Tema 2 – JavaScript i Node.js

Guia d'estudi

1. Introducció

La presentació d'aquest tema té tres seccions principals:

1. Una introducció que explica per què JavaScript i NodeJS s'han triat com el llenguatge de programació i l'intèrpret, respectivament, a utilitzar en TSR.
2. Una segona secció sobre el llenguatge JavaScript.
3. Una tercera secció sobre l'intèrpret node.

El fitxer amb la presentació ofereix un breu resum per a cadascun d'aquests apartats. Aquesta guia hauria d'ampliar els resums. No obstant això, el format habitual d'aquestes guies no és suficient per a descriure amb la profunditat necessària els aspectes més destacats dels dos últims apartats. A més, hi ha excel·lents publicacions sobre aquesta matèria. Per això, ens limitarem a seleccionar els apartats més rellevants d'algunes de les publicacions existents, convidant al lector a llegir-les amb cura. Posteriorment inclourem altres seccions on s'explicaran alguns conceptes importants.

2. JavaScript

La nostra presentació sobre JavaScript esmenta aquesta possible estructura per al seu contingut:

1. Característiques principals.
2. Alternatives per a l'execució de codi. Aquesta secció és autocontinguda. No cal ampliar-la.
3. Sintaxi.
4. Valors primitius i compostos.
5. Variables. Àmbit. Clausures.
6. Operadors.
7. Sentències.
8. Funcions.
9. Vectors.
10. Programació funcional.
11. Objectes.
12. Serialització. JSON.
13. "Callbacks". Execució asincrònica. Promeses.
14. Esdeveniments.

Aquesta llista ha d'interpretar-se solament com un guió. El seu contingut pot estendre's en aquest URL: https://www.tutorialspoint.com/es6/index.htm. Allí, Tutorials Point facilita un tutorial sobre ECMAScript 6 (l'edició de JavaScript assumida en les eines utilitzades en aquesta assignatura). Hi ha també una guia ràpida que resumeix el contingut del tutorial: https://www.tutorialspoint.com/es6/es6_quick_guide.htm.

Del tutorial, n'hi ha prou amb utilitzar aquestes parts (n'hi ha moltes més):

- Overview: Cobreix la secció 1 de la nostra llista.

- Environment: Descriu com instal·lar NodeJS i Visual Studio Code en diversos sistemes. Tots dos s'utilitzen en aquesta assignatura.
- Syntax: Cobreix les seccions 3 i 5 de la nostra llista.
- Variables: Cobreix les seccions 4 i 5 en la nostra llista.
- Operators: Cobreix la secció 6 de la llista.
- Decison making: Cobreix parcialment la secció 7.
- Loops: També cobreix parcialment la secció 7.
- Functions: Cobreix la secció 8 (i també parcialment la 5) de la llista.
- Arrays: Cobreix la secció 9.
- Classes: Cobreix, de manera avançada, la secció 11.
- Promises: Cobreix, de manera succinta, les seccions 13 i 14 de la llista.
- Error handling: Encara que aquesta part no es discuteix directament en la presentació del tema, el seu estudi és rellevant. Recomanem la seua lectura, encara que l'apartat sobre onerror() no pot aplicar-se a NodeJS, perquè només té sentit en els navegadors web.

La resta del tutorial cobreix parts de JavaScript que només s'utilitzaran als navegadors web. El seu contingut no és rellevant per a aquesta assignatura.

Observe's que el tutorial no cobreix completament tot el contingut del tema. Per exemple, les seccions 5 (Clausures), 10 i 12 no han sigut cobertes. La secció 10 descriu una part de JavaScript que no és important en aquesta assignatura i la secció 12 pot entendre's quan es consulta algun exemple. D'altra banda, les clausures sí que són importants. Per tant, necessitem alguna font addicional per a entendre millor què és una clausura en JavaScript.

Una primera solució per a aquest problema és aquesta descripció de les clausures, disponible en el lloc w3schools.com: https://www.w3schools.com/js/js_function_closures.asp.

Una altra solució consisteix a llegir algun llibre sobre JavaScript. N'hi ha molts. Un bon exemple és Eloquent JavaScript, 3rd edition, de Marijn Haverbeke. Inclou una introducció breu i 21 capítols. D'ells, els recomanables per a la nostra assignatura són:

- Values, types and operators (Capítol 1): Cobreix les seccions 1, 3, 4 i 6 (aquesta última, parcialment) de la llista.
- Program structure (Capítol 2): Cobreix les seccions 5, 6 i 7 de la llista.
- Functions (Capítol 3): Cobreix les seccions 5 i 8 de la llista.
- Data structures: Objects and arrays (Capítol 4): Cobreix les seccions 9, 11 i 12 de la llista.
- Higher-order functions (Capítol 5): Cobreix en profunditat el que es resumeix en la secció 10 de la llista. Ha de considerar-se una lectura opcional, perquè la secció 10 s'inclou en el Tema 2 per completitud. No és una part central en la nostra assignatura.
- Asynchronous programming (Capítol 11): Cobreix en profunditat les seccions 13 i 14 de la llista.

