJS reference. Only some of their aspects (those relevant for this subject) are described.

1. Introduction

- JavaScript is a scripting language, interpreted, dynamic and portable
  - High level of abstraction
    - □ Simple programs
    - □ Fast development
- Programming language initially designed for providing dynamic behaviour to web pages
  - Current browsers include an interpreter of this language
- Event-driven with asynchronous interactions supported with "callbacks"
  - This boosts both throughput and scalability
- No support for multi-threading
  - No shared objects. No need for synchronisation mechanisms
  - But we should take care about when a variable gets its value
    - □ Callback management
- It supports both functional and object-oriented programming

1. Introduction

- NodeJS:
  - Development platform based on the JavaScript interpreter (known as V8) being used by Google in its Chrome browser
    - Node.js provides a series of modules that facilitates the development of distributed applications
  - It defines:
    - Programming interfaces
    - Common utilities
    - Interpreter
    - Module management
    - …
  - Most technologies being considered to set the learning results and competences of NIST can be easily integrated or developed using NodeJS

2. JavaScript. Full possible contents

- Main characteristics
- Code execution alternatives
- Syntax
- Values
  - Primitive
  - Compound (objects)
- Variables
  - Dynamic typing
  - Properties and methods
  - Scope
- Operators
- Statements
- Functions
  - Arity
- Arrays
- Functional programming
- Object orientation
- JSON
- Events
- Callbacks
  - Asynchrony with callbacks
  - Asynchrony with promises

Take a look at the guide for any missing parts!

2. JavaScript. Contents

1. Main characteristics
2. Code execution alternatives
3. Variables
   1. Dynamic typing
   2. Scope
4. Functions
   1. Arity
   2. Functions and scope for variables
   3. Closures
5. Events
6. Callbacks
   1. Asynchrony with callbacks. Limitations
   2. Asynchrony using promises

2.1 Main characteristics of JavaScript

- Imperative and structured
  - Syntax similar to that of Java.
- Multi-paradigm
  - Functional programming:
    - □ Functions are “objects” and can be used as arguments to other functions.
  - Object-oriented programming:
    - □ Based on prototypes, instead of regular classes and inheritance.
    - □ However, prototypes may emulate object orientation.
- Related programming languages

Java	syntax, primitive values vs. objects
Scheme	functional programming
Self	prototype-based inheritance
Perl and Python	string, array and regular expressions

2.2 Code execution alternatives

- How to run its programs? Two basic alternatives:
  1. Using the interpreter included in web browsers.
     - Writing "script" elements in the HTML of a web page:
       - □ <script type='text/javascript'> ... code ... </script>
       - □ <script type='text/javascript' src='file.js'></script>
     - Or using the JavaScript console in your browser.
       - □ Example: Chrome → Tools → JavaScript console

```
> console.log ("Hello")
Hello     → output
undefined → result
```

  2. Using an external interpreter
     - For instance, "node"
       - □ This is the approach to be used in this course.
       - □ The interpreter can be downloaded and installed from http://nodejs.org
         - □ **node** is the command that runs this interpreter

2.3.1 Variables. Dynamic types

- JavaScript is not a "strongly typed" language
  - A variable is declared (with "**let**" or "**var**") before its first use, but without any specification of type → Prone to errors
  - A variable may hold, in an execution, elements of different types (i.e., its type is "dynamic").
    - **let** x=4 // **x** is now a number…
    - x='Text' //…, later a String…
    - x= {colour:'red', brand:'Seat', model:'Toledo', year:2016} // …now, an object…
    - x = [1,2,3,'test',6] // …, here, an array… → Array admits diff. types of values at the same time
    - x = function() {return 'Example'} //…at this point, a function…
    - let y = x() // What is held in **y**? the result of x( )

> x=23

> x = "example"

No Problem

> function y( ) {return 1{

- JavaScript type management is weak. We should take care of its implicit type conversions:
  - **let** x = "7" // Value of x is "7" (a String)
  - x == 7 // **true** (implicit type conversion) → weak typing => similar value
  - x === 7 // **false** (strict comparison)
  - x + 23 // Its result is "723" (+ concatenates strings)
  - x + "2" // Its result is "72"
  - x * 2 // Its result is 14 since operator * has no meaning for strings

2.3.2 Variables. Scope

let x = 5 ✓
} let x = 3 ✓
{ let x = 2 ⊗

! We cannot define a variable multiple times in the same scope (with let)

- Lexic scope
  - The scope of a variable is…
    - Local to the block where it has been declared (using **let**) ≈ similar to Java
    - Local to the function where it has been declared (using **var**) → outside a f(x), "var" has global scope → even if we define inside block, it still exists outside it
    - Global (entire file) when…
      - □ It is not declared inside a function
        - □ Equivalent to assume an implicit global function that holds the entire file
      - □ Or when its is declared in a function without using **let** or **var**
        - □ Example: x = 3. **Not recommended!!**

! Close brackets or it won't work

! You cannot declare in the same scope the same variable with let & var

USE ALWAYS WITH LET OR VAR

- A statement…
  - may access all variables that have been defined in the scopes that include that statement
  - variables are searched from the inner to the outer scope

2.4 Functions

- Anonymous functions
  - function (args) {…}
    - Alternative syntax: (args) => …
  - It is a value that can be assigned, passed as an argument,…
    - Example: const double = function (x) {return 2*x}
      - □ Alternative: const double = (x) => 2*x
  - To be invoked as identifier(args), returning a single value
    - Example: let x = double(28)

! If an argument is not received ⇒ NaN, but no exception is raised

function example(a, b, c) } return a + b + c {
example () → NaN
example (1, 2, 3) → 6
example (1, 2, 3, ~~4, 5, 6~~) → 6
IGNORED

- Declaration
  - function name(args) {...}
  - Equivalent to: const name = function (args) {…}
    - ← function double(x) {return 2*x} ...is equivalent to... →
    - ANONYMOUS ver. ← const double = function (x) {return 2*x}
- They can be declared everywhere, even inside another function (i.e., they can be nested)
- They provide the scope for variable definition
  - When variables are defined using **var**
- Arguments are passed by value (as in Java)
  - But objects are actually passed by reference
- Functions are objects
  - with properties and methods
- A single return value, but it may be a composed element (i.e., an object)

