Unit 2 – JavaScript and Node.js

Student Guide

1. Introduction

The presentation for this unit consists of three main sections:

1. An introduction that explains why JavaScript and NodeJS have been chosen as the programming language and interpreter, respectively, to be used in this subject.
2. A second section on the JavaScript programming language.
3. A third section on the NodeJS interpreter.

The presentation file lists the main features in those sections. This student guide should provide a longer explanation in each one of those parts. However, the guides in this subject cannot provide as many information as the existing published books on JavaScript. Therefore, instead of summarising the contents of any of those books, we will provide pointers to some sections in several freely available sites or books. In this way, the interested student may get enough information for advancing in these parts of this subject. The same happens in regard to the NodeJS interpreter.

Let us analyse which sections of several reference documents are worth reading in each part (JavaScript and NodeJS). Later on, this guide includes a few sections that describe some relevant concepts.

2. JavaScript

Our presentation on JavaScript mentions this possible structure for its contents:

1. Main/basic characteristics.
2. Code execution alternatives. This section is self-contained. Nothing else is needed in this regard.
3. Language syntax.
4. Primitive and compound values.
5. Variables. Scope. Closures.
6. Operators.
7. Statements.
8. Functions.
9. Arrays.
10. Functional programming.
11. Objects.
12. Serialisation. JSON.
13. Callbacks. Asynchronous execution. Promises.
14. Events.

That presentation only provides a short summary for some of those sections. Its contents may be extended in the following URL: https://www.tutorialspoint.com/es6/index.htm. There, the Tutorials Point website provides a tutorial on ECMAScript 6 (the release of JavaScript assumed by the tools used in our subject). There is also a shorter quick guide, that summarises the tutorial contents at: https://www.tutorialspoint.com/es6/es6_quick_guide.htm.

From that tutorial, we should only consider these parts:

- Overview: It covers section 1 in our list.
- Environment: It describes how to install NodeJS and Visual Studio Code in multiple systems. Both tools are used in our subject.
- Syntax: It covers sections 3 and 5 in our list.
- Variables: This covers section 4 and 5 in our list.
- Operators: This covers section 6 in our list.
- Decision making: It partially covers section 7.
- Loops: This partially covers section 7.
- Functions: This covers sections 8 (and, partially, section 5) in our list.
- Arrays: This covers section 9 in our list.
- Classes: This covers, in an advanced way, section 11 in our list.
- Promises: This covers, in a succint way, sections 13 and 14 in our list.
- Error handling: Although this part is not directly discussed in our presentation of this unit, it is relevant. We recommend to read it, although the subpart about onerror() cannot be applied to NodeJS, since it is only relevant in a web browser.

The rest of that tutorial covers parts of the JavaScript language that are only relevant for web browsers when they manage an HTML page that contains JavaScript code.

Note that the tutorial does not cover completely all the contents in our presentation. For instance, sections 5 (Closures), 10 and 12 have not been covered. Section 10 is not needed in our subject and section 12 may be immediately understood once an example is found. On the other hand, closures are very important. Therefore, we need any complementary source for understanding what a JavaScript closure is.

A first solution to the latter problem is this short discussion on closures that is available at the w3schools.com site: https://www.w3schools.com/js/js_function_closures.asp.

Another solution consists in reading any of the available books on JavaScript. There are many. Eloquent JavaScript, 3rd edition, from Marijn Haverbeke, is a good example. It consists of a short introduction and 21 chapters. From that set, we recommend the following chapters:

- Values, types and operators (Chapter 1): It covers the sections 1, 3, 4 and 6 (partially, the latter) in our list.
- Program structure (Chapter 2): It covers sections 5, 6 and 7 from our list.
- Functions (Chapter 3): This chapter covers sections 5 and 8 in our list.
- Data structures: Objects and arrays (Chapter 4): It covers sections 9, 11 and 12 from our list.
- Higher-order functions (Chapter 5): It covers in depth what is summarised in section 10 from our list. This may be considered an optional reading, since section 10 is only included in Unit 2 for completeness. It is not a central part in our subject.
- Asynchronous programming (Chapter 11): This chapter covers in depth the contents from sections 13 and 14 in our list.

Besides, that book provides multiple short program examples and a list of exercises in each chapter.

In order to extend those external references and book, we provide in the following subsections some descriptions on those topics that are not discussed in depth. Their section numbers match those used in the presentation contents.

2.4.3. Functions. Closures

In some cases, it is useful to define a function inside another, returning the internal function. This internal function may still use the variables or parameters declared in its encompassing function. This is still valid once the encompassing function has completed its execution. The fact that this external scope is maintained and remains valid is known as a "function closure".

Let us describe an example of use of closures. The "log()" function in the Math standard JavaScript module returns the natural logarithm of its received argument. We may use closures for implementing a function that "builds" and returns another that computes logarithms in any specified base. To this end, the following logarithm property is considered:

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}.$$

And the needed fragment of JavaScript code is:

```
function logBase(b) {
   return function(x) {
          return Math.log(x)/Math.log(b);
   }
}
```

In this example, the builder function is "logBase()" and it uses a parameter "b" for specifying the base to be used. Its single sentence returns an internal function. Such function is anonymous and receives a single parameter: "x". The code of such returned function applies the logarithm property that has been presented above, remembering the argument received in the builder function.

Using these functions, we may generate new functions that compute logarithms in any given base. In the following code fragment, two functions are generated for using base 2 and base 8. The resulting code is quite intuitive:

```
log2 = logBase(2);
log8 = logBase(8);

console.log("Logarithm with base 2 of 1024 is: " + log2(1024));  // 10
console.log("Logarithm with base 2 of 1048576 is: " + log2(1048576)); // 20
console.log("Logarithm with base 8 of 4096 is: " + log8(4096));  // 4
```

In some cases, we may use some of the local objects in the function that provides the closure scope, instead of using only one of its parameters. In that case, it should be noted that those objects are accessed by reference and this might have unexpected results.

