Tema 2 – JavaScript y NodeJS

Guía de estudio

1. Introducción

La presentación ofrecida en el aula sobre este tema tiene tres secciones principales:

1. Una introducción que explica por qué JavaScript y NodeJS se han elegido como el lenguaje de programación y el intérprete, respectivamente, a utilizar en TSR.
2. Una segunda sección sobre el lenguaje JavaScript.
3. Una tercera sección sobre el intérprete node.

La presentación ofrece un breve resumen para cada uno de esos apartados. Esta guía debería ampliar esos resúmenes. Sin embargo, el formato habitual de estas guías no es suficiente para describir con la profundidad necesaria los aspectos más destacados de esos dos últimos apartados. Además, hay excelentes publicaciones sobre esa materia. Por ello, nos limitaremos a seleccionar los apartados más relevantes de algunas de esas publicaciones, invitando al lector a que las consulte con detenimiento. Posteriormente incluiremos otras secciones en las que se explicarán algunos conceptos relevantes.

2. JavaScript

Nuestra presentación sobre JavaScript menciona esta posible estructura para su contenido:

1. Características principales.
2. Alternativas en la ejecución de código. Esta sección es auto-contenida. No se necesita ampliarla.
3. Sintaxis.
4. Valores primitivos y compuestos.
5. Variables. Ámbito. Clausuras.
6. Operadores.
7. Sentencias.
8. Funciones.
9. Vectores.
10. Programación funcional.
11. Objetos.
12. Serialización. JSON.
13. “Callbacks”. Ejecución asincrónica. Promesas.
14. Eventos.

La presentación solo facilita un pequeño resumen para alguna de esas secciones. Su contenido puede extenderse en este URL: https://www.tutorialspoint.com/es6/index.htm. Allí, Tutorials Point facilita un tutorial sobre ECMAScript 6 (la edición de JavaScript asumida en las herramientas utilizadas en esta asignatura). Hay también una guía rápida que resume el contenido del tutorial: https://www.tutorialspoint.com/es6/es6_quick_guide.htm.

En ese tutorial, basta con utilizar estas partes (hay muchas más):

- Overview: Cubre la sección 1 de nuestra lista.
- Environment: Describe cómo instalar NodeJS y Visual Studio Code en varios sistemas. Ambos se utilizan en esta asignatura.

- Syntax: Cubre las secciones 3 y 5 de nuestra lista.
- Variables: Cubre las secciones 4 y 5 en nuestra lista.
- Operators: Cubre la sección 6 de la lista.
- Decision making: Cubre parcialmente la sección 7.
- Loops: También cubre parcialmente la sección 7.
- Functions: Cubre la sección 8 (y también parcialmente la 5) de la lista.
- Arrays: Cubre la sección 9.
- Classes: Cubre, de manera avanzada, la sección 11.
- Promises: Cubre, de manera sucinta, las secciones 13 y 14 de la lista.
- Error handling: Aunque esta parte no se discute directamente en la presentación del tema, su estudio es relevante. Recomendamos su lectura, aunque el apartado sobre onerror() no puede aplicarse a NodeJS, pues solo tiene sentido en los navegadores web.

El resto del tutorial cubre partes de JavaScript que solo se utilizarán en los navegadores web. Su contenido no es relevante para esta asignatura.

Obsérvese que el tutorial no cubre completamente todo el contenido del tema. Por ejemplo, las secciones 5 (Clausuras), 10 y 12 no han sido cubiertas. La sección 10 describe una parte de JavaScript que no es importante en esta asignatura y la sección 12 puede entenderse fácilmente cuando se consulten unos pocos ejemplos. Por otra parte, las clausuras son muy importantes. Por tanto, necesitamos alguna fuente adicional para entender mejor qué es una clausura en JavaScript.

Una primera solución para ese problema es esta descripción de las clausuras, disponible en el sitio w3schools.com: https://www.w3schools.com/js/js_function_closures.asp.

Otra solución consiste en leer algún libro sobre JavaScript. Hay muchos. Un ejemplo es Eloquent JavaScript, 3rd edition, de Marijn Haverbeke. Incluye una introducción breve y 21 capítulos. De entre ellos, los recomendables para nuestra asignatura son:

- Values, types and operators (Capítulo 1): Cubre las secciones 1, 3, 4 y 6 (esta última, parcialmente) de la lista.
- Program structure (Capítulo 2): Cubre las secciones 5, 6 y 7 de la lista.
- Functions (Capítulo 3): Trata sobre las secciones 5 y 8 de la lista. En este capítulo se define correctamente qué es una clausura, pero quizá sean necesarios algunos ejemplos más.
- Data structures: Objects and arrays (Capítulo 4): Cubre las secciones 9, 11 y 12 de la lista.
- Higher-order functions (Capítulo 5): Cubre en profundidad lo que se resume en la sección 10 de la lista. Debe considerarse una lectura opcional, pues la sección 10 se incluye en el Tema 2 por completitud. No es una parte central en nuestra asignatura.
- Asynchronous programming (Capítulo 11): Cubre en profundidad las secciones 13 y 14 de la lista.

