Seminar 1 – Node.js

Student Guide (Part 2). Callbacks and Promises

1. Introduction

This guide extends the sections of the JavaScript/Node.js seminar related to callbacks and promises. The next section (Callbacks) consists of two parts. The first one describes how to implement callbacks to be invoked in asynchronous functions. The second one explains how to emulate asynchronous behaviour in Node, applying such approach to some concrete examples. The second section (Promises) explains how to implement the examples shown in the first section using promises.

2. Callbacks

2.1. Implementation of callbacks in asynchronous functions

Asynchronous programming in Node.js is based on the usage of callback functions. When a programme contains an invocation to an asynchronous function (e.g., ***f_async***) that has a callback as its last parameter, such programme 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: var 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 sentences...');
14: console.log('sqrt(2) =', Math.sqrt(2));
```

The call to the asynchronous ***readFile*** function from module ***fs*** is not blocking this programme. 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 sentences...
sqrt(2) = 1.4142...
Error: ENOENT, open '… /mydata.txt'
```

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

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: var 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 programme. 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: var fs = require("fs");
 2: var 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:       });
```

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

```
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 programme is hard to follow. In this example, additionally, there is no error management. With a minimal error management the resulting programme 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 programme is the following:

```
 1: var fs = require('fs');
 2:
 3: fs.readdir('.', function (err, files) {
 4:    var 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 the list of files in the current directory. With this list, using the ***forEach*** operator, 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.

2.2. Emulating asynchronous functions

In the examples of the previous section the programmes 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 programme:

```
 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 programme...')
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 programme. Therefore, its output is:

```
Starting the programme...
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 programme 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 programme 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:     var 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:    var m = fact(n)
14:    cb('factorial('+n+') = '+m)
15: }
16:
17: console.log('Starting the programme...')
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 programme...
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 programme 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 programme 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:    var err = eval_err(n,'fibonacci')
 7:    var 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:    var err = eval_err(n,'factorial')
15:    var 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 programme...')
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 programme is:

```
Starting the execution...
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" programme:

```
 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:       var err = eval_err(n,'fibonacci')
 8:       var 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:
15: function fact_async(n,cb) {
16:    process.nextTick(function(){
17:       var err = eval_err(n,'factorial')
18:       var 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 programme...')
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...')
```

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

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 programme 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:       var err = eval_err(n,'fibonacci')
10:       var 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:       var err = eval_err(n,'factorial')
20:       var 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 programme
47: console.log('Starting the programme...')
48:
49: var facts = []
50: for (var i=0; i<=10; i++)
51:     fact_async(i, show_fibo_back)
52: console.log('Factorials...')
53:
54: var fibs = []
55: for (var i=0; i<=20; i++)
56:     fibo_async(i, show_fact_back)
57: console.log('Fibonaccis...')
```

In this programme 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 programme is:

```
Starting the programme...
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 programme 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 programme
15:  console.log('Starting the programme...')
16:
17:  var n = 10
18:  var facts = []
19:  var fibs = []
20:
21:  for (var i=0; i<n; i++)
22:     fact_async(i, show_fact_back)
23:  console.log('Factorials...')
24:
25:  for (var 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 programme...
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 programme...
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:          var 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:             var s = facts.reduce(add)
30:             console.log('Factorial addition ['+0+'..'+(n-1)+'] =', s)
31:         }
32:     }
33: }
34:
35: // *** main programme
36: console.log('Starting the programme...')
37:
38: var n = 10
39: var facts = []
40: var fibs = []
41:
42: for (var i=0; i<n; i++)
43:     fact_async(i, show_fact_back)
44: console.log('Factorials...')
45:
46: for (var i=0; i<n; i++)
47:     fibo_async(i, show_fibo_back)
48: console.log('Fibonaccis...')
```

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

```
Starting the programme...
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.

3. Promises

As it has been seen in the seminar, the ***bluebird*** module allows programmers to implement asynchronous functions based on promises using:

- ***promisify***,
- ***defer***, or
- the promise constructor.

This section starts using these three alternatives on the example presented in Section 2.

To begin with, let us consider using ***promisify***. This is the easiest alternative to adapt an asynchronous function that uses a Node.js callback (i.e., one with a first parameter for errors and a second parameter for results).

In the following example, we use ***promisify*** with ***fibo_back2*** as its argument, declared in the ***fibo3.js*** programme. We can use this function because it complies with the ***function(err,res)*** convention[4]. Because of this, ***fact_back2*** could be also adapted using this approach.

```
 1: // ***  fibo7.js
 2:
 3: bb = require('bluebird')
 4:
 5: // *** functions
 6: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 7:
 8: function fibo_back2(n,cb) {
 9:    var err = eval_err(n,'fibonacci')
10:    var m   = err ? '': fibo(n)
11:    cb(err,'fibonacci('+n+') = '+m)
12: }
13:
14: function eval_err(n,s) {
15:    return (typeof n != 'number') ? s+'('+n+') ??? : '+n+' is not a number' : ''
16: }
17:
18: // onFulfilled handler
19: function onSuccess(res) { console.log(res) }
20:
21: // onRejected handler
22: function onCauseError(err) { console.log(err.cause);  }
23:
24: // *** main programme
25: console.log('Starting the programme...')
26:
```