Let us see an example. In the following sample we want to develop a function that returns an array holding three functions. Each function returns the name and population of each one of the three most populated countries in the world. Unfortunately, this version does not work:

```
 1: function populations() {
 2:    const pops = [1365590000, 1246670000, 318389000];
 3:    const names = ["China", "India", "USA"];
 4:    const placeholder = ["1st", "2nd", "3rd", "4th"];
 5:    let array = [];
 6:    for (i=0; i<pops.length; i++)
 7:          array[i] = function() {
 8:             console.log("The " + placeholder[i] +
 9:                         " most populated country is " +
10:                         names[i] + " and its population is " +
11:                         pops[i]);
12:          };
13:    return array;
14: }
15:
16: let ps = populations();
17:
18: first = ps[0];
19: second = ps[1];
20: third = ps[2];
21:
22: first();
23: second();
24: third();
```

The sentence being used in line 16 calls the “populations()” function. We expect that the “ps” array gets the three requested functions. In that case, the sentences shown in lines 22 to 24 would print the requested information. However, their output is the following:

```
The 4th most populated country is undefined and its population is undefined
The 4th most populated country is undefined and its population is undefined
The 4th most populated country is undefined and its population is undefined
```

This may be explained because when the functions “first()”, “second()” and “third()” are called, the “i” variable in the “populations()” function holds a value of 3. The access to such variable is made by reference. So, we are accessing to “names[3]” (“4th”) and “pops[3]” (undefined). It does not matter that when such functions were created the value of “i” was 0, 1 and 2, respectively. Currently, since “I” is a global variable, its value is 3 and this is the used value. Because of this, the output being printed is not what we tried to print.

We should use function closures in order to access to “i”. Our aim is that the function in lines 7 to 12 remembers the value of “i” in that iteration of the loop. We are not interested in the value of “i” when we call that function but in the value of “i” when the function is stored in the array.

A correct solution is shown here:

```
 1: function populations() {
 2:    const pops = [1365590000, 1246670000, 318389000];
 3:    const names = ["China", "India", "USA"];
 4:    const placeholder = ["1st", "2nd", "3rd", "4th"];
 5:    let array = [];
 6:    for (i=0; i<pops.length; i++)
 7:          array[i] = function(x) {
 8:             return function() {
 9:                   console.log("The " + placeholder[x] +
10:                               " most populated country is " +
11:                               names[x] + " and its population is " +
12:                               pops[x]);
13:             };
14:          }(i);
15:    return array;
16: }
17:
18: let ps = populations();
19:
20: first = ps[0];
21: second = ps[1];
22: third = ps[2];
23:
24: first();
25: second();
26: third();
```

The changes are highlighted in boldface. We need another encompassing function that receives as its single argument the current value of “i”. To this end, this new function receives an “x” parameter and the function being returned uses “x” instead of “i”. So, in each iteration of the loop the encompassing function is invoked using “i” as its argument (this is the aim of the current line 14).

Using this hint, the new program output is:

```
The 1st most populated country is China and its population is 1365590000
The 2nd most populated country is India and its population is 1246670000
The 3rd most populated country is USA and its population is 318389000
```

This is the expected output.

ECMAScript 6 introduced a second way of solving this problem. To this end, it set another approach for declaring variables. Traditionally, the “var” keyword had been used to this end. ECMAScript 6 added the “let” keyword. Then, if the variable used as the loop counter is declared with “let”, each time it is used in the loop its current value is passed instead of a reference (i.e., pointer) to the variable. Thus, instead of using closures, we could use “let” when we declare variable “i” in the loop of the program. The resulting program version is:

```
 1: function populations() {
 2:    const pops = [1365590000, 1246670000, 318389000];
 3:    const names = ["China", "India", "USA"];
 4:    const placeholder = ["1st", "2nd", "3rd", "4th"];
 5:    let array = [];
 6:    for (let i=0; i<pops.length; i++)
 7:          array[i] = function() {
 8:             console.log("The " + placeholder[i] +
 9:                         " most populated country is " +
10:                         names[i] + " and its population is " +
11:                         pops[i]);
12:          };
13:    return array;
14: }
15:
16: let ps = populations();
17:
18: first = ps[0];
19: second = ps[1];
20: third = ps[2];
21:
22: first();
23: second();
24: third();
```

Now, its output is identical to that obtained in the previous program variant based on closures. Thus, it is correct.

2.5. Events

This guide describes events in detail in its Section 3.2, where the Node.js "events" module is also presented.

2.6. Callbacks

Implementation of callbacks in asynchronous functions

Asynchronous programming in Node.js is based on the usage of callback functions. When a program contains an invocation to an asynchronous function (e.g., ***f_async***) that has a callback as its last parameter, such program will not block its execution until ***f_async*** is completed. Instead, it goes on with the sentences that follow ***f_async***. In the meantime, that asynchronous function is going on and, once it is completed (in a forthcoming turn) its callback is run.

In "Control Flow in Node"[1], Tim Caswell shows some examples that illustrate how asynchronous functions and callbacks work. The next example is based on one from Caswell.

[1] Control Flow in Node - Tim Caswell: http://howtonode.org/control-flow, http://howtonode.org/control-flow-part-ii

```
 1: const fs = require('fs');
 2:
 3: fs.readFile('mydata.txt', function (err, buffer) {
 4:   if (err) {
 5:     // Handle error
 6:     console.error(err.stack);
 7:     return;
 8:   }
 9:   // Do something
10:   console.log( buffer.toString() );
11: });
12:
13: console.log('Other statements...');
14: console.log('sqrt(2) =', Math.sqrt(2));
```

The call to the asynchronous ***readFile*** function from module ***fs*** is not blocking this program. Those sentences in lines 13 and 14 are executed before ***readFile*** has terminated and, therefore, before invoking its callback. If this attempt to read 'mydata.txt' fails, the callback will execute the sentence in line 6. In that case, its output will be:

```
Other statements...
sqrt(2) = 1.4142...
Error: ENOENT, open '… /mydata.txt'
```

Function ***readFile*** receives as its first argument the name of the file to be read and "returns" its contents. In a strict sense, it does not return anything; instead, the file contents are passed as the second argument to its callback when the read operation terminates successfully. On the other hand, the first callback argument (*err* in this example) receives the error message when the read operation has found any problem (as it has been assumed here).