A més, el llibre facilita múltiples exemples breus de programes i també exercicis en cada capítol.

Per a ampliar aquestes referències externes i el llibre, donarem en les següents seccions algunes descripcions d'alguns elements que no s'han presentat fins ara amb prou profunditat. Els seus nombres de secció es corresponen amb els utilitzats a la presentació.

2.4.3. Funcions. Clausures

En alguns casos resulta útil declarar una funció dins d'una altra. En aquestes situacions pot ser que resulte interessant retornar la funció interna. Aquesta funció interna seguirà tenint accés a les variables utilitzades en la funció que l'engloba, així com als seus arguments. Això segueix sent vàlid encara que la invocació que la va crear haja acabat. El fet que es mantinga el context extern es coneix com a "clausura".

Vegem un exemple en el qual podrem utilitzar clausures. La funció "log()" del mòdul Math de JavaScript retorna el logaritme neperià del valor rebut com a argument. Utilitzarem clausures per a implantar una funció que en "construïsca" i retorne una altra capaç de calcular el logaritme en la base que especifiquem. Per a fer això aprofitarem aquesta propietat dels logaritmes:

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

El codi a utilitzar seria:

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

Com pot observar-se, la funció "constructora" es diu "logBase()" i utilitza el paràmetre "b" per a especificar la base a utilitzar. La seua única sentència s'encarrega de retornar una funció. Aquesta funció és una funció anònima que rep com a únic paràmetre un valor "x". El codi de la funció retornada s'encarrega d'aplicar la propietat que hem enunciat a dalt, recordant l'argument rebut en la funció constructora.

Prenent aquest codi com a punt de partida, podrem generar les funcions necessàries per a calcular logaritmes en diferents bases. En el següent fragment utilitzem tant la base 2 com la 8. Observe's que resulta senzill seguir el codi resultant:

```
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
```

Pot ser que necessitem usar alguna variable de la funció que proporcione el context de la clausura, en lloc d'utilitzar únicament algun dels seus paràmetres. En aquest cas ha de recordar-se que l'accés a les variables es fa per referència i això pot ser problemàtic.

Vegem-ne un exemple. En el següent fragment de codi es desitja desenvolupar una funció que retorne un vector que conté tres funcions. Cadascuna d'elles retornarà el nom i població de cadascun dels tres països més poblats del nostre planeta. Aquesta solució no funciona:

```
 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();
```

La crida efectuada en la línia 16 executa el codi de la funció “populations()”. Seria d'esperar que el vector “ps” continguera les tres funcions sol·licitades i que les crides efectuades en les línies 22 a 24 mostraren el nom i població dels tres països més poblats. No obstant això, l'eixida obtinguda és:

```
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
```

Això es justifica perquè quan cridem a les funcions “first()”, “second()” i “third()” el valor de la variable “i” dins de la funció “populations()” és 3. Poc importa que quan es van crear les tres funcions valguera 0, 1 i 2, respectivament. Seguim utilitzant la variable “i” quan s'invoquen les funcions creades (és com si s'estiguera passant per referència) i el seu valor actual és 3. Per això, l'eixida obtinguda no és correcta.

Com podem resoldre-ho? Utilitzant clausures a l'hora d'accedir al valor de "i". El nostre objectiu és que la funció que està sent declarada entre les línies 7 i 12 recorde el valor que tenia "i" en aquella iteració del bucle. No ens interessa el valor que tindrà la "i" quan invoquem a les funcions que retornarem en el vector sinó el que té mentres itera.

La solució es mostra a continuació:

```
 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();
```

Els canvis s'han ressaltat en negreta. Necessitem una altra funció que la tanque i que reba com a únic argument el valor actual de la variable "i". Per a fer això, la nova funció rep un paràmetre "x" i la funció que retornarem accedirà a "x" en lloc d'accedir a "i". En cada iteració del bucle s'invoca la funció que engloba a la que anàvem a retornar, passant com a argument el valor actual de la variable "i" (vegeu la línia 14). L'única cosa que fa la funció contenidora és retornar la funció que en la versió anterior estava llistada entre les línies 7 i 12.

Amb aquest "truc" aconseguim que l'eixida proporcionada pel programa siga:

```
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
```

Aquesta era l'eixida a obtenir. Amb això queda demostrat que ha de portar-se certa cura a l'hora d'utilitzar bucles per a generar funcions utilitzant clausures.

ECMAScript 6 va introduir una segona manera de resoldre aquest problema, proporcionant una altra manera de declarar variables. Tradicionalment, s'havia utilitzat la paraula reservada "var" per a fer això. ECMAScript 6 va introduir la paraula reservada "let". Si declarem amb "let" la variable índex d'un bucle, cada vegada que s'utilitze la variable, el seu valor es passa directament a la sentència que l'use, en comptes de passar una referència a la variable. Així, en comptes d'usar clausures, hauria hi hagut prou amb afegir "let" allí on es declarava (implícitament) la variable "i" del bucle vist anteriorment. Així, la versió resultant seria:

```
 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();
```

I la seua eixida serà idèntica a la versió basada en clausures. És a dir, seria correcta.