Además, el libro facilita múltiples ejemplos breves de programas y también ejercicios en cada capítulo.

Para ampliar esas referencias externas y libros, facilitamos en las subsecciones siguientes descripciones de aquellos elementos que todavía no se han cubierto con la profundidad adecuada. Sus números de sección coinciden con los utilizados en la presentación.

2.4.3. Clausuras

En algunos casos resulta útil declarar una función dentro de otra. En esas situaciones puede que resulte interesante retornar la función interna. Esa función interna seguirá teniendo acceso a las variables utilizadas en la función que la engloba, así como a sus argumentos. Esto sigue siendo válido aunque la invocación que la creó haya terminado. El hecho de que se mantenga ese contexto externo se conoce como "clausura".

Veamos un ejemplo en el que podremos utilizar clausuras. La función "log()" del módulo Math de JavaScript retorna el logaritmo neperiano del valor recibido como argumento. Utilizaremos clausuras para implantar una función que "construya" y retorne otra capaz de calcular el logaritmo en la base que especifiquemos. Para ello aprovecharemos esta propiedad de los logaritmos:

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

El código a utilizar sería:

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

Como puede observarse, la función "constructora" se llama "logBase()" y utiliza el parámetro "b" para especificar la base a utilizar. Su única sentencia se encarga de retornar una función. Esa función es una función anónima que recibe como único parámetro un valor "x". El código de esa función retornada se encarga de aplicar la propiedad que hemos enunciado arriba, recordando el argumento recibido en la función constructora.

Tomando ese código como punto de partida, podremos generar las funciones necesarias para calcular logaritmos en diferentes bases. En el siguiente fragmento utilizamos tanto la base 2 como la 8. Obsérvese que resulta sencillo seguir el código resultante:

```
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
```

Puede que necesitemos usar alguna variable de la función que proporcione el contexto de la clausura, en lugar de utilizar únicamente alguno de sus parámetros. En ese caso debe recordarse que el acceso a esas variables se hace por referencia y eso puede ser problemático.

Veamos un ejemplo de ello. En el siguiente fragmento de código se desea desarrollar una función que devuelva un vector que contiene tres funciones. Cada una de ellas devolverá el nombre y población de cada uno de los tres países más poblados de nuestro planeta. Esta solución 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 llamada efectuada en la línea 16 ejecuta el código de la función “populations()”. Sería de esperar que el vector “ps” contuviese las tres funciones solicitadas y que las llamadas efectuadas en las líneas 22 a 24 mostraran el nombre y población de los tres países más poblados. Sin embargo, la salida obtenida es:

```
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
```

Esto se justifica porque cuando llamamos a las funciones “first()”, “second()” y “third()” el valor de la variable “i” dentro de la función “populations()” es 3. Poco importa que cuando se crearon esas tres funciones valiese 0, 1 y 2, respectivamente. Seguimos utilizando la variable “i” cuando se invocan esas funciones creadas (es como si se estuviera pasando por referencia) y su valor actual es 3. Por ello, la salida obtenida no es la esperada.

¿Cómo podemos resolverlo? Utilizando clausuras a la hora de acceder al valor de "i". Nuestro objetivo es que la función que está siendo declarada entre las líneas 7 y 12 recuerde el valor que tenía "i" en esa iteración del bucle. No nos interesa el valor que tendrá la "i" cuando invoquemos a las funciones que retornaremos en el vector sino el que tiene en este momento.

La solución se muestra a continuación:

```
 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();
```

Los cambios se han resaltado en negrita. Necesitamos otra función que la encierre y que reciba como único argumento el valor actual de la variable "i". Para ello, esa nueva función recibe un parámetro "x" y la función que retornaremos accederá a "x" en lugar de acceder a "i". Por ello en cada iteración del bucle se invoca a esa función que engloba a la que íbamos a retornar, pasando como argumento el valor actual de "i" (véase la línea 14). Lo único que hace esa función contenedora es retornar a la función que en la versión anterior estaba listada entre las líneas 7 y 12.