2.4.1 Functions. Arity

- Arity (number of arguments)
  - A function with n arguments may be invoked using...
    - Exactly n values
    - Less than n values. The remaining arguments receive the "undefined" value
    - More than n values. The unexpected arguments are ignored
  - Arguments are accessed...
    - by name
    - or with the "arguments" pseudo-array
  - The arity may be enforced
    - function f(x,y) {if (arguments.length !=
  - Or default values may be assigned

```
function greetings() {
   for (let i=0; i<arguments.length; i++) {
     console.log("Hi, " + arguments[i])
   }
}
greetings("Tom", "John", "Mary")
```

```
Hi, Tom
Hi, John
Hi, Mary
```

DEFAULT VALUES in case

```
function greetings(x = "Anne", y = "John") {
   console.log("Hello, " + x); console.log("Hello, " + y)
}
greetings("Mary"); console.log("\n-----")
greetings(undefined, "George", "Joseph")
```

What...? OK DISCARDED

```
Hello, Mary
Hello, John
-----
Hello, Anne
Hello, George
```

- There cannot be two functions with the same name, even when they are defined with different arities

→ NO OVERLOADING

2.4.2 Functions. Scope

- Example adapted from https://www.evl.uic.edu/luc/bvis546/Essential_Javascript_--_A_Javascript_Tutorial.pdf in order to show different variable scopes:
  - Read [1] in order to get more information about the scope in JavaScript.

```
function alert(x) {  // Needed in Node.js in order
   console.log(x);    // to print messages to stdout.
}

let global = 'this is global';

function scopeFunction() {
~~let~~ ~~var~~ → 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 var
                      // keyword so this is global!'
  alert(isPrivate); // This 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.
```

2.4.3 Functions. Closures

- Closure = function + connection to variables in outer scopes
  - Functions remember the scope where they have been created

```
function createFunc() {
  let name= "Mozilla"
  return function() {console.log(name)}
}
let myFunc = createFunc()
myFunc() // it shows  "Mozilla"
```

It is remembered

  - Another example

generator
f(x)

```
function multiplyBy(x) {
  return function(y) {return x*y}
}
let triplicate = multiplyBy(3)
y = triplicate(21) // Returns 63
```

- Additional details in [1]

2.4.3 Functions. Scope. Closures

```
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))
```

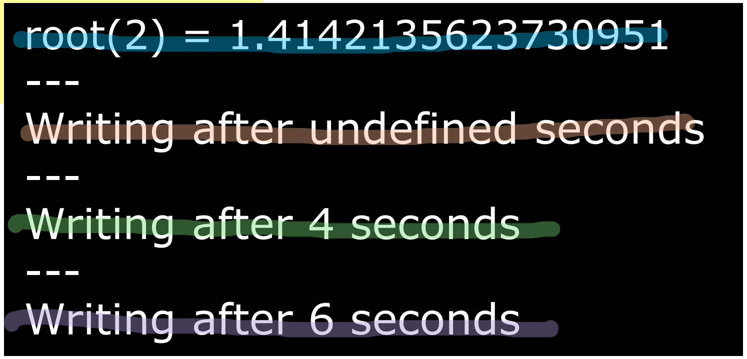
returns undefined

set Timeout (writing (3), 3000)

FAILS

set Timeout (writing, 3000, 3)

WORKS



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

! MUST BE A POINTER TO A function

set Timeout (func, delay , arg 1, arg2 ...)

ms

optional

Places func at time delay in the current event queue

There is NO GUARANTEE that at delay ms the program will run → The main thread still has events

2.5 Events

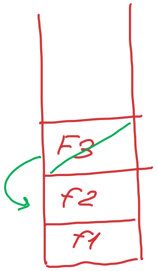
- JavaScript is single-threaded
  - But multiple activities may be executed
  - Setting them as events
- There is an event queue that…
  - accepts external interactions
  - holds pending activities
  - is turn-based

FIFO QUEUE

- Each kind of event may be managed in a different way
  - But all event answers are executed by the same thread
  - This imposes a sequence-based management
    - i.e., a new event isn't processed until the current one is finished

2.5 JS. Events

- JavaScript execution management
  - REGULAR Call stack → we keep the context of each f(x) is stored in a stack
    - Each call to a function pushes an execution context (arguments, local variables...) to this stack.
    - When that function execution ends, its context is popped.
    - If this stack becomes empty, the event queue is scanned.
  - Event queue
    - It queues an execution context per received event.
    - They are executed in sequential order, when the call stack becomes empty.