2.5. Callbacks

Implementació de callbacks per a funcions asincròniques

La programació asincrònica en Node.js es basa en l'ús de les funcions callback. Quan un programa conté la invocació d'una funció asincrònica, per exemple ***f_async***, que porte un callback com a últim argument, aquest programa no es bloqueja fins que ***f_async*** concloga, sinó que continuarà executant les instruccions següents que tinga. Mentrestant, la funció asincrònica s'anirà executant i, a la seua fi, en un torn posterior, s'executarà el seu callback.

En "Control Flow in Node"[1], Tim Caswell presenta alguns exemples interessants per a comprendre el funcionament de les funcions asincròniques i els callbacks. El següent codi, amb lleus modificacions, està pres d'un exemple de Caswell.

```
 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('executant altres instruccions');
14: console.log('arrel(2) =', Math.sqrt(2));
```

La crida a la funció asincrònica ***readFile*** del mòdul ***fs*** no bloqueja aquest programa. Les instruccions de les línies 13 i 14 s'executaran abans que es complete ***readFile*** i, per tant, abans de la funció anònima de callback que rep. Si l'intent de lectura del fitxer 'mydata.txt' fallara, en el callback s'executa la instrucció de la línia 6. En tal cas, l'eixida d'aquest programa seria:

```
executant altres instruccions
arrel(2) = 1.4142 ...
Error: ENOENT, open '… /mydata.txt'
```

La funció ***readFIle*** rep un nom de fitxer i "retorna" el seu contingut. En sentit estricte, no retorna res, sinó que ho passa com a argument a la seua funció callback. El segon argument del callback (*buffer* en l'exemple anterior) rep el contingut del fitxer quan l'operació de lectura es duu a terme correctament. El primer argument del callback (*err* en l'exemple) rep el missatge d'error generat quan la lectura falla per alguna causa (com ha succeït en l'anterior exemple d'execució).

Definir la funció de callback com a anònima és usual si només s'utilitza una vegada, com en l'exemple anterior. Però, en cas que s'use més vegades, el recomanable és definir-la amb nom. El següent exemple de Tim Caswell, modificació de l'anterior, il·lustra aquest cas.

```
 1: const fs = require('fs');
 2:
 3: function callback(err, buffer) {
 4:   if (err) {
 5:     // Handle error
```

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

```
 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);
```

La implementació de funcions asincròniques mitjançant callbacks permet la implantació sense un altre límit que les necessitats del programador. En "Accessing the File System in Node.js"[2], Colin Ihrig presenta un exemple d'implantació bastant profunda de callbacks:

```
 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:         let buffer = new Buffer(stats.size);
 9:
10:         fs.read(fd, buffer, 0, buffer.length, null, function(error, bytesRead, buffer) {
11:           let data = buffer.toString("utf8", 0, buffer.length);
12:
13:           console.log(data);
14:           fs.close(fd);
15:         });
16:       });
17:     });
18:   }
19: });
```

En aquest exemple, es llig el contingut d'un fitxer amb l'ajuda d'un buffer. El callback de la funció asincrònica ***exists*** del mòdul ***fs*** conté una crida a una altra funció asincrònica, ***stats***, que el seu callback, al seu torn, invoca a una altra funció asincrònica, ***open***, que el seu callback invoca a la funció asincrònica ***read***, la qual també té un callback.

Certament hi ha altres formes de llegir un fitxer: usar ***readFile***, com en els exemples proposats per Caswell, o usar versions sincròniques de les funcions del mòdul ***fs***. La fi de presentar aquest últim programa és mostrar les possibilitats d'implantació de callbacks... i els seus inconvenients. Un alt nivell d'implantació, com s'aprecia, dificulta la lectura del codi. En l'exemple, a més, no es duu a terme cap gestió d'errors. Si això es fera, el codi encara seria

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

més difícil d'interpretar, i de seguir la seua execució. En aquestes situacions, es fa molt convenient l'ús de les promeses com a alternativa en la programació asincrònica.

El niat de callbacks és una manera d'assegurar que les successives operacions (des de la més externa a la més interna) s'executen en seqüència. Però també poden plantejar-se situacions en les quals convinga agrupar un conjunt d'operacions paral·leles. Un exemple el proporciona Tim Caswell (en les pàgines web abans citades): la lectura de tots els fitxers existents en un directori. El codi, amb lleus modificacions, és el següent:

```
 1: const fs = require('fs');
 2:
 3: fs.readdir('.', function (err, files) {
 4:    let count = files.length,
 5:    results = {};
 6:    files.forEach(function (filename) {
 7:      fs.readFile(filename, function (data) {
 8:        console.log(filename, 'has been read');
 9:        results[filename] = data;
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: });
```

El callback de la funció asincrònica ***readdir*** rep la llista dels fitxers existents en el directori actual. Amb aquesta llista, i mitjançant l'operador ***forEach***, s'inicia la lectura de cadascun dels fitxers, mitjançant la funció asincrònica ***readFile***. Així, totes les operacions de lectura s'executen en paral·lel i poden acabar en qualsevol ordre. Conforme conclouen, s'executen els seus respectius callbacks. En aquests, s'actualitza una variable comptador, ***count***, la qual cosa permet detectar quan totes les lectures han acabat per si es vol fer alguna operació posterior.