Con este "truco" conseguimos que la salida proporcionada por el programa sea:

```
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
```

Esa era la salida a obtener. Con esto queda demostrado que debe llevarse cierto cuidado a la hora de utilizar bucles para generar funciones utilizando clausuras.

ECMAScript 6 introdujo una segunda forma de resolver este problema. Para ello se estableció una segunda forma de declarar variables. Tradicionalmente, se había utilizado la palabra reservada "var" para ello. ECMAScript 6 introdujo la palabra reservada "let". Al declarar con "let" la variable índice en un bucle, cada vez que se utilice la variable, su valor se pasa directamente a la sentencia que lo use, en lugar de pasar una referencia a la variable. Así, en lugar de utilizar clausuras, habría bastado con añadir "let" en la declaración (hasta el momento implícita) de la variable "i" dentro del bucle del programa visto hasta el momento. La versión resultante sería:

```
 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();
```

Y su salida es idéntica a la del programa anterior, en el que habíamos usado clausuras.

2.5. Callbacks

Implementación de callbacks para funciones asíncronas

La programación asíncrona en Node.js se basa en el uso de las funciones *callback*. Cuando un programa contiene la invocación de una función asíncrona, llamémosla ***f_async***, que lleve un *callback* como último argumento, dicho programa no se bloquea hasta que ***f_async*** concluya, sino que continuará ejecutando las instrucciones siguientes que tenga. Mientras tanto, la función asíncrona se irá ejecutando y, a su final, en un turno posterior, se ejecutará su *callback*.

En "Control Flow in Node"[1], Tim Caswell presenta algunos ejemplos interesantes para comprender el funcionamiento de las funciones asíncronas y los callbacks. El siguiente código, con leves modificaciones, está tomado de un ejemplo 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('ejecutando otras instrucciones');
14: console.log('raiz(2) =', Math.sqrt(2));
```

La llamada a la función asíncrona ***readFile*** del módulo ***fs*** no bloquea este programa. Las instrucciones de las líneas 13 y 14 se ejecutarán antes que se complete ***readFile*** y, por tanto, antes de la función anónima de callback que recibe. Si el intento de lectura del fichero 'mydata.txt' fallara, en el callback se ejecuta la instrucción de la línea 6. En tal caso, la salida de este programa sería:

```
ejecutando otras instrucciones
raiz(2) = 1.4142 ...
Error: ENOENT, open '… /mydata.txt'
```

La función ***readFIle*** recibe un nombre de fichero y "devuelve" su contenido. En sentido estricto, no devuelve nada, sino que lo pasa como argumento a su función callback. El segundo argumento del callback (*buffer* en el ejemplo anterior) recibe el contenido del fichero cuando la operación de lectura se lleva a cabo correctamente. El primer argumento del callback (*err* en el ejemplo) recibe el mensaje de error, generado cuando la lectura falla por alguna causa (como ha sucedido en el anterior ejemplo de ejecución).

Definir la función de callback como anónima es usual si sólo se utiliza una vez, como en el ejemplo anterior. Pero, en el caso de que se use más veces, lo recomendable es definirla con nombre. El siguiente ejemplo de Tim Caswell, modificación del anterior, ilustra este caso.

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

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

```
 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);
```

La implementación de funciones asíncronas mediante callbacks permite la anidación sin otro límite que las necesidades del programador. En "Accessing the File System in Node.js"[2], Colin Ihrig presenta un ejemplo de anidación bastante 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 este ejemplo, se lee el contenido de un fichero con la ayuda de un buffer. El callback de la función asíncrona ***exists*** del módulo ***fs*** contiene una llamada a otra función asíncrona, ***stats***, cuyo callback, a su vez, invoca a otra función asíncrona, ***open***, cuyo callback invoca a la función asíncrona ***read***, la cual también tiene su callback.

Ciertamente existen otras formas de leer un fichero: usar ***readFile***, como en los ejemplos propuestos por Caswell, o usar versiones síncronas de las funciones del módulo ***fs***. El fin de presentar este último programa es mostrar las posibilidades de anidación de callbacks... y sus inconvenientes. Un alto nivel de anidación, como se aprecia, dificulta la lectura del código. En este ejemplo, además, no se lleva a cabo ninguna gestión de errores. Si esto se hiciera, el

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

código aún sería más difícil de interpretar, y de seguir su ejecución. En estas situaciones, se hace muy conveniente el uso de las promesas como alternativa en la programación asíncrona.

El anidamiento de callbacks es un modo de asegurar que las sucesivas operaciones (desde la más externa a la más interna) se ejecutan en secuencia. Pero también pueden plantearse situaciones en las que convenga agrupar un conjunto de operaciones paralelas. Un ejemplo lo proporciona Tim Caswell (en las páginas web antes citadas): la lectura de todos los ficheros existentes en un directorio. El código, con leves modificaciones, es el siguiente:

```
 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 función asíncrona ***readdir*** recibe la lista de los ficheros existentes en el directorio actual. Con esta lista, y mediante el operador ***forEach***, se inicia la lectura de cada uno de esos ficheros, mediante la función asíncrona ***readFile***. Así pues, todas las operaciones de lectura se ejecutan en paralelo y pueden terminar en cualquier orden. Conforme concluyen, se ejecutan sus respectivos callbacks. En éstos, se actualiza una variable contador, ***count***, lo que permite detectar cuando todas las lecturas han terminado por si se quiere realizar alguna operación posterior.

Implementación de funciones asíncronas

En los ejemplos de la sección anterior, se han invocado funciones asíncronas ya existentes, funciones del módulo ***fs***. Otros módulos estándar de Node.js proporcionan otras funciones asíncronas. En tales casos, el programador sólo necesita invocarlas, proporcionando los argumentos adecuados (lo que implica implementar la función callback que será su último argumento). En esta sección se presenta cómo implementar una función cualquiera, inicialmente síncrona, para que tenga un comportamiento asíncrono.

Considérese el siguiente código:

```
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('Iniciando ejecución...')
12: console.log('fibo(40) =', fibo(40))
13: console.log('lanzado cálculo fibonacci...')
14: console.log('fact(10) =', fact(10))
15: console.log('lanzado cálculo factorial...')
```

Las dos funciones aquí definidas (la función ***fibo*** para calcular el término enésimo de la sucesión de Fibonacci y la función ***fact*** para calcular el factorial del número recibido como argumento) son síncronas. Por ello, las invocaciones a las mismas, en las líneas 12 y 14 de este código, bloquean el programa. La salida obtenida es:

```
Iniciando ejecución...
fibo(40) = 165580141
lanzado cálculo fibonacci...
fact(10) = 3628800
lanzado cálculo factorial...
```

Además, la segunda línea de esta salida tarda algunos segundos en mostrarse, dado el coste computacional de la función ***fibo***. Sería deseable disponer de versiones asíncronas de las funciones, de modo que se invocaran sin bloquear el programa en las líneas 12 y 14, y los mensajes de las líneas 13 y 15 se mostraran de inmediato (y se ejecutaran otras instrucciones posteriores, si las hubiera).

Una forma bastante sencilla de implementar esas versiones asíncronas de las funciones consiste en añadir ***console.log*** como función callback y usar la función ***setTimeout*** para envolver la invocación de las funciones (y pasarlas así a la cola de eventos con unos temporizadores adecuados). Con estas modificaciones, se tendría el siguiente 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('Iniciando ejecución...')
18: setTimeout( function(){
19:    fibo_back1(40, console.log)
20: }, 2000 )
21: console.log('lanzado cálculo fibonacci...')
22: setTimeout( function(){
23:    fact_back1(10, console.log)
24: }, 1000 )
25: console.log('lanzado cálculo factorial...')
```

La salida obtenida al ejecutar esta segunda versión del programa es:

```
Iniciando ejecución...
lanzado cálculo fibonacci...
lanzado cálculo factorial...
factorial(10) = 3628800
fibonacci(40) = 165580141
```

Además, sólo la aparición de la última línea de esta salida presenta un retraso apreciable.

Aunque el código anterior es correcto y funciona como se esperaba, la función usada como callback, ***console.log***, no se ajusta al convenio habitual. En Node.js, se recomienda que las funciones callback reciban, como primer argumento, la información de error (si la función asíncrona fallara) y, como segundo argumento, el resultado proporcionado (cuando la función asíncrona termina correctamente).

Si se modifica el anterior programa teniendo en cuenta el citado convenio, se tendría lo siguiente:

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

Ahora, las funciones asíncronas, ***fibo_back2*** y ***fact_back2***, reciben como callback la función ***show_back*** (implementada en las líneas 19 a 22). Se ha añadido otra función auxiliar, ***eval_err*** (en las líneas 24 a 27), que genera el mensaje de error cuando corresponde (se ha supuesto que las situaciones de error se dan cuando las funciones se invocan con un argumento no numérico).

La salida obtenida al ejecutar el programa ***fibo3.js*** es:

```
Iniciando ejecución...
lanzado cálculo fibonacci...
lanzado cálculo factorial...
factorial(10) = 3628800
factorial(ana) ??? : ana is