[4] We cannot apply ***promisify*** to ***fibo_back1*** and ***fact_back1*** from *fibo2.js*, since their callback has a signature ***function(res)*** that does not comply with the ***promisify*** requirements.

```
27:  var elems = [10, '5', true]
28:  var fibo_promise = bb.promisify(fibo_back2)
29:  for (var i in elems)
30:     fibo_promise(elems[i]).then( onSuccess, onCauseError )
```

The output from this programme is:

```
Starting the programme...
fibonacci(10) = 89
[Error: fibonacci(5) ??? : 5 is not a number]
[Error: fibonacci(true) ??? : true is not a number]
```

This output shows that the first promise, ***fibo_promise(10)***, has been successfully resolved (executing the ***onSuccess*** handler) but the other two promises, ***fibo_promise('5')*** and ***fibo_promise(true)***, have been rejected (using the ***onCauseError*** handler).

The next programme solves this same problem using ***defer***:

```
 1:  // ***  fibo8.js
 2:
 3:  bb = require('bluebird')
 4:
 5:  // fibo – sync
 6:  function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 7:
 8:  // fibo - promise-defer version
 9:  function fibo_prom1(n) {
10:     var deferred = bb.defer()
11:     if ( typeof(n)=='number' )  deferred.resolve( fibo(n) )
12:     else                        deferred.reject( n+' is an incorrect arg' )
13:     return deferred.promise
14:  }
15:
16:  // onFulfilled handler, with closure
17:  function onSuccess(i) {
18:     return function (res) { console.log('fibonacci('+i+') =', res) }
19:  }
20:
21:  // onRejected handler
22:  function onError(err) { console.log('Error:', err); }
23:
24:  // *** main programme
25:  console.log('Starting the programme...')
26:
27:  var elems = [10, '5', true]
28:  for (var i in elems)
29:     fibo_prom1(elems[i]).then( onSuccess(elems[i]), onError )
```

Its output will be:

```
Starting the programme...
fibonacci(10) = 89
Error: 5 is an incorrect arg
Error: true is an incorrect arg
```

Function ***fibo_prom1*** creates a ***PromiseResolver*** object using ***defer*** and verifying whether the type of the received argument is correct and, depending on this, calling either the ***resolve*** or the ***reject*** method. Once this has been done, the promise is returned. On that returned promise, the ***then*** method is called, invoking the appropriate handler. In this example when the ***onSuccess*** handler is called a closure is used for passing the appropriate argument.

Finally, we solve the same problem using promise constructors. The resulting programme is:

```
 1: // *** fibo9.js
 2:
 3: bb = require('bluebird')
 4:
 5: // fibo – sync
 6: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 7:
 8: // fibo - new-promise version
 9: function fibo_prom2(n) {
10:    return new bb(function(fulfill, reject) {
11:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
12:       else                     fulfill( fibo(n) )
13:    })
14: }
15:
16: // onFulfilled handler, with closure
17: function onSuccess(i) {
18:    return function (res) { console.log('fibonacci('+i+') =', res) }
19: }
20:
21: // onRejected handler
22: function onError(err) { console.log('Error:', err); }
23:
24: // *** Main programme
25: console.log('Starting the programme...')
26: var elems = [10, '5', true]
27: for (var i in elems)
28:    fibo_prom2(elems[i]).then( onSuccess(elems[i]), onError )
```

The output from ***fibo9.js*** is the same as that from ***fibo8.js***. In function ***fibo_prom2*** 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.

Consequently, we may say the any one of these implementation alternatives (***promisify***, ***defer*** and constructor) provides the same functionality.

In the following programme we are using the alternative based on ***defer***. It tries to solve the problem of computing factorials and fibonaccis holding their results in two arrays. The resulting programme is:

```
 1: // ***  fibo10.js
 2:
 3: bb = require('bluebird')
 4:
 5: // fibo – sync
 6: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 7:
 8: // fibo - promise-defer version
 9: function fibo_prom1(n) {
10:    var deferred = bb.defer()
11:    if ( typeof(n)=='number' )  deferred.resolve( fibo(n) )
12:    else                        deferred.reject( n+' is an incorrect arg' )
13:    return deferred.promise
14: }
15:
16: // fact - sync
17: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
18:
19: // fact - promise-defer version
20: function fact_prom1(n) {
21:    var deferred = bb.defer()
22:    if ( typeof(n)=='number' )  deferred.resolve( fact(n) )
23:    else                        deferred.reject( n+' is an incorrect arg' )
24:    return deferred.promise
25: }
26:
27: // onFulfilled handler, with closure
28: function onSuccess(s, x, i) {
29:    return function (res) {
30:       if ( x!=null ) x[i] = res
31:       console.log(s+'('+i+') =', res)
32:    }
33: }
34:
35: // onRejected handler
36: function onError(err) { console.log('Error:', err); }
37:
38: // *** Main programme
39: console.log('Starting the programme...')
40:
41: var n = 10
42: var fibs = []
43: var fibsPromises = []
44: var facts = []
45: var factsPromises = []
46:
47: // Generate the promises.
```

```
48: for (var i=0; i<n; i++) {
49:    fibsPromises[i] = fibo_prom1(i)
50:    factsPromises[i] = fact_prom1(i)
51: }
52:
53: // Show the results.
54: for (var i=0; i<n; i++) {
55:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
56:    factsPromises[i].then( onSuccess('factorial', facts, i), onError )
57: }
```