Callback functions are usually anonymous (i.e., they do not have any identifier to be used as their name) when they are used just once, as in the previous example. However, if they need to be used multiple times, it is convenient to define them with a name. The following example, taken from Caswell's guide, illustrates this approach.

```
 1: const fs = require('fs');
 2:
 3: function callback(err, buffer) {
 4:   if (err) {
 5:     // Handle error
 6:     console.error(err.stack);
 7:     return;
 8:   }
 9:   // Do something
10:   console.log( buffer.toString() );
11: }
12:
13: fs.readFile('mydata.txt', callback);
14: fs.readFile('rolodex.txt', callback);
```

In some cases, a sequence of asynchronous functions needs to be used in a given program. When that happens, those functions invocations are nested in their respective callbacks. In "Accessing the File System in Node.js"[2], Colin Ihrig presents an example of this kind:

```
 1: const fs = require("fs");
 2: const fileName = "foo.txt";
 3:
 4: fs.exists(fileName, function(exists) {
 5:   if (exists) {
 6:     fs.stat(fileName, function(error, stats) {
 7:       fs.open(fileName, "r", function(error, fd) {
 8:         var buffer = new Buffer(stats.size);
 9:
10:         fs.read(fd, buffer, 0, buffer.length, null, function(error, bytesRead, buffer) {
11:           var data = buffer.toString("utf8", 0, buffer.length);
12:
13:           console.log(data);
14:           fs.close(fd);
15:         });
16:       });
17:     });
18:   }
19: });
```

In this example, the contents of a file are read using a buffer. The callback of ***exists*** contains a call to another asynchronous function, ***stat***, whose callback invokes another asynchronous function, ***open***, whose callback calls another asynchronous function, ***read***, that also has a callback.

There are other simpler ways of reading files, e.g. using ***readFile*** as in the examples presented by Caswell or using synchronous versions from the functions in module ***fs***. This last example only wants to show that, at times, callbacks may be deeply nested... and this may raise several problems. To begin with, the resulting program is hard to follow. In this example, additionally, there is no error management. With a minimal error management, the resulting program would be longer and harder to read and interpret. In those cases, it is convenient to replace callbacks with promises in order to implement asynchronous functions.

Callback nesting is a way to ensure that a set of asynchronous operations are executed sequentially. But callbacks could be also used to group and parallelise that set of asynchronous operations. An example of the latter is also provided by Caswell: to read all files in a given directory. The resulting program is the following:

```
1: const fs = require('fs');
2:
3: fs.readdir('.', function (err, files) {
```

[2] Accessing the File System in Node.js - Colin Ihrig: http://www.sitepoint.com/accessing-the-file-system-in-node-js/

```
 4:    let count = files.length,
 5:    results = {};
 6:    files.forEach(function (filename) {
 7:      fs.readFile(filename, function (er2, data) {
 8:        console.log(filename, 'has been read');
 9:        if (!er2) results[filename] = data.toString();
10:        count--;
11:        if (count <= 0) {
12:          // Do something once we know all the files are read.
13:          console.log('\nTOTAL:', files.length, 'files have been read');
14:        }
15:      });
16:    });
17: });
```

The ***readdir*** callback receives as its second argument an array with the names of all files in the given directory (the current directory in this example). With this array, using the ***forEach*** method, it reads each one of those files using ***readFile***. Thus, all read file operations are executed in parallel and may terminate in any order. As they are terminated, their respective callbacks are run. In those callbacks the ***count*** variable is increased. This allows detecting when all read operations have concluded, printing a message in that case.

Emulating asynchronous functions

In the examples of the previous section the programs invoked already existing asynchronous functions, taken from the ***fs*** module. Other standard Node.js modules also provide asynchronous functions. In all those cases, the programmer only needs to call those functions, providing appropriate arguments (and this implies that a correct callback should be implemented). In this section we will explain how to convert an initially synchronous function into another that is split in multiple execution turns, emulating an asynchronous behaviour.

Please consider the following program:

```
 1: // ***  fibo1.js
 2:
 3: function fibo(n) {
 4:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
 5: }
 6:
 7: function fact(n) {
 8:    return (n<2) ? 1 : n * fact(n-1)
 9: }
10:
11: console.log('Starting the program...')
12: console.log('fibo(40) =', fibo(40))
13: console.log('Fibonacci...')
14: console.log('fact(10) =', fact(10))
15: console.log('Factorial...')
```

The two functions being defined in this example are synchronous: ***fibo*** computes the n-th term of the Fibonacci succession and ***fact*** computes the factorial of the number given as its argument. Since those functions are synchronous, their calls in lines 12 and 14 block such a program. Therefore, its output is:

```
Starting the program...
fibo(40) = 165580141
Fibonacci...
fact(10) = 3628800
Factorial...
```

Moreover, the second line in that output takes several seconds to be shown due to the large amount of mathematical operations needed to compute it. It would be convenient to implement those functions in an asynchronous way, preventing the program from being blocked in lines 12 and 14 and allowing an immediate printing of the messages in lines 13 and 15.