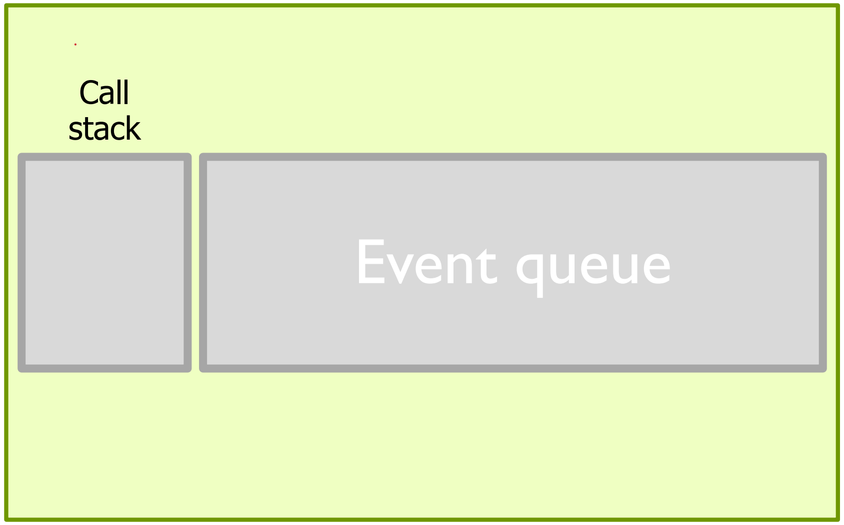


CALL STACK
⇣
Handle the execution
of the main thread

2.5 JS. Events

After processing

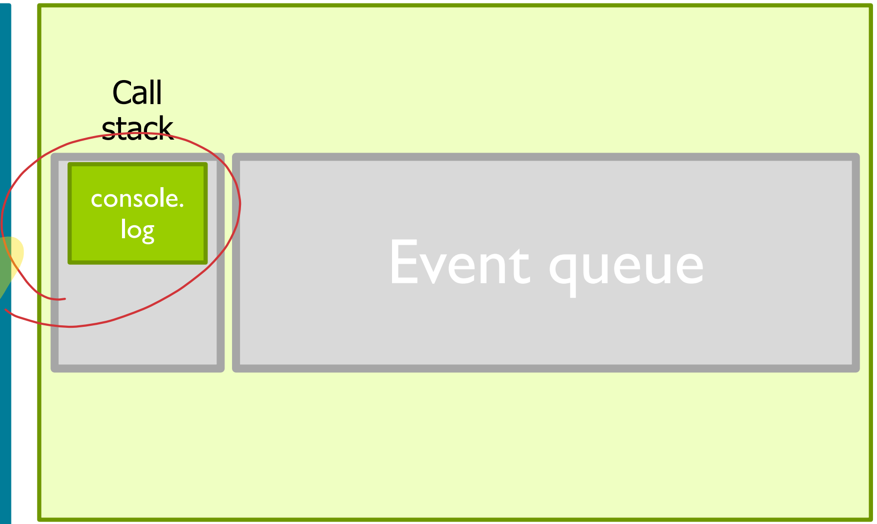
```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
 }

 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
```

2.5 JS. Events

```
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)
```

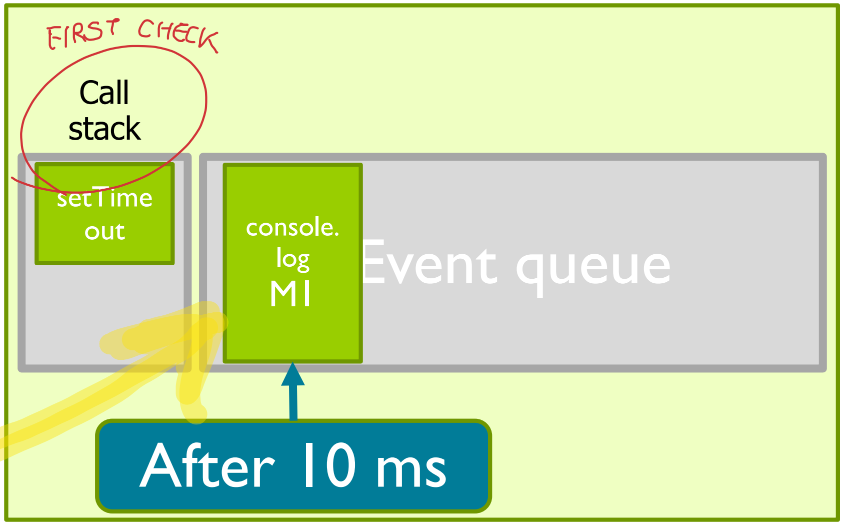


```
C:\> node turnQueue.js
Starting the process...
```

2.5 JS. Events

```
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)
```

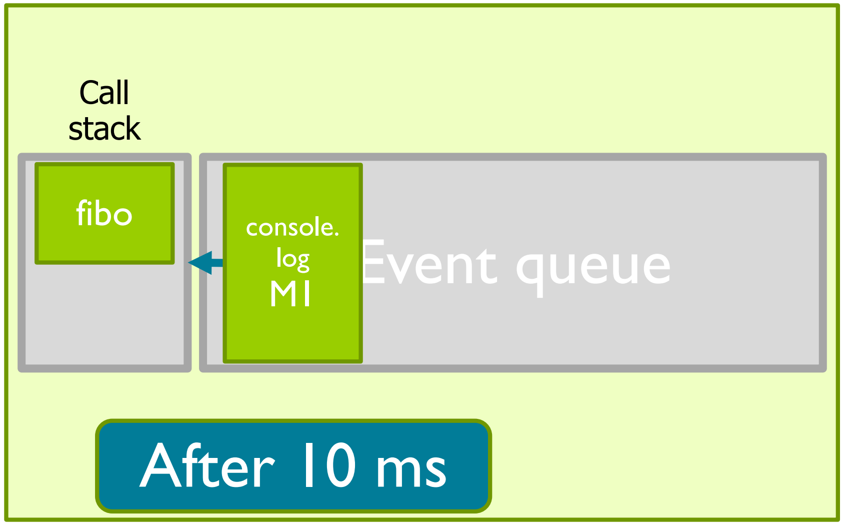
Even with delay = 0, ALWAYS PLACED AT THE END OF EVENT Queue



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

2.5 JS. Events

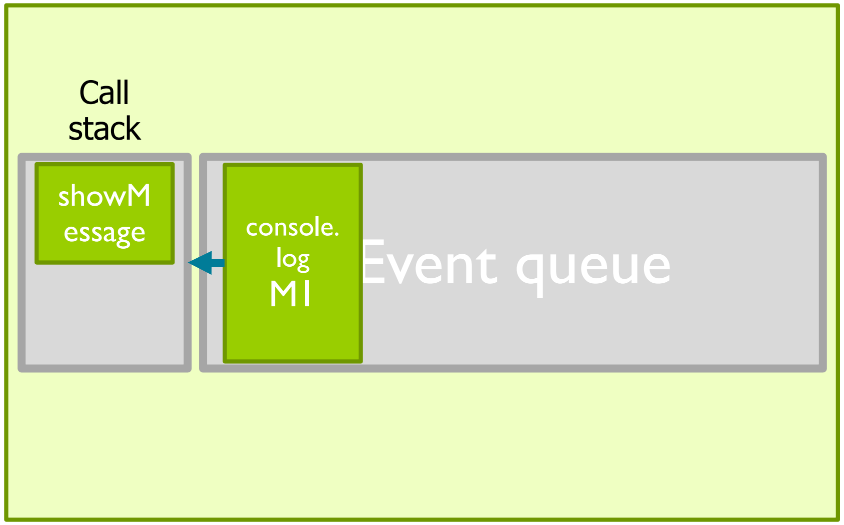
```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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)
```