Its output is:

```
Starting the programme...
fibonacci(0) = 1
factorial(0) = 1
…
fibonacci(9) = 55
factorial(9) = 362880
```

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

```
 1: // *** fibo11.js
 2:
 3: bb = require('bluebird')
 4:
 5: // fibo – sync
 6: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 7:
 8: // fibo - promise-defer version
 9: function fibo_prom1(n) {
10:    var deferred = bb.defer()
11:    if ( typeof(n)=='number' )  deferred.resolve( fibo(n) )
12:    else                        deferred.reject( n+' is an incorrect arg' )
13:    return deferred.promise
14: }
15:
16: // fact - sync
17: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
18:
19: // fact - promise-defer version
20: function fact_prom1(n) {
21:    var deferred = bb.defer()
22:    if ( typeof(n)=='number' )  deferred.resolve( fact(n) )
23:    else                        deferred.reject( n+' is an incorrect arg' )
24:    return deferred.promise
25: }
26:
27: // onFulfilled handler, with closure
```

```
28: function onSuccess(s, x, i) {
29:    return function (res) {
30:       if ( x!=null ) x[i] = res
31:       console.log(s+'('+i+') =', res)
32:    }
33: }
34:
35: // onRejected handler
36: function onError(err) { console.log('Error:', err); }
37:
38: // onFulfilled handler, for array of promises
39: function addAll(z, x) {
40:    return function () {
41:       var s = 0
42:       for (var i in x) s += x[i]
43:       console.log(z, '=', x, '; addition =', s)
44:    }
45: }
46:
47: // onRejected handler, for array of promises
48: function showFinalError() {
49:    console.log('Something wrong has happened...')
50: }
51:
52: // *** Main programme
53: console.log('Starting the programme...')
54:
55: var n = 10
56: var fibs = []
57: var fibsPromises = []
58: var facts = []
59: var factsPromises = []
60:
61: // Generate the promises.
62: for (var i=0; i<n; i++) {
63:    fibsPromises[i] = fibo_prom1(i)
64:    factsPromises[i] = fact_prom1(i)
65: }
66:
67: // Show the results.
68: for (var i=0; i<n; i++) {
69:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
70:    factsPromises[i].then( onSuccess('factorial', facts, i), onError )
71: }
72:
73: // Show the summary.
74: bb.all(fibsPromises).then( addAll('fibs', fibs), showFinalError )
75: bb.all(factsPromises).then( addAll('facts', facts) )
```

The output from ***fibo11.js*** is:

```
Starting the programme...
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 programmes ***fibo10.js*** and ***fibo11.js***, functions ***fibo_prom1*** and ***fact_prom1*** are extremely similar. In the next programme we provide an equivalent but shorter implementation:

```
 1: // ***  fibo12.js
 2:
 3: bb = require('bluebird')
 4:
 5: // fibo – sync
 6: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 7:
 8: // fact - sync
 9: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
10:
11: // "func" - promise-defer version
12: function func_prom1(f,n) {
13:    var deferred = bb.defer()
14:    if ( typeof(n)=='number' ) deferred.resolve( f(n) )
15:    else                       deferred.reject(n+' is an incorrect arg')
16:    return deferred.promise
17: }
18:
19: // onFulfilled handler, with closure
20: function onSuccess(s, x, i) {
21:    return function (res) {
22:       if ( x!=null ) x[i] = res
23:       console.log(s+'('+i+') =', res)
24:    }
25: }
26:
27: // onRejected handler
28: function onError(err) { console.log('Error:', err); }
29:
30: // onFulfilled handler, for array of promises
```

```
31: function addAll(z, x) {
32:     return function () {
33:         var s = 0
34:         for (var i in x) s += x[i]
35:         console.log(z, '=', x, '; addition =', s)
36:     }
37: }
38:
39: // onRejected handler, for array of promises
40: function showFinalError() {
41:     console.log(Something wrong has happened...')
42: }
43:
44: // *** Main programme
45: console.log('Starting the programme...')
46:
47: var n = 10
48: var fibs = []
49: var fibsPromises = []
50: var facts = []
51: var factsPromises = []
52:
53: // Generate the promises.
54: for (var i=0; i<n; i++) {
55:     fibsPromises[i]  = func_prom1(fibo, i)
56:     factsPromises[i] = func_prom1(fact, i)
57: }
58:
59: // Show the results.
60: for (var i=0; i<n; i++) {
61:     fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
62:     factsPromises[i].then( onSuccess('factorial', facts, i), onError )
63: }
64:
65: // Show the summary.
66: bb.all(fibsPromises).then( sumAll('fibs', fibs), showFinalError )
67: bb.all(factsPromises).then( sumAll('facts', facts) )
```