An easy way to emulate an asynchronous execution of those functions consists in using ***console.log*** as their callback and using ***setTimeout*** to delay the call of those functions to a later turn. With those modifications, the resulting program is:

```
 1: // *** fibo2.js
 2:
 3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 4:
 5: function fibo_back1(n,cb) {
 6:    let m = fibo(n)
 7:    cb('fibonacci('+n+') = '+m)
 8: }
 9:
10: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
11:
12: function fact_back1(n,cb) {
13:    let m = fact(n)
14:    cb('factorial('+n+') = '+m)
15: }
16:
17: console.log('Starting the program...')
18: setTimeout( function(){
19:    fibo_back1(40, console.log)
20: }, 2000 )
21: console.log('Fibonacci...')
22: setTimeout( function(){
23:    fact_back1(10, console.log)
24: }, 1000 )
25: console.log('Factorial...')
```

And its resulting output will be:

```
Starting the program...
Fibonacci...
Factorial...
factorial(10) = 3628800
fibonacci(40) = 165580141
```

Only the execution of the last line in this output is presented with any delay.

Although the previous program runs correctly, the function being used as a callback (***console.log***) does not respect the Node.js conventions. In Node.js, callback functions regularly declare as their first parameter an error (in order to receive error messages when the asynchronous function fails) and as their second parameter the asynchronous function result (but only when no error has happened).

If we adapt the previous program in order to respect those conventions, the result is:

```
 1: // ***  fibo3.js
 2:
 3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 4:
 5: function fibo_back2(n,cb) {
 6:    let err = eval_err(n,'fibonacci')
 7:    let m   = err ? '': fibo(n)
 8:    cb(err,'fibonacci('+n+') = '+m)
 9: }
10:
11: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
12:
13: function fact_back2(n,cb) {
14:    let err = eval_err(n,'factorial')
15:    let m   = err ? '': fact(n)
16:    cb(err,'factorial('+n+') = '+m)
17: }
18:
19: function show_back(err,res) {
20:    if (err) console.log(err)
21:    else     console.log(res)
22: }
23:
24: function eval_err(n,s) {
25:    return (typeof n != 'number') ?
26:              s+'('+n+') ??? : '+n+' is not a number' : ''
27: }
28:
29: console.log('Starting the program...')
30: setTimeout( function(){
31:    fibo_back2(40, show_back)
32:    fibo_back2('Pep', show_back)
```

```
33: }, 2000 )
34: console.log('Fibonacci...')
35: setTimeout( function(){
36:    fact_back2(10, show_back)
37:    fact_back2('Ana', show_back)
38: }, 1000 )
39: console.log('Factorial...')
```

Now the asynchronous functions ***fibo_back2*** and ***fact_back2*** receive as their callback ***show_back*** (implemented in lines 19 to 22). Another auxiliary function has been added, ***eval_err*** (in lines 24 to 27). It provides an error messages when needed; i.e., when the argument being provided is not a number.

The output being obtained when we run this program is:

```
Starting the program...
Fibonacci...
Factorial...
factorial(10) = 3628800
factorial(Ana) ??? : Ana is not a number
fibonacci(40) = 165580141
fibonacci(Pep) ??? : Pep is not a number
```

The functions that we have implemented up to now comply with the syntactic conventions for asynchronous functions but their behaviour is not yet asynchronous: it is being emulated using ***setTimeout***.

Another emulation approach (a bit more realistic) consists in using function ***nextTick*** from module ***process***[3]. That function delays the execution of an action till the next iteration of the event loop. Let us use ***nextTick*** in the following adaptation of our "fibo" program:

```
 1: // ***  fibo4.js
 2:
 3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 4:
 5: function fibo_async(n,cb) {
 6:    process.nextTick(function(){
 7:       let err = eval_err(n,'fibonacci')
 8:       let m   = err ? '': fibo(n)
 9:       cb(err,'fibonacci('+n+') = '+m)
10:    });
11: };
12:
13: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
14:
```

[3] Understanding process.nextTick() - Kishore Nallan: http://howtonode.org/understanding-process-next-tick

```
15: function fact_async(n,cb) {
16:    process.nextTick(function(){
17:       let err = eval_err(n,'factorial')
18:       let m   = err ? '': fact(n)
19:       cb(err,'factorial('+n+') = '+m)
20:    });
21: }
22:
23: function show_back(err,res) {
24:    if (err) console.log(err)
25:    else     console.log(res)
26: }
27:
28: function eval_err(n,s) {
29:    return (typeof n != 'number') ?
30:              s+'('+n+') ??? : '+n+' is not a number' : ''
31: }
32:
33: console.log('Starting the program...')
34: fact_async(10, show_back)
35: fact_async('Ana', show_back)
36: console.log('Factorial...')
37: fibo_async(40, show_back)
38: fibo_async('Pep', show_back)
39: console.log('Fibonacci...')
```

The output from ***fibo4.js*** is the same than that from ***fibo3.js***. Now, lines 33 to 39 do not use any ***setTimeout***. Additionally, the calls to ***fact_async*** (lines 34-35) and ***fibo_async*** (lines 37-38) have exchanged their positions since they no longer use any explicit delay and we still want to show first the results from the factorial calls.

Let us assume that we want to apply those two functions onto a set of values, placing the results in two arrays. A program to implement this is:

```
 1: // ***  fibo5.js
 2:
 3: // *** functions
 4:
 5: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 6:
 7: function fibo_async(n,cb) {
 8:    process.nextTick(function(){
 9:       let err = eval_err(n,'fibonacci')
10:       let m   = err ? '': fibo(n)
11:       cb(err,n,m)
12:    });
13: };
14:
15: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
16:
```

```
17: function fact_async(n,cb) {
18:    process.nextTick(function(){
19:       let err = eval_err(n,'factorial')
20:       let m   = err ? '': fact(n)
21:       cb(err,n,m)
22:    });
23: }
24:
25: function show_fibo_back(err,num,res) {
26:    if (err) console.log(err)
27:    else {
28:       console.log('fibonacci('+num+') = '+res)
29:       fibs[num]=res
30:    }
31: }
32:
33: function show_fact_back(err,num,res) {
34:    if (err) console.log(err)
35:    else {
36:       console.log('factorial('+num+') = '+res)
37:       facts[num]=res
38:    }
39: }
40:
41: function eval_err(n,s) {
42:    return (typeof n != 'number') ?
43:              s+'('+n+') ??? : '+n+' is not a number' : ''
44: }
45:
46: // ***main program
47: console.log('Starting the program...')
48:
49: let facts = []
50: for (let i=0; i<=10; i++)
51:    fact_async(i, show_fibo_back)
52: console.log('Factorials...')
53:
54: let fibs = []
55: for (let i=0; i<=20; i++)
56:    fibo_async(i, show_fact_back)
57: console.log('Fibonaccis...')
```

In this program we have modified the callbacks. Now, we have two different callbacks, each one for each asynchronous function. They hold the result of their computations in the corresponding array slot, besides showing that result in the console.