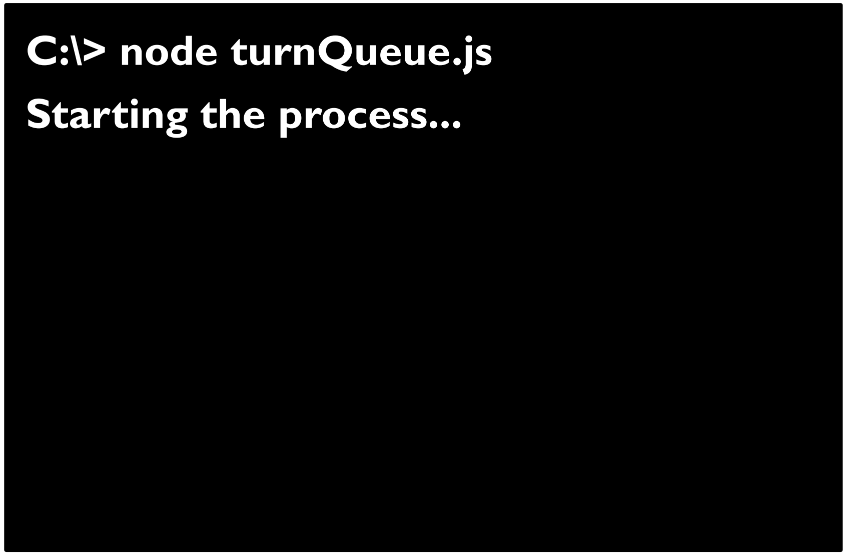


```
C:\> node turnQueue.js
Starting the process...
```

2.5 JS. Events

```
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)
```

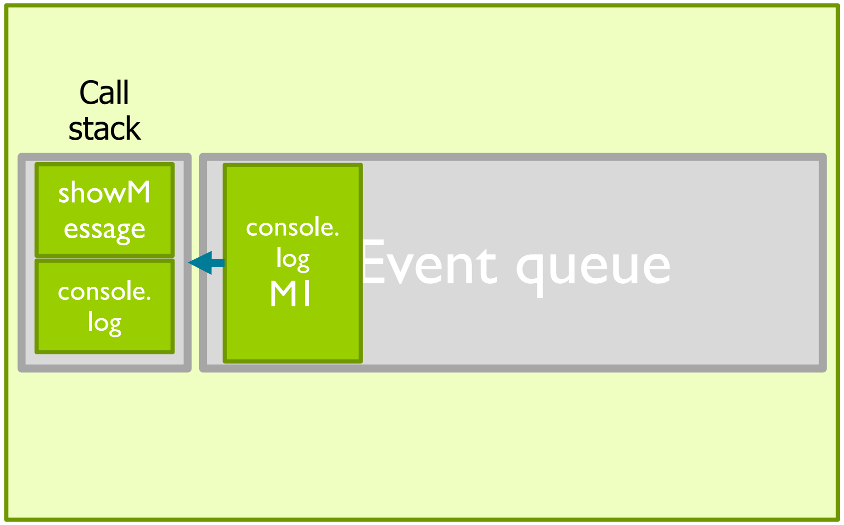




```
C:\> node turnQueue.js
Starting the process...
```

2.5 JS. Events

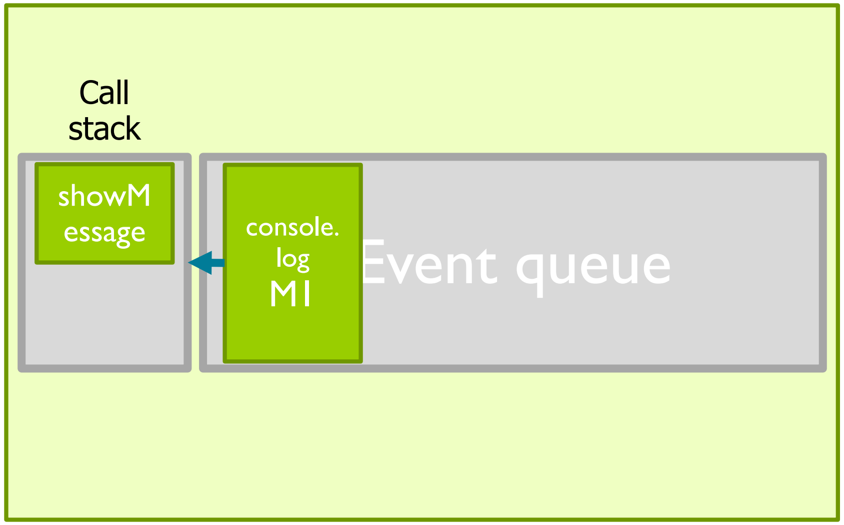
```
function fibo(n) {
    return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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. Events

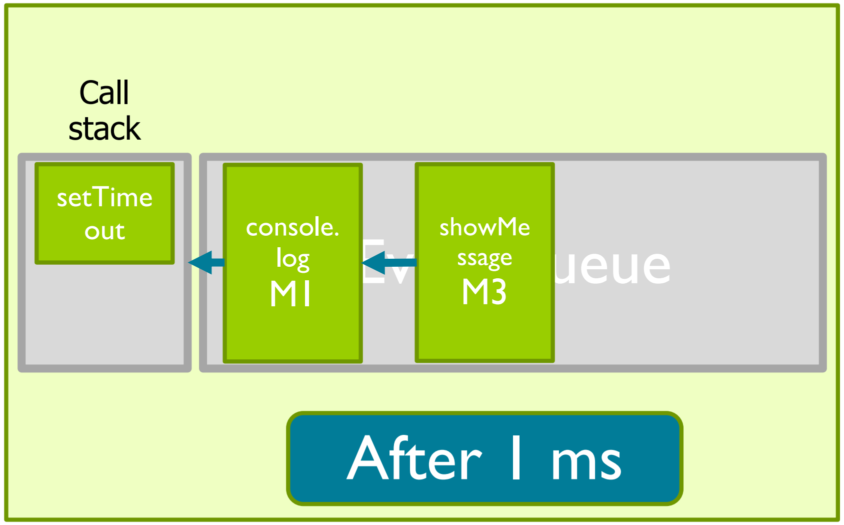
```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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. Events

```
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)
```