Implementació de funcions asincròniques

En els exemples de la secció anterior, s'han invocat funcions asincròniques ja existents, funcions del mòdul ***fs***. Altres mòduls estàndard de Node.js proporcionen altres funcions asincròniques. En aquests casos, el programador només necessita invocar-les, proporcionant els arguments adequats (el que implica implementar la funció callback que serà el seu últim argument). En aquesta secció es presenta com implementar una funció qualsevol, inicialment sincrònica, perquè tinga un comportament asincrònic.

Considere's el següent codi:

```
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('Iniciant execució...')
12: console.log('fibo(40) =', fibo(40))
13: console.log('llançat càlcul fibonacci...')
14: console.log('fact(10) =', fact(10))
15: console.log('llançat càlcul factorial...')
```

Les dues funcions ací definides (la funció ***fibo*** per a calcular el terme enèsim de la successió de Fibonacci i la funció ***fact*** per a calcular el factorial del nombre rebut com a argument) són sincròniques. Per això, les seues invocacions, en les línies 12 i 14 d'aquest codi, bloquegen el programa. L'eixida obtinguda és:

```
Iniciant execució...
fibo(40) = 165580141
llançat càlcul fibonacci...
fact(10) = 3628800
llançat càlcul factorial...
```

A més, la segona línia d'aquesta eixida tarda alguns segons a mostrar-se, donat el cost computacional de la funció ***fibo***. Seria desitjable disposar de versions asincròniques de les funcions, de manera que s'invocaren sense bloquejar el programa en les línies 12 i 14, i els missatges de les línies 13 i 15 es mostraren immediatament (i s'executaren altres instruccions posteriors, si n'hi haguera alguna).

Una forma senzilla d'implantar aquestes versions asincròniques de les funcions consisteix a afegir ***console.log*** com a funció callback i usar la funció ***setTimeout*** per a organitzar la invocació de les funcions (i passar-les així a la cua d'esdeveniments amb uns temporitzadors adequats). Amb aquestes modificacions, es tindria el següent programa:

```
 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('Iniciant execució...')
18: setTimeout( function(){
19:    fibo_back1(40, console.log)
20: }, 2000 )
21: console.log('llançat càlcul fibonacci...')
22: setTimeout( function(){
23:    fact_back1(10, console.log)
24: }, 1000 )
25: console.log('llançat càlcul factorial...')
```

L'eixida obtinguda en executar aquesta segona versió del programa és:

```
Iniciant execució...
llançat càlcul fibonacci...
llançat càlcul factorial...
factorial(10) = 3628800
fibonacci(40) = 165580141
```

A més, només l'aparició de l'última línia d'aquesta eixida presenta un retard apreciable.

Encara que el codi anterior és correcte i funciona com s'esperava, la funció usada com callback, ***console.log***, no s'ajusta al conveni habitual. En Node.js, es recomana que les funciones callback reben, com a primer argument, la informació d'error (si la funció asincrònica fallara) i, com a segon argument, el resultat proporcionat (quan la funció asincrònica acaba correctament).

Si es modifica l'anterior programa tenint en compte aquest conveni, es tindria el següent:

```
 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('Iniciant execució...')
30: setTimeout( function(){
31:    fibo_back2(40, show_back)
32:    fibo_back2('pep', show_back)
33: }, 2000 )
34: console.log('llançat càlcul fibonacci...')
35: setTimeout( function(){
36:    fact_back2(10, show_back)
37:    fact_back2('ana', show_back)
38: }, 1000 )
39: console.log('llançat càlcul factorial...')
```

Ara, les funcions asincròniques, ***fibo_back2*** i ***fact_back2***, reben com callback la funció ***show_back*** (implementada en les línies 19 a 22). S'ha afegit una altra funció auxiliar, ***eval_err*** (en les línies 24 a 27), que genera el missatge d'error quan correspon (s'ha suposat que les situacions d'error es donen quan les funcions s'invoquen amb un argument no numèric).

L'eixida obtinguda en executar el programa ***fibo3.js*** és:

```
Iniciant execució...
llançat càlcul fibonacci...
llançat càlcul factorial...
factorial(10) = 3628800
factorial(ana) ??? : ana is not a number
fibonacci(40) = 165580141
fibonacci(pep) ??? : pep is not a number
```

Les funcions “asincròniques” definides fins al moment ho són només formalment ja que reben un callback com a argument, però no ho són en el seu comportament: l'asincronia es deu a la seua invocació com a argument de la funció ***setTimeout***.