A summary of the output from this program is:

```
Starting the program...
Factorials...
Fibonaccis...
```

```
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
factorial(10) = 3628800
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(19) = 6765
fibonacci(20) = 10946
```

Now, let us consider an extension of the previous program that computes the addition of all computed factorials and the addition of all computed fibonaccis. A first try could be...:

```
 1: // ***  fibo6a.js
 2:
 3: // *** functions:  implemented as in fibo5.js
 4: function fibo(n) { … }
 5: function fibo_async(n,cb) { … }
 6: function fact(n) { … }
 7: function fact_async(n,cb) { … }
 8: function show_fibo_back(err,num,res) { … }
 9: function show_fact_back(err,num,res) { … }
10: function eval_err(n,s) { … }
11:
12: function add(a,b) { return a+b }
13:
14: // *** main program
15: console.log('Starting the program...')
16:
17: const n = 10
18: let facts = []
19: let fibs = []
20:
21: for (let i=0; i<n; i++)
22:    fact_async(i, show_fact_back)
23: console.log('Factorials...')
24:
25: for (let i=0; i<n; i++)
26:    fibo_async(i, show_fibo_back)
27: console.log('Fibonaccis...')
28:
29: console.log('Factorial addition ['+0+'..'+(n-1)+'] =', facts.reduce(add))
30: console.log('Fibonacci addition ['+0+'..'+(n-1)+'] =', fibs.reduce(add))
```

However, its output is:

```
Starting the program...
Factorials...
Fibonaccis...
```

```
…
… /fibo6a.js:29
   console.log('Factorial addition ['+0+'..'+(n-1)+'] =', facts.reduce(add))

TypeError: Reduce of empty array with no initial value
   at Array.reduce (native)
   …
```

This result (an error when we have tried to call ***reduce*** on an uninitialised empty array) is generated because lines 29 and 30 are run before the asynchronous functions and their callbacks.

A solution to this problem consists in putting the sentences from lines 29 and 30 in an anonymous function that is placed in the Node event queue using ***process.nextTick*** to this end:

```
process.nextTick( function(){
   console.log('Factorial addition ['+0+'..'+(n-1)+'] =', facts.reduce(add))
   console.log('Fibonacci addition ['+0+'..'+(n-1)+'] =', fibs.reduce(add))
});
```

Or, alternatively:

```
setTimeout( function(){
   console.log('Factorial addition ['+0+'..'+(n-1)+'] =', facts.reduce(add))
   console.log('Fibonacci addition ['+0+'..'+(n-1)+'] =', fibs.reduce(add))
}, 1 );
```

Any of those two solutions provides the following output:

```
Starting the program...
Factorials...
Fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
Factorial addition [0..9] = 409114
Fibonacci addition [0..9] = 143
```

Another alternative consists in modifying the callbacks to, once all array slots have been computed, compute the additions. Such solution is:

```
1:  // *** fibo6b.js
2:
```

```
 3: // *** functions:  implemented as in fibo6a.js, except their callbacks.
 4: function fibo(n) { … }
 5: function fibo_async(n,cb) { … }
 6: function fact(n) { … }
 7: function fact_async(n,cb) { … }
 8: function eval_err(n,s) { … }
 9: function add(a,b) { … }
10:
11: function show_fibo_back(err,num,res) {
12:    if (err) console.log(err)
13:    else {
14:       console.log('fibonacci('+num+') = '+res)
15:       fibs[num]=res
16:       if (num==n-1) {
17:          let s = fibs.reduce(add)
18:          console.log('Fibonacci addition ['+0+'..'+(n-1)+'] =', s)
19:       }
20:    }
21: }
22:
23: function show_fact_back(err,num,res) {
24:    if (err) console.log(err)
25:    else {
26:       console.log('factorial('+num+') = '+res)
27:       facts[num]=res
28:       if (num==n-1) {
29:          let s = facts.reduce(add)
30:          console.log('Factorial addition ['+0+'..'+(n-1)+'] =', s)
31:       }
32:    }
33: }
34:
35: // *** main program
36: console.log('Starting the program...')
37:
38: const n = 10
39: let facts = []
40: let fibs = []
41:
42: for (let i=0; i<n; i++)
43:    fact_async(i, show_fact_back)
44: console.log('Factorials...')
45:
46: for (let i=0; i<n; i++)
47:    fibo_async(i, show_fibo_back)
48: console.log('Fibonaccis...')
```

The output from ***fibo6b.js*** is:

```
Starting the program...
Factorials...
```

```
Fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
Factorial addition [0..9] = 409114
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
Fibonacci addition [0..9] = 143
```

This output is correct. It only differs from those solutions given in ***fact6a.js*** in the order in which the results are shown.

2.6.2. Promises

There is another way of supporting asynchronous executions in JavaScript: using promises. The ECMAscript 6 standard has unified the existing variants for building JavaScript promises (there were many supporting alternatives, being implemented by several node modules). The standardised solution is based on a Promise class, building promises using its default constructor (i.e., with the **new** keyword).

This section uses that variant, revising the examples shown in the previous section about the Fibonacci series, but using promises in this case.