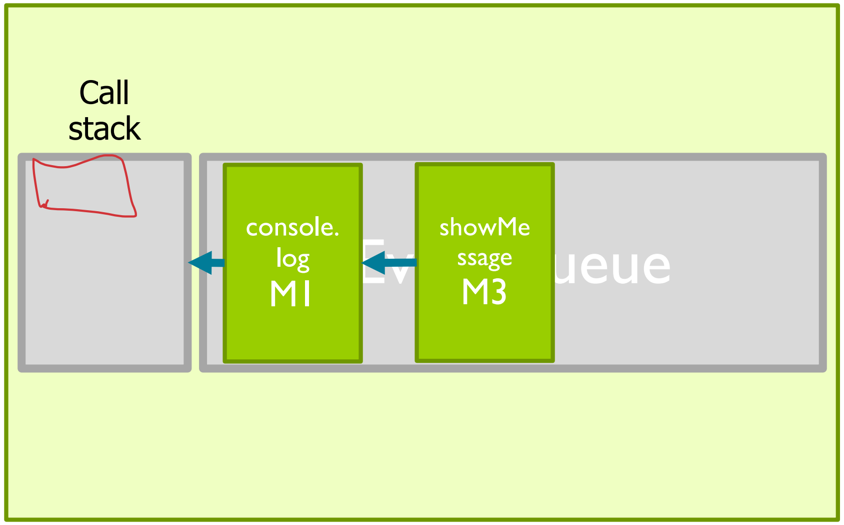


```
C:\> node turnQueue.js
Starting the process...
M2: The result is: 165580141
```

2.5 JS. Events

! We first process the hole program, putting all TimeOut events in the EventQueue, and only when we have processed the program we check & execute the EventQueue

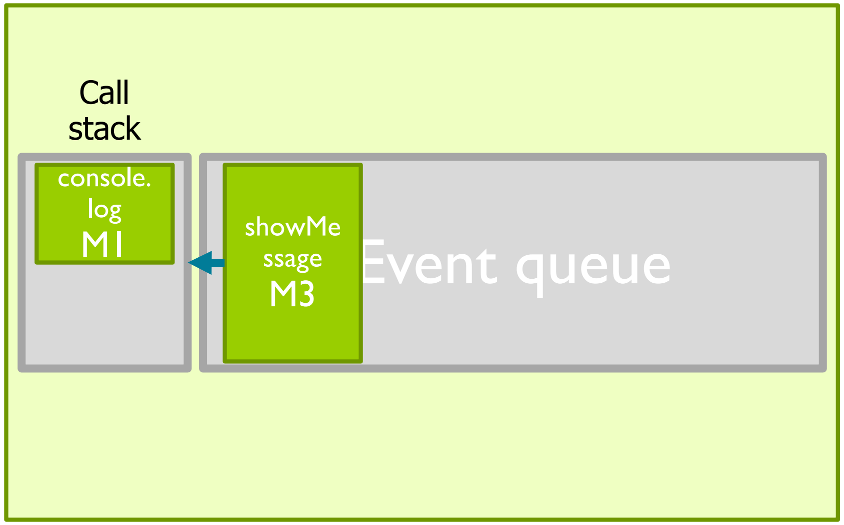
```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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. Events

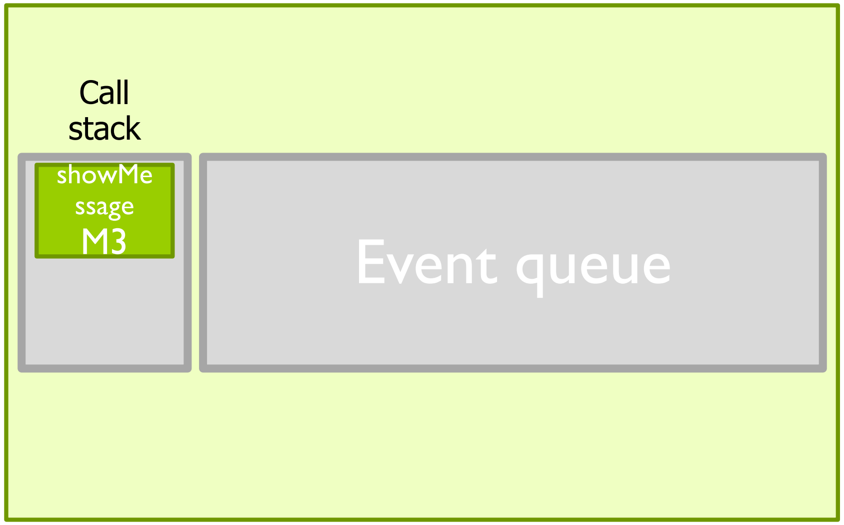
```
function fibo(n) {
    return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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. Events

```
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)
```