Una forma de generar versions realment asincròniques consisteix a utilitzar la funció ***nextTick*** de ***process***[3]. Aquesta funció permet retardar l'execució d'una acció fins a la següent iteració del bucle d'esdeveniments. Considere's la següent modificació del programa tenint en compte això:

```
1: // *** fibo4.js
2:
3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
```

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

```
 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:
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('Iniciant execució...')
34: fact_async(10, show_back)
35: fact_async('ana', show_back)
36: console.log('llançats càlcul factorial...')
37: fibo_async(40, show_back)
38: fibo_async('pep', show_back)
39: console.log('llançats càlcul fibonacci...')
```

L'eixida obtinguda en executar el programa ***fibo4.js*** és la mateixa que en executar ***fibo3.js***. Observe's, en les línies 33 a 39, que no es requereix l'ús de ***setTimeout***. Observe's també que s'ha modificat l'ordre de les instruccions: les invocacions a ***fact_async*** (línies 34-35) es fan abans que les invocacions a ***fibo_async*** (línies 37-38), atès que ara no s'estableix un temporitzador explícit a cada funció, i es vol seguir mostrant abans el resultat del càlcul de factorial.

Supose's ara que es desitjara calcular un cert nombre de valors resultat d'ambdues funcions i que aquests valors s'emmagatzemaren en sengles vectors. Un programa que fera això seria el següent:

```
1: // ***  fibo5.js
2:
```

```
 3: // *** funcions
 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: // ***programa principal
47: console.log('Iniciant execució...')
48:
49: let facts = []
50: for (var i=0; i<=10; i++)
51:    fact_async(i, show_fibo_back)
52: console.log('llançat càlcul de factorials...')
53:
54: let fibs = []
```

```
55: for (var i=0; i<=20; i++)
56:     fibo_async(i, show_fact_back)
57: console.log('llançat càlcul de fibonaccis...')
```

En aquest programa, s'ha modificat la implementació del callback. Ara hi ha dues funcions callback (una per a cada funció asincrònica), que emmagatzemen el resultat del càlcul en la posició indicada de l'array corresponent, a més de mostrar-ho en consola.

L'eixida obtinguda en executar el programa ***fibo5.js*** (no es mostra l'eixida completa, les línies amb punts suspensius representen l'eixida no reproduïda explícitament) és:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de 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
```

A continuació, considere's ampliar el programa perquè proporcione la suma de tots els factorials calculats i la suma de tots els termes de Fibonacci calculats. Podria pensar-se que una possible solució seria la següent:

```
 1: // *** fibo6a.js
 2:
 3: // *** funcions:  mateixa implementació que en el programa 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 suma(a,b) { return a+b }
13:
14: // *** programa principal
15: console.log('Iniciant execució...')
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('llançat càlcul de factorials...')
24:
25: for (let i=0; i<n; i++)
26:     fibo_async(i, show_fibo_back)
27: console.log('llançat càlcul de fibonaccis...')
28:
29: console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
30: console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
```

No obstant això, l'eixida obtinguda en executar ***fibo6a.js*** és:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
…
… /fibo6a.js:29
   console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))

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

Aquest resultat (error en intentar aplicar la funció ***reduce*** a un vector buit, no inicialitzat) es produeix perquè les línies 29 i 30 s'executen abans que les funcions asincròniques i les seues callbacks.

Una solució senzilla a aquest problema és reorganitzar les instruccions de les línies 29 i 30 amb una funció que passe la seua execució a la cua de torns de Node. Per exemple:

```
process.nextTick( function(){
   console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
   console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
});
```

O també mitjançant:

```
setTimeout( function(){
   console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
   console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
}, 1 );
```

Qualsevol d'aquestes dues solucions, proporciona la següent eixida en executar el programa:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
```

```
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
suma factorials [0..9] = 409114
suma fibonaccis [0..9] = 143
```

Una altra possibilitat és modificar els callbacks de les funcions asincròniques perquè, una vegada calculades totes les components del vector, calculen les sumes. Aquesta solució seria la següent:

```
 1: // ***  fibo6b.js
 2:
 3: // *** funcions:  mateixa implementació que en fibo6a.js, excepte 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 suma(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(suma)
18:          console.log('suma fibonaccis ['+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(suma)
30:          console.log('suma factorials ['+0+'..'+(n-1)+'] =', s)
31:       }
32:    }
33: }
34:
35: // *** programa principal
```

```
36: console.log('Iniciant execució...')
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('llançat càlcul de factorials...')
45:
46: for (let i=0; i<n; i++)
47:    fibo_async(i, show_fibo_back)
48: console.log('llançat càlcul de fibonaccis...')
```

L'eixida obtinguda en executar ***fibo6b.js*** és:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
suma factorials [0..9] = 409114
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
suma fibonaccis [0..9] = 143
```

Aquesta eixida és correcta i només difereix (respecte a les solucions d'organitzar les crides a ***reduce*** amb ***setTimeout*** o ***nextTick***) en l'ordre en el qual es presenten els resultats en la consola.

2.5.2. Promeses

Hi ha una altra manera de construir execucions asincròniques en JavaScript: mitjançant promeses. Encara que durant els darrers anys ha hi hagut múltiples propostes per a introduir promeses en JavaScript, suportades per diferents mòduls que podien incloure's en els programes que escrivíem, l'estàndard ECMAscript 6 només ha acceptat la variant basada en constructors, proporcionant l'objecte Promise per a aquesta finalitat.

Aquesta secció aplica la variant estàndard a l'exemple de càlcul dels termes de la successió de Fibonacci, presentat en la secció anterior.

El programa a utilitzar és el següent:

```
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 )
```

La funció ***fibo_promise*** construeix i torna un objecte promesa. Aquesta funció té dos paràmetres. El primer és el callback que cal utilitzar quan la promesa genere un resultat correcte. Com es pot veure en aquest exemple, aquest callback rep com a argument el resultat de la invocació a la funció que es pretén convertir en promesa. En l'exemple ha sigut la funció fibo(). El segon paràmetre és el callback que s'invocarà quan la promesa genere un error o excepció.

Tant la segona com la tercera component del vector "elems" generen un error en invocar fibo(), mostrant-lo de seguida. D'altra banda, fibo(25) necessita prou temps per completar els seus càlculs. Per això, en l'eixida del programa anterior s'observarà primer el missatge d'error per a la cadena "5", seguit per un missatge similar per al valor booleà "true". Per a concloure es mostrarà el resultat de fibo(25).

En el següent programa es considera de nou el problema de calcular factorials i termes de Fibonacci emmagatzemant-los en vectors. Una implementació usant promeses és la següent:

```
 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: }
```

L'eixida obtinguda en executar el programa ***fibo8.js*** és:

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

Considere's ara el mateix problema, i a més el problema de sumar les components dels vectors resultat. Un programa correcte usant promeses és el següent:

```
 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) )
```

L'eixida obtinguda en executar el programa ***fibo9.js*** és:

```
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 ] ; sum = 143
facts = [ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880 ] ; sum = 409114
```

Com s'acaba de comprovar, l'ús de vectors de promeses resol satisfactòriament el problema de calcular la suma de components d'un vector. El manejador d'èxit associat al resultat del vector de promeses, funció ***sumAll***, s'executa solament després que totes les promeses han sigut

satisfetes, és a dir, després que s'hagen calculat tots els valors que es desitjava emmagatzemar al vector.

L'encadenament de les promeses usant la funció ***then*** facilita al programador establir la seqüència adequada d'accions. Com es pot apreciar, aquesta seqüència és més fàcil de llegir en el codi (respecte a la solució basada en callbacks).

En els programes ***fibo8.js*** i ***fibo9.js***, es pot apreciar l'extrema similitud de les funcions ***fibo_promise*** i ***fact_promise***. En el següent programa es proporciona una altra implementació equivalent, amb menys codi:

```
 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

La documentació oficial sobre NodeJS (https://nodejs.org/en/docs/) facilita tota la informació necessària en aquest segon bloc del tema. Pot consultar-se si és necessari. Podem distingir dues classes de documentació:

- Una referència per a la seua API. Des de la pàgina inicial de documentació es pot accedir a la informació per a l'actual última edició i per a la LTS. No obstant això, hi haurà una versió més antiga als laboratoris. De fet, l'edició que primer va cobrir l'estàndard ECMAScript 6 va ser la NodeJS 6.x.x. La seua referència està ací: https://nodejs.org/docs/latest-v6.x/api/. No és necessari un coneixement exhaustiu d'aquest conjunt d'operacions.

Guies complementàries. Disponibles en: https://nodejs.org/en/docs/guides/. Hi ha tres tipus de guies: (1) general, (2) conceptes importants i (3) relatives a mòduls. El segon tipus descriu alguns conceptes centrals del "runtime" de NodeJS, tals com els temporitzadors, el bucle de gestió d'esdeveniments, les diferències entre operacions bloquejants i no bloquejants, etc. Aquestes guies del segon tipus són una lectura recomanable.

A més d'aquestes referències, les seccions següents descriuen com NodeJS gestiona la seua cua interna d'esdeveniments i com fa la gestió de mòduls, ja que la documentació sobre aquests aspectes no és gens fàcil de trobar.

3.2. Asincronia. Cua d'esdeveniments

El “runtime” de NodeJS té una cua d'esdeveniments (també coneguda com a “cua de torns”). Això es deu al fet que JavaScript és un llenguatge que no suporta múltiples fils d'execució. Per això, quan en alguna part d'un programa es genere alguna altra activitat, aquesta activitat es modela com un nou “esdeveniment” i és afegida a la cua d'esdeveniments o torns. Els torns se serveixen en ordre FIFO. Per a iniciar el servei d'un nou torn haurà d'haver acabat l'execució del torn anterior.