A first example program is:

```
 1: // ***  fibo7.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fibo - new-promise version
 7: function fibo_promise(n) {
 8:    return new Promise(function(fulfill, reject) {
 9:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
10:          // Unfortunately, fibo() is a synchronous function. We should
11:          // convert it into something apparently "asynchronous". To this end,
12:          // we place its computation in the next scheduler turn.
13:       else    setTimeout( function() {fulfill( fibo(n) )}, 1 )
14:    })
15: }
16:
17: // onFulfilled handler, with closure
18: function onSuccess(i) {
19:    return function (res) { console.log('fibonacci('+i+') =', res) }
20: }
21:
22: // onRejected handler
```

```
23: function onError(err) { console.log('Error:', err); }
24:
25: // *** main program
26: console.log('Execution is starting...')
27: const elems = [25, '5', true]
28:
29: for (let i in elems)
30:    fibo_promise(elems[i]).then( onSuccess(elems[i]), onError )
```

In function ***fibo_promise*** we construct and return a promise object. In the function being passed as the single argument for that constructor we should take care of setting the ***onSuccess*** handler as its first parameter and the ***onError*** handler as its second, calling them when needed.

The following program solves the problem of computing factorials and fibonaccis holding their results in two arrays. Its implementation based on promises is:

```
 1: // ***  fibo8.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fibo - promise version
 7: function fibo_promise(n) {
 8:    return new Promise(function(fulfill, reject) {
 9:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
10:       else    setTimeout( function() {fulfill( fibo(n) )}, 1 )
11:    })
12: }
13:
14: // fact - sync
15: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
16:
17: // fact - promise version
18: function fact_promise(n) {
19:   return new Promise(function(fulfill, reject) {
20:      if (typeof(n)== 'number' ) setTimeout( function() {fulfill( fact(n) )}, 1 )
21:      else   reject( n+' is an incorrect arg' )
22:   })
23: }
24:
25: // onFulfilled handler, with closure
26: function onSuccess(s, x, i) {
27:    return function (res) {
28:       if ( x!=null ) x[i] = res
29:       console.log(s+'('+i+') =', res)
30:    }
31: }
32:
33: // onRejected handler
```

```
34: function onError(err) { console.log('Error:', err); }
35:
36: // *** main program
37: console.log('Execution is starting...')
38:
39: const n = 10
40: let fibs = []
41: let fibsPromises = []
42: let facts = []
43: let factsPromises = []
44:
45: // Generate the promises.
46: for (let i=0; i<n; i++) {
47:    fibsPromises[i] = fibo_promise(i)
48:    factsPromises[i] = fact_promise(i)
49: }
50:
51: // Show the results.
52: for (let i=0; i<n; i++) {
53:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
54:    factsPromises[i].then( onSuccess('factorial', facts, i), onError )
55: }
```

Its output is:

```
Execution is starting...
fibonacci(0) = 1
factorial(0) = 1
…
fibonacci(9) = 55
factorial(9) = 362880
```

Let us extend that program, presenting also the result of adding all slots in each one of the arrays. The resulting program is:

```
 1: // ***  fibo9.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fibo - promise version
 7: function fibo_promise(n) {
 8:    return new Promise(function(fulfill, reject) {
 9:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
10:       else    setTimeout( function() {fulfill( fibo(n) )}, 1 )
11:    })
12: }
13:
14: // fact - sync
15: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
```

```
16:
17: // fact - promise version
18: function fact_promise(n) {
19:   return new Promise(function(fulfill, reject) {
20:     if (typeof(n)== 'number' ) setTimeout( function() {fulfill( fact(n) )}, 1 )
21:     else   reject( n+' is an incorrect arg' )
22:   })
23: }
24:
25: // onFulfilled handler, with closure
26: function onSuccess(s, x, i) {
27:    return function (res) {
28:       if ( x!=null ) x[i] = res
29:       console.log(s+'('+i+') =', res)
30:    }
31: }
32:
33: // onRejected handler
34: function onError(err) { console.log('Error:', err); }
35:
36: // onFulfilled handler, for array of promises
37: function sumAll(z, x) {
38:    return function () {
39:       let s = 0
40:       for (let i in x) s += x[i]
41:       console.log(z, '=', x, '; sum =', s)
42:    }
43: }
44:
45: // onRejected handler, for array of promises
46: function showFinalError() {
47:    console.log('Something wrong has happened...')
48: }
49:
50: // *** main program
51: console.log('Execution is starting...')
52:
53: const n = 10
54: let fibs = []
55: let fibsPromises = []
56: let facts = []
57: let factsPromises = []
58:
59: // Generate the promises.
60: for (let i=0; i<n; i++) {
61:    fibsPromises[i] = fibo_promise(i)
62:    factsPromises[i] = fact_promise(i)
63: }
64:
65: // Show the results.
66: for (let i=0; i<n; i++) {
67:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
```

```
68:     factsPromises[i].then( onSuccess('factorial', facts, i), onError )
69: }
70: // Show the summary.
71: Promise.all(fibsPromises).then( sumAll('fibs', fibs), showFinalError )
72: Promise.all(factsPromises).then( sumAll('facts', facts) )
```

The output from ***fibo9.js*** is:

```
Execution is starting...
fibonacci(0) = 1
factorial(0) = 1
…
fibonacci(9) = 55
factorial(9) = 362880
fibs = [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ] ; addition = 143
facts = [ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880 ] ; addition = 409114
```

As it has been shown, promise arrays solve easily the problem of computing the addition of array elements. The onSuccess handler, ***addAll***, is run only once all promises have been resolved; i.e., once all values to be stored in the array have been computed and held.

Chaining promises with ***then*** provides an easy mechanism for setting the sequence in which a set of actions should be run. Moreover, the resulting code is also easy to read (at least, easier than a solution exclusively based on callbacks).