Call stack

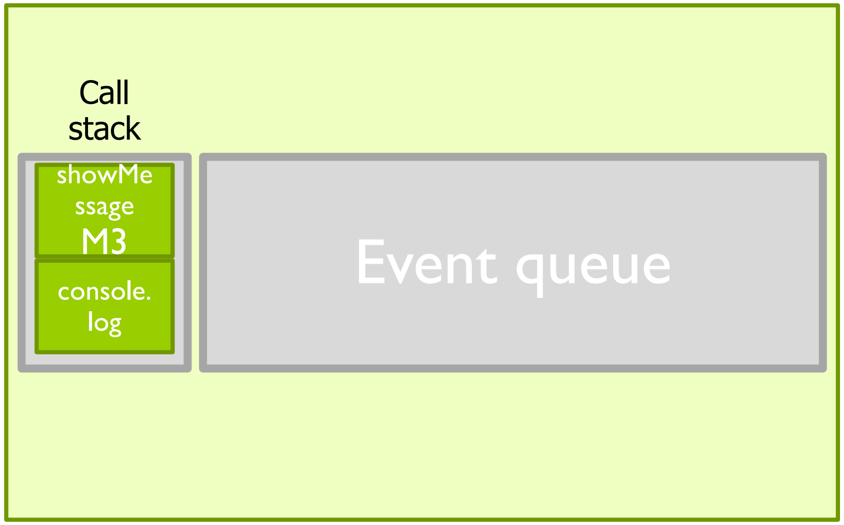
showMessage M3

Event queue

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

2.5 JS. Events

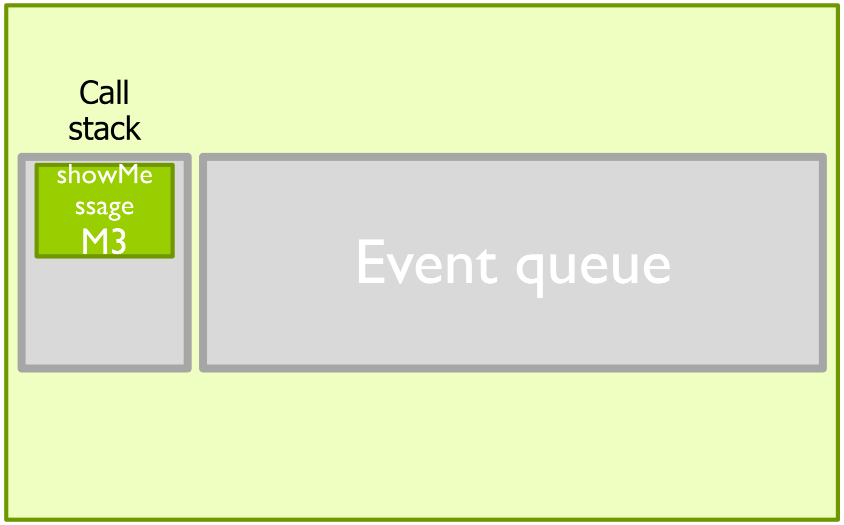
```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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...
M3: The result is: 165580141
```

2.5 JS. Events

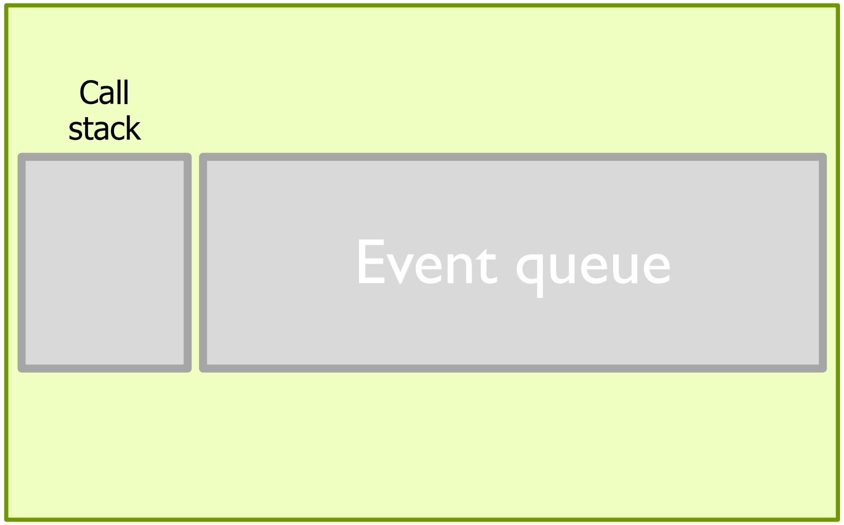
```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

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...
M3: The result is: 165580141
```

2.5 JS. Events

```
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...
M3: The result is: 165580141
C:\>
```

2.6 Callbacks

- A "callback function" is….: → GOAL : Interpret the results
  - …a reference to a function that is passed as an argument to another function B. B invokes that callback when it is terminating its execution.
  - Example: Let us assume a fadeIn() method that progressively vanishes an element that is displayed.
    - □ It is called as: element.fadeIn(speed, function() {…})
    - □ The second argument is a callback function that will be invoked when "element" has completely disappeared.
  - Example 2: Function writingClosure(4) generates the callback for setTimeout in:
    - □ setTimeout(writingClosure(4), 4000)
- Callback functions allow asynchronous invocations:
  - An agent calls B(args,C), being C a callback
  - When B is terminated, it calls C
    - □ Thus, B reports its completion and provides its result

2.6 Callbacks

! when we execute a function, by default it returns "undefined", which can cause problems

```
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))
```

ASYNCHRONOUS

fs.readFile(..) → auxilary thread:
- opens file
- read contents
- completes the reading
- closes the file

KEEP execution

```
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
```

- The arguments for **callback** will be provided by the invoked function (**readFile** in this example)
  - Check the Node.js documentation: File System -> readFile

2.6.1 Callbacks. Limitations

- Callback nesting is not restricted. However, there are practical limits:
  - Exceptions in nested callbacks. If an exception is not catched, it is propagated to the caller.
  - If we do not guarantee a uniform management in all operations, some exceptions may be lost or managed in unexpected operations.

```
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 Callbacks. Limitations

- Other problems
  - The code is hard to read. The execution order is not intuitive.
  - Uncertainty on the turn in which the callback will be run. We cannot rely on its execution in a concrete turn.

```
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')
          }
       })
    })
})
```

Different executions on the same folder may show different orders in their lists of files.

2.6.2 Asynchronous execution. Promises

- Asynchronous executions may be also built using promises
  - Operation calls follow the traditional format (easy to read)
    - There is no callback argument
  - The result of that call is a "promise" object.
    - It represents a future value on which we may associate operations and manage errors
    - It may be in one of the following states
      - □ pending. Initial state. The operation has not yet concluded (unknown result).
      - □ resolved. The operation has terminated and we can get its result. This is a final state that cannot change.
        - □ rejected. The operation has terminated with error. The reason is given.
        - □ fulfilled. The operation has terminated successfully. A value is returned.
  - A function is associated to each final state (rejected vs fulfilled). Such function is run when the main thread finishes its current turn.
    - Actually, it is enqueued in a new turn as a future event.

2.6.2 Asynchronous execution. Promises

- **Creation:** With the Promise() constructor.
  - Promise((resolutionFunc, rejectionFunc) => {...})
    - resolutionFunc(param)
      - □ *Callback* to be used when the promise is successful. With one formal parameter.
    - rejectionFunc(param)
      - □ *Callback* to be used in rejected promises. One formal parameter.