Analitzem aquest exemple:

```
 1: function fibo(n) {
 2:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1);
 3: }
 4: console.log("Iniciant execució...");
 5: // Esperar 10 ms per a escriure un missatge.
 6: // Implica generar nou esdeveniment.
 7: setTimeout( function() {
 8:   console.log( "M1: Vull escriure açò..." );
 9: }, 10 );
10: // Més de 5 segs. per a executar fibo(40)
11: let j = fibo(40);
12: function altreMissatge(m,o) {
13:   console.log( m + ": El resultat és: " + o );
14: }
15: // M2 s'escriu abans que M1 perquè el
16: // fil "principal" no se suspèn...
17: altreMissatge("M2",j);
18: // M3 ja s'escriu després de M1.
19: // És un altre esdeveniment (a executar després d'1 ms).
20: setTimeout( function() {
21:  altreMissatge("M3",j);
22: }, 1 );
```

La funció setTimeout() s'utilitza en JavaScript per a programar l'execució de la funció rebuda en el seu primer paràmetre al cap del nombre de mil·lisegons especificats en el seu segon paràmetre.

Seguim una traça d'aquest programa:

- L'execució comença a les línies 1 a 3, on es declara una funció recursiva “fibo()” que rebrà un argument enter i calcularà el nombre de Fibonacci associat al seu valor.
- Arribem ara a la línia 4 que escriu el missatge “Iniciant execució...” en la pantalla.
- En les línies 7 a 9 s'utilitza setTimeout() per a programar l'escriptura d'un altre missatge (“M1: Vull escriure açò...”) dins de 10 ms. De moment això no té cap efecte.
- Arribem a la línia 11 on s'invoca la funció “fibo()” amb el valor 40. La seua execució necessitarà alguns segons. Durant aquest termini hauran transcorregut els 10 ms esmentats en el punt anterior. Això implica que la funció que ha d'escriure el missatge

M1 per pantalla està ja situada en la cua d'esdeveniments, esperant el seu torn. Aquest torn no començarà mentre no acabe el fil principal.

- El fil principal arribarà poc després a les línies 12 a 14, on es declara la funció altreMissatge() que utilitzarem posteriorment per a escriure el nombre de Fibonacci obtingut.
- Amb això arribem ja a la línia 17. En ella s'escriu per primera vegada el resultat. En pantalla es mostrarà el següent: "M2: El resultat és: 165580141"
- I finalment arribem a les línies 19 a 21, on es programa novament la invocació d'altreMissatge() (missatge M3) després d'una espera d'1 ms. Amb això acaba l'execució del fil principal.
- En aquest moment en la cua d'esdeveniments solament trobem un torn actiu. Iniciem la seua execució. En aquest torn s'imprimirà per pantalla aquest missatge: "M1: Vull escriure açò..." Durant aquesta escriptura haurà finalitzat l'espera d'1 ms iniciada en el punt anterior. Amb això, es diposita un nou context en la cua d'esdeveniments, que vol escriure el missatge M3.
- Una vegada mostrat el missatge M1, el primer torn que havíem creat finalitza. Es revisa la cua d'esdeveniments i s'observa que en ella existeix encara un altre context. S'inicia la seua execució, que mostra per pantalla el missatge: "M3: El resultat és: 165580141".
- Amb això acaba l'execució del programa. L'eixida completa proporcionada és:

```
Iniciant execució...
M2: El resultat és: 165580141
M1: Vull escriure açò...
M3: El resultat és: 165580141
```

Com pot observar-se, les dues primeres línies han sigut impreses pel fil principal. Per la seua banda, les línies que comencen amb M1 i M3 han sigut impreses utilitzant dos esdeveniments. Encara que el primer dels esdeveniments va ser generat molt prompte (mentre s'estava executant la funció fibo(); prou abans que es volguera escriure el missatge M2) el seu resultat no va poder mostrar-se llavors per pantalla. Ha hagut d'acabar abans el fil principal.

Revisem ara aquest altre exemple:

```
 1: /*************************************************************/
 2: /* Events1.js                                                */
 3: /*************************************************************/
 4: const ev = require('events');
 5: let emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: const i1 = "print";
 8: const i2 = "read";
 9: // Auxiliary variables.
10: let num1 = 0;
11: let num2 = 0;
12:
13: // Listener functions are registered in
```

```
14: // the event emitter.
15: emitter.on(i1, function() {
16:   console.log( "Event " + i1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(i2, function() {
19:   console.log( "Event " + i2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(i1, function() {console.log(
24:  "Something has been printed!!");});
25:
26: // Generate the events periodically...
27: // First event generated every 2 seconds.
28: setInterval( function() {
29:    emitter.emit(i1);}, 2000 );
30: // Second event generated every 3 seconds.
31: setInterval( function() {
32:    emitter.emit(i2);}, 3000 );
```

El programa principal instància l'objecte “emitter” en la línia 5 i crea quatre variables entre les línies 7 i 11. Posteriorment associa tres funcions als dos esdeveniments que van a crear-se. Finalment, en les línies 28 i 29 es programa l'esdeveniment “i1” (print) perquè ocórrega cada dos segons i en les línies 31 i 32 es programa l'esdeveniment “i2” (read) perquè ocórrega cada tres segons. Fet això, el fil principal acaba.