In programs ***fibo8.js*** and ***fibo9.js***, functions ***fibo_promise*** and ***fact_promise*** are extremely similar. In the next program we provide an equivalent but shorter implementation:

```
 1: // *** fibo10.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fact - sync
 7: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
 8:
 9: // "func" - promise version
10: function func_promise(f,n) {
11:     return new Promise(function(fulfill, reject) {
12:         if (typeof(n)!='number') reject( n+' is an incorrect argument' )
13:         else     setTimeout( function() {fulfill( f(n) )}, 1 )
14:     })
15: }
16:
17: // onFulfilled handler, with closure
18: function onSuccess(s, x, i) {
19:     return function (res) {
20:         if ( x!=null ) x[i] = res
21:         console.log(s+'('+i+') =', res)
```

```
22:     }
23: }
24:
25: // onRejected handler
26: function onError(err) { console.log('Error:', err); }
27:
28: // onFulfilled handler, for array of promises
29: function sumAll(z, x) {
30:     return function () {
31:         let s = 0
32:         for (let i in x) s += x[i]
33:         console.log(z, '=', x, '; sum =', s)
34:     }
35: }
36:
37: // onRejected handler, for array of promises
38: function showFinalError() {
39:     console.log('Something wrong has happened...')
40: }
41:
42: // *** main program
43: console.log('Execution is starting...')
44:
45: const n = 10
46: let fibs = []
47: let fibsPromises = []
48: let facts = []
49: let factsPromises = []
50:
51: // Generate the promises.
52: for (let i=0; i<n; i++) {
53:     fibsPromises[i]  = func_promise(fibo, i)
54:     factsPromises[i] = func_promise(fact, i)
55: }
56:
57: // Show the results.
58: for (let i=0; i<n; i++) {
59:     fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
60:     factsPromises[i].then( onSuccess('factorial', facts, i), onError )
61: }
62:
63: // Show the summary.
64: Promise.all(fibsPromises).then( sumAll('fibs', fibs), showFinalError )
65: Promise.all(factsPromises).then( sumAll('facts', facts) )
```

3. NodeJS

The official documentation page (https://nodejs.org/en/docs/) from the NodeJS website provides all the information needed in this second half of the unit. Please refer to it in case of need. Note that it contains two types of documentation:

- The API reference. From the initial documentation page, we may reach that reference for the currently latest and LTS releases. However, we may have an older version in the labs. Indeed, the first release covering ECMAScript 6 was the NodeJS 6.x.x one. Its API reference is available here: https://nodejs.org/docs/latest-v6.x/api/. You do not need to know or memorise all those available operations.
- Complementary guides. Available at: https://nodejs.org/en/docs/guides/. There are three types of guides: (1) general, (2) core concepts and (3) module-related. The second type describes some central concepts of the NodeJS runtime, as JavaScript timers, the runtime event loop, differences among blocking and non-blocking operations, etc. Those guides that belong to the second type are a recommended reading.

Besides those references, the following sections describe how NodeJS deals with its internal event/turn queue and with its modules, since the documentation on those issues is scarce.

3.2. Asynchrony. Event Queue

The NodeJS runtime maintains an event queue. This is needed because JavaScript does not support multiple threads of execution. Because of this, when a new activity is generated in a program, that new activity is implemented as a new "event" and it is inserted at the end of the queue. That queue is managed in FIFO order. This means that the service of a new event demands that the previous one was completed.

Let us analyse this example:

```
 1: function fibo(n) {
 2:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1);
 3: }
 4: console.log("Starting...");
 5: // Wait for 10 ms to write the message.
 6: // A new event is generated.
 7: setTimeout( function() {
 8:   console.log( "M1: First message..." );
 9: }, 10 );
10: // More than 5 seconds needed.
11: let j = fibo(40);
12: function anotherMessage(m,u) {
13:    console.log( m + ": Result is: " + u );
14: }
15: // M2 is written before M1 since the main
16: // thread has not been suspended...
17: anotherMessage("M2",j);
18: // M3 is written after M1.
19: // It is scheduled in 1 ms.
20: setTimeout( function() {
21:  anotherMessage("M3",j);
22: }, 1 );
```

Function setTimeout() is used in JavaScript for scheduling the execution of the function received in its first parameter after the number of milliseconds specified in its second parameter.

Let us follow a trace of this program:

- Execution is started in lines 1 to 3, where the recursive function “fibo()” is defined. It receives an integer argument and computes the Fibonacci number associated to that value.
- Line 4 prints the message “Starting...” on screen.
- Lines 7 to 9 use setTimeout() to schedule the printing of another message (“M1: First message...”) after 10 ms. At the moment, this has no effect.
- Line 11 invokes the “fibo()” function with argument 40. Its execution lasts several seconds. In that term, the interval of 10 ms mentioned in the previous paragraph is concluded. This means that the function that prints the M1 message is already placed in the event queue, waiting for its turn. Such turn will not be started until the main thread is terminated.
- This main thread eventually arrives to lines 12 to 14, where the function anotherMessage() is defined. Such function will be used later for printing the obtained Fibonacci number.
- Execution is now at line 17. It prints the result. The message being printed is “M2: Result is: 165580141”.
- Finally, lines 20 to 22 are executed, where the program schedules the execution of anotherMessage() (M3 message) after 1 ms. This terminates the execution of the main thread.
- At this time, there is only a single event in the queue. Its execution is started. Such event prints message “M1: First message...” on screen. While this message is being printed, the 1 ms interval started in the previous point terminates. As a result, a new context is inserted in the event queue. Its aim is to print the M3 message.
- Once M1 has been printed, such first event is concluded. The event queue is revised and another context is found. Its execution is started, printing this message “M3: Result is: 165580141”.
- This terminates the program execution. Its full output is:

```
Starting...
M2: Result is: 165580141
M1: First message...
M3: Result is: 165580141
```

As it can been seen, the two first lines were printed by the main thread. In turn, the M1 and M3 messages were printed using two events. Although the first of those events was generated very soon (while the function fibo() was being executed; before printing message M2) its result could not be shown at that time on screen. The main thread of the program needs to terminate before those additional events started their execution.