```
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)
```

Asynchronous function to be converted into a promise.

In case of error, the promise is rejected.

If no error arises, the promise is resolved.

2.6.2 Asynchronous execution. Promises

- **Management:** Methods then() and catch()
  - then(onFulfilled [, onRejected]):
    - Sets promise resolution management, using these callbacks:
      - □ onFulfilled, to be called when the promise is fulfilled.
      - □ onRejected, to be called when the promise is rejected.
  - catch(onRejected):
    - onRejected: callback to be invoked when the promise is rejected.
  - Both then() and catch() return a promise.
    - Thus, it is easy to "chain" different management stages.
      - □ A single catch() at the end of that chain manages all promise rejections.

2.6.2 Asynchronous execution. Promises

- **Management of multiple promises:** Methods all() and any()
  - all(promiseArray):
    - Returns a single promise that...
      - □ ...is fulfilled if all promises in promiseArray have been fulfilled.
      - □ ...is rejected as soon as one of the promises in promiseArray becomes rejected.
  - any(promiseArray):
    - Returns a single promise that...
      - □ ...is fulfilled as soon as one of the promises in promiseArray is fulfilled.
      - □ ...is rejected if all promises in promiseArray have been rejected.

2.6.2 Asynchronous execution. Promises

```
// Receive a set of file names, show the length of each one of them, and the global length.
const  fsp = require("fs").promises
let names=process.argv.slice(2) // Array of file names.
let allLength=0 // Accumulated length

// This generator provides the file name to the callback.
// The callback, given the file contents, computes the file length and shows it.
// It also adds that length to the global length.
function genCallback(n) {
                return (data) =>
                                {allLength+=data.length; console.log( n +": " + data.length)}
}
// Generate the promises...
// Next uncommented line is equivalent to this fragment:
/*
let myFiles
for(let i in names)
   myFiles[i] = fsp.readFile(names[i],"utf8").then( genCallback(names[i]) )
*/
let myFiles = names.map((name)=>fsp.readFile(name,"utf8").then(genCallback(name)))

// Use all() in order to report the global length once all files have been read.
Promise.all(myFiles).then(()=>console.log("Total length: "+allLength))
                                                .catch(()=>console.error("Error reading some files!"))
```

2.6.2 Asynchronous execution. Promises

- **async/await**
  - **Goal:** Control when each promise resolution should be managed.
  - **async:** This keyword precedes a function declaration and states that the function returns a promise.
    - If it returns a value, that value is converted into a fulfilled promise.
  - **await:** This keyword precedes a function call that returns a promise and **waits** for the promise resolution.
    - On promise fulfillment, its result is returned.
    - On promise rejection, an exception is thrown, to be managed with:
      - □ **try {...} catch**(ex) **{...}**
      - □ The **catch()** method.
    - **await** must be used inside **async** functions.

2.6.2 Asynchronous execution. Promises

```
// Variation of program in slide 38...
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 Asynchronous execution. Promises

```
// 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 Callbacks vs promises

- Example: Asynchronous read of a file
  - The version based on promises needs that an asynchronous function (in this case, fsp.readFile) returns a promise.

Callbacks

```
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)
         }
      }
})
```

Promises

```
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)
})
```

3.1 NodeJS. Introduction

- NodeJS is a special JavaScript interpreter:
  - Independent. Valid for writing server agents.
    - Built on the basis of the V8 engine (JavaScript in Google Chrome).
    - Not embedded in a web browser.
  - Designed for agile development of scalable applications
    - Event-based I/O model
      - □ Event loop executed by a single thread
      - □ Activity never blocks
    - Concurrency model based on events and callbacks

3.1 NodeJS. Introduction

- Useful for:
  - Development of server components.
  - Non-critical applications.
  - Applications with light REST/JSON interfaces.
  - Single-page applications (that use AJAX to interact with servers)
  - Streaming data.
- References:
  - Interpreter: http://nodejs.org/download/
  - Documentation: http://nodejs.org/api/
  - GitHub repository: https://github.com/nodejs/
  - Conferences: http://www.nodeconf.com/ and http://jsconf.com

3.1 NodeJS. Introduction

- Modularity
  - With "require()", other modules can be included in a program.
- Methods in NodeJS modules are usually asynchronous
  - Method retorns control immediately, and the result is obtained with a callback.
  - A single thread of execution...
    - With no shared objects or critical sections
      - □ All dangers of traditional concurrent program disappear
      - □ But in order to avoid inconsistencies, we should consider in which turn each callback will run.
    - That never blocks
      - □ Since I/O operations and other system services are asynchronous
  - But most module methods have also a synchronous version (without callbacks)
    - E.g., fs.readFile() is asynchronous, but there exists an fs.readFileSync() variant.

3.2 Asynchrony

- How is this asynchrony achieved??
  - Programmers see a single thread, but…
    - An event queue is handled by the node runtime.
      - □ At each time, the Node runtime dequeues the first event and executes it.
      - □ This action defines a "turn".
    - NOTE: setTimeout(f,0) stores function f() in the queue.
      - □ Useful when we need to execute f() once the current activity was finished.
  - Asynchronous modules are based on the **libuv** [7] library.
    - **libuv** maintains a "*thread pool*".

3.2 Asynchrony

- When a blocking operation is called…
  1. A thread T is taken from the "*pool*".
  2. Invocation arguments are given to T, including the "callback" scope.
  3. The invoking thread returns and our program goes on.
     - T remains in the "ready-to-run" state.
  4. T executes all operation sentences.
     - It might block in some of them.
  5. When T finishes that operation, it calls its associated *"callback"*…
     1. T creates a scope for such "callback", passing the needed arguments.
     2. T stores such scope in the event queue.
     3. T comes back to the "*pool*".
     4. The "*callback*" is executed in a future turn.
        - □ When it becomes the first in the event queue.
        - □ This avoids race conditions.

3.3. Module management

- Exports
  - Programmers should decide which objects and method are exported by a module.
  - Each of those elements should be declared as a property of the “module.exports” object (or, simply, “exports”).
    - Example:

```
// Module Circle.js

exports.area = function(r) {
  return Math.PI * r * r;
}
```

```
exports.circumference = function(r) {
  return 2 * Math.PI * r;
}
```

- Require
  - Modules are imported using “require()”.
  - The module global object may be assigned to a variable. This names its context/scope.
    - Example 1: **const** HTTP = require(‘http’);
    - Example 2: **const** st = require(‘./Circle.js’);
      console.log(“Area of a circle with radius 5:“ + st.area(5));

3.4. Events module

- The **events** module is needed for implementing event generators.
  - Generators should be instances of EventEmitter.
  - A generator throws events using its method **emit(event [,arg1][,arg2][...])**.
    - emit() executes the event handlers in the current turn.
    - If we do not want such behavior...
      - □ setTimeout(function() {emit(event,...);},0)
- Event "*listeners*" may be registered in the event emitters:
  - Using method **on(event,listener)** from the emitter.
    - **addListener(event, listener)** does the same.
    - The "*listener*" is a "*callback*".
  - The "*listener*" is invoked each time the event is thrown.
  - There may be multiple "*listeners*" for the same event.
- Documentation available in:
  - http://nodejs.org/api/events.html

3.4. Events module

```
const ev = require('events')
const emitter = new ev.EventEmitter()                 // DON'T FORGET NEW OPERATOR!!
const e1 = "print", e2 = "read"                       // Names of the events.

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. Stream module

- Stream objects are needed to access data streams.
- Four variants:
  - Readable: read-only.
  - Writable: write-only.
  - Duplex: allow both read and write actions.
  - Transform: similar to Duplex, but its writes usually depend on its reads.
- All they are EventEmitter. Managed events:
  - Readable: readable, data, end, close, error.
  - Writable: drain, finish, pipe, unpipe.
- Examples:
  - Readable: process.stdin, files, HTTP requests (server), HTTP responses (client), ...
  - Writable: process.stdout, process.stderr, files, HTTP requests (client), HTTP responses (server),...
  - Duplex: TCP sockets, files, ...
- Documentation available in:
  - http://nodejs.org/api/stream.html

3.5. Stream module. Example

- Interactive version of the computation of the circumference given a radius.
- process.stdin is a "Readable" *stream*.

```
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 instead of "Buffers".
// 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 that has been read is "str".  Remove its trailing endline.
  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. Net module

- “net” module: management of TCP sockets:
  - net.Server: TCP server.
    - Generated using **net.createServer([options,][connectionListener])**.
      - □ “connectionListener”, when used, has a single parameter: a TCP socket already connected.
    - Events that may manage: listening, connection, close, error.
  - net.Socket: Socket TCP.
    - Generated using “new net.Socket()” or “net.connect(options [,listener])” or “net.connect(port [,host][,listener])”
    - Implements a Duplex Stream.
    - Events that may manage: connect, data, end, timeout, drain, error, close.
- Documentation available in:
  - http://nodejs.org/api/net.html

3.6. Net module

- Example (from the NodeJS documentation):

Server

```
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');
  });
```

Client

```
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 Net module. Example 2

Server

```
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')
})
```

Client

```
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 Net module. Example 3

Server

```
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)
```

Client

```
const net = require('net’)
if (process.argv.length != 4) {…}
let fun = process.argv[2]
let num = Math.abs(parseInt(process.argv[3]))

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. HTTP Module

- To implement web servers (and also their clients).
- Consists of the following classes:
  - http.Server: EventEmitter that models a web server.
  - http.ClientRequest: HTTP request.
    - It is a Writable Stream and an EventEmitter.
    - Events: response, socket, connect, upgrade, continue.
  - http.ServerResponse: HTTP response.
    - It is a Writable Stream and an EventEmitter.
    - Events: close.
  - http.IncomingMessage: It implements the requests (for the web server) and the responses associated to ClientRequests.
    - It is a Readable Stream.
    - Events: close.
- Documentation available in:
  - http://nodejs.org/api/http.html

3.7. HTTP Module

- A minimal web server: Given as an example in: 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. Learning Results

- When this seminar is concluded, the student should be able to:
  - Identify JavaScript (with NodeJS) as an example of programming language that admits asynchronous programming.
  - Identify JavaScript as a programming language that avoids multiple concurrency problems/dangers.
  - Build small programs in NodeJS using an event-driven paradigm.
  - Know multiple sources in order to delve into NodeJS and JavaScript programming.

5. References

Basic (Recommended)

1. Tim Caswell: "Learning JavaScript with Object Graphs". Available in: https://howtonode.org/object-graphs, 2011.
   - Note! Although it refers to previous versions of JavaScript, its description of closures is worth reading.
2. Tutorials Point: "ES6 (ECMAScript 6) Quick Guide". Available in: https://www.tutorialspoint.com/es6/es6_quick_guide.htm, July 2018
3. Joyent, Inc.: "Node.js v16.17.0 Documentation", available in: https://www.nodejs.org/dist/latest-v16.x/docs/api/, September 2022.
4. Lydia Hallie: "JavaScript Visualized: Event Loop", available in: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif/, January 2020

Advanced (Non-mandatory)

5. David Flanagan: "JavaScript: The Definitive Guide", 6th ed., O'Reilly Media, 1098 pgs., March 2011. ISBN: 978-0-596-80553-1 (printed edition), 978-0-596-80552-4 (ebook).
6. Marijn Haverbeke: "Eloquent JavaScript", 3rd ed., No Starch Press, 460 pgs., October 2018. Available at: https://eloquentjavascript.net/ (May 2018)
7. Nikhil Marathe: "An Introduction to libuv (Release 1.0.0)", July 2013. Available in: http://nikhilm.github.io/uvbook/index.html
8. Tutorials Point: "ES6 (ECMAScript 6) Tutorial". Available in: https://www.tutorialspoint.com/es6/index.htm, July 2018