Cada vegada que es genere l'esdeveniment “i1” es dipositaran en la cua d'esdeveniments dues funcions, segons s'indica en les línies 15 a 17 (aquesta funció imprimeix el nombre de vegades que ha ocorregut l’esdeveniment) i en les línies 23 i 24 (aquesta funció sempre imprimeix el missatge “Something has been printed!!”). Per la seua banda, cada vegada que es genere l'esdeveniment “i2” s'inserirà en la cua d'esdeveniments la funció especificada en les línies 18 a 20 que imprimeix el nombre de vegades que ha ocorregut l'esdeveniment “read”.
Com els intervals de generació dels esdeveniments són molt amplis, la cua d'esdeveniments romandrà normalment buida. Quan es generen els esdeveniments, es deixen les seues funcions tractadores en la cua d'esdeveniments i passen a executar-se de seguida, buidant-se de nou la cua de torns.

La part inicial de l'eixida proporcionada serà:

```
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!!
Event read has happened 4 times.
Event print has happened 6 times.
Something has been printed!!
Event print has happened 7 times.
Something has been printed!!
Event read has happened 5 times.
Event print has happened 8 times.
Something has been printed!!
Event read has happened 6 times.
Event print has happened 9 times.
Something has been printed!!
Event print has happened 10 times.
Something has been printed!!
Event read has happened 7 times.
Event print has happened 11 times.
Something has been printed!!
Event read has happened 8 times.
Event print has happened 12 times.
Something has been printed!!
Event print has happened 13 times.
Something has been printed!!
Event read has happened 9 times.
```

Com a exercici senzill, justifique quina serà l'eixida del següent programa, on s'estén lleugerament l'exemple anterior.

```
 1: /*************************************************************/
 2: /* Events2.js                                                */
 3: /*************************************************************/
 4: const ev = require('events');
 5: let emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: const i1 = "print";
 8: const i2 = "read";
 9: // Auxiliary variables.
10: let num1 = 0;
11: let num2 = 0;
12:
13: // Listener functions are registered in
14: // the event emitter.
15: emitter.on(i1, function() {
16:   console.log( "Event " + i1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(i2, function() {
19:   console.log( "Event " + i2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(i1, function() {console.log(
```

```
24:   "Something has been printed!!");});
25:
26:  // Generate the events periodically...
27:  // First event generated every 2 seconds.
28:  setInterval( function() {
29:     emitter.emit(i1);}, 2000 );
30:  // Second event generated every 3 seconds.
31:  setInterval( function() {
32:     emitter.emit(i2);}, 3000 );
33:  // Loop.
34:  while (true)
35:     console.log(“.”);
```

3.3. Gestió de mòduls

La presentació descriu com poden exportar-se les funcions “públiques” que anem a definir en un mòdul, utilitzant “`exports`”, i com han d'importar-se des d'uns altres mitjançant “`require()`”. Allí es proporcionen alguns exemples d'ús.

El que no es comenta en ella és que “`exports`” és només un àlies para “`module.exports`” i que tant “`module.exports`” com “`module`” són objectes JavaScript. Això implica que, en ser objectes, disposaran d'un conjunt dinàmic de propietats, que podrem ampliar o reduir a voluntat. Cada mòdul té un objecte “module” privat. No és un objecte global comú a tots els fitxers. Amb ell podem especificar quines operacions i quines variables seran exportades pel mòdul.

Per a afegir una propietat o mètode a un objecte n'hi ha prou amb declarar-los i assignar-los un valor. És el que es mostra en els exemples d'aquella fulla. Tant “`area()`” com “`circumference()`” s'utilitzen com a noves propietats de l'objecte “module.exports”. A l'ésser de tipus funció, s'aconsegueix així exportar-les com a operacions del mòdul.

Per a eliminar una propietat n'hi ha prou amb utilitzar l'operador “**delete**” seguit del nom de la propietat a eliminar.

Aquest mecanisme permetrà que puguem construir mòduls que importen a uns altres i modifiquen algunes de les seues operacions, mantenint les altres. Per exemple, aquest codi...

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

delete c.circumference;

c.circumferència= function( r ) {
  return MATH.PI * r * 2;
}

module.exports = c;
```

...és capaç de renombrar l'operació “`circumference()`” del mòdul “Circle.js”, passant a anomenar-la “`circumferència()`”, sense afectar la resta d'operacions del mòdul original. Per a aconseguir-ho només necessita fer el següent:

1. Quan s'importa un mòdul mitjançant "`require()`", assignem l'objecte exportat a una variable del nostre programa. En aquest cas és la variable "`c`", que mantindrà l'objecte exportat en el fitxer "Circle.js".
2. A continuació eliminem una de les operacions exportades: "`circumference()`". L'altra operació, "`area()`", no s'ha vist afectada.
3. Posteriorment afegim una nova operació "`circumferència()`" que en aquest cas manté un codi similar al que tenia l'operació original.
4. Com a última sentència, exportem tot l'objecte "`c`", amb el que s'oferiran les operacions "`area()`" i "`circumferència()`" a qui ho importe.

Si aquest codi es guardara en un fitxer "Cercle.js", ja tindríem un mòdul amb aquest nom que exportaria les dues operacions.