Let us revise this other example:

```
 1: /*************************************************************/
 2: /* Events1.js                                                */
 3: /*************************************************************/
 4: const ev = require('events')
 5: const emitter = new ev.EventEmitter()  // DON'T FORGET NEW OPERATOR !!!!!
 6:
 7: const e1 = "print", e2 = "read"              // Names of the events.
 8:
 9: function createListener( eventName ) {
10:         let num = 0
11:         return function () {
12:                 console.log("Event " + eventName + " has " +
13:                         "happened " + ++num + " times.")
14:         }
15: }
16:
17: // Listener functions are registered in the event emitter.
18: emitter.on(e1, createListener(e1))
19: emitter.on(e2, createListener(e2))
20: // There might be more than one listener for the same event.
21: emitter.on(e1, function() {
22:         console.log( "Something has been printed!!")
23: })
24:
25: // Generate the events periodically...
26: setInterval(function() {emitter.emit(e1)}, 2000) // First event emitted every 2s
27: setInterval(function() {emitter.emit(e2)}, 3000) // Second event emitted every 3s
```

This program creates an "emitter" object in line 5 and defines two constants in line 7: the names of the events to handle. Later, in lines 9 to 15, it defines a function createListener() in order to generate (with a closure that holds the event name and how many times such event happened) the callbacks that should manage each event. Lines 18 to 23 associate those listeners to their corresponding events. Note that event "e1" (print) has two listeners. Finally, in line 26, event "e1" (print) is scheduled every 2 seconds and in line 27 event "e2" (read) is scheduled every 3 seconds. After this, the main thread is terminated.

Every two seconds, an anonymous function that calls emit(e1) is inserted in the event queue. When that context becomes the first in the queue, it will synchronously call the two listeners for the "e1" event (in the order the program associated them to that event using emitter.on()). Thus, the first listener prints the number of times such event has happened, while the second prints the message "Something has been printed!!". Note that the execution of emit() and those two listeners happen in the same indivisible turn, since calls to listeners are synchronous and sequential. Something similar occurs in regard to event "e2", every three seconds.

Since the raising intervals for these events are long and the main program has no other pending activities, the event queue usually remains empty. Each time those intervals elapse, an anonymous function that calls emit() is inserted in the event queue, but it starts almost immediately, runs all its associated listeners and empties again the event queue.

An initial fragment of the output is...

```
Event print has happened 1 times.
Something has been printed!!
Event read has happened 1 times.
Event print has happened 2 times.
Something has been printed!!
Event read has happened 2 times.
Event print has happened 3 times.
Something has been printed!!
Event print has happened 4 times.
Something has been printed!!
Event read has happened 3 times.
Event print has happened 5 times.
Something has been printed!!
```

As an easy exercise, justify which is the output of the following example, where the previous program is slightly extended.

```
 1: /*************************************************************/
 2: /* Events2.js                                                */
 3: /*************************************************************/
 4: const ev = require('events')
 5: const emitter = new ev.EventEmitter()  // DON'T FORGET NEW OPERATOR!!
 6:
 7: const e1 = "print", e2 = "read"              // Names of the events.
 8:
 9: function createListener( eventName ) {
10:         let num = 0
11:         return function () {
12:                 console.log("Event " + eventName + " has " +
13:                         "happened " + ++num + " times.")
14:         }
15: }
16:
17: // Listener functions are registered in the event emitter.
18: emitter.on(e1, createListener(e1))
19: emitter.on(e2, createListener(e2))
20: // There might be more than one listener for the same event.
21: emitter.on(e1, function() {
22:         console.log( "Something has been printed!!")
23: })
24:
25: // Generate the events periodically...
26: setInterval(function() {emitter.emit(e1)}, 2000) // First event emitted every 2s
27: setInterval(function() {emitter.emit(e2)}, 3000) // Second event emitted every 3s
28: // Loop.
29: while (true)
30:    console.log(".")
```

3.3. Module Management

The presentation describes how to export "public" functions in a module, using "`exports`", and how they are imported using "`require()`". That slide also presents some examples of their use.

But "`exports`" is only an alias for "`module.exports`". Moreover, both "`module.exports`" and "`module`" are regular JavaScript objects. This means that they hold a dynamic set of properties (i.e., attributes) that we could enlarge or decrease at our own. Each module has a private "`module`" object. It is not a global object being shared by all source files in a given application. Using this object we may specify which operations and objects are exported in a module.

In order to add a property or method to an object, we only need to define them, assigning an initial value, if needed. Both "`area()`" and "`circumference()`" are used as new properties of the "`module.exports`" object. Since they are functions, they are exported as public operations of that module.

For removing a property we only need the "**delete**" keyword, followed by the name of the property to be removed.

This mechanism allows building modules that import other modules, modifying some of their operations but maintaining the others. For instance, this sample of code...

```
var c = require('./Circle');

delete c.circumference;

c.circunferencia= function( r ) {
  return Math.PI * r * 2;
}

module.exports = c;
```

...renames the "`circumference()`" operation of module "Circle.js". Its new name is "`circunferencia()`", and this does not modify any of other operations in that module.

A detailed analysis of the actions taken in this code fragment is:

1. When the module "`Circle.js`" is imported using "`require()`", we assign its exported object to variable "`c`" in our program.
2. Later, its "`circumference()`" operation is removed. The other operations or properties are not affected. In this example, there is only another operation: "`area()`".
3. Later we add a new operation: "`circunferencia()`".
4. As a last sentence, we export the entire "`c`" object. This provides the "`area()`" and "`circunferencia()`" operations to the programs that import it.

If we store this code fragment into a "`Círculo.js`" file, we will have a new module with that name exporting those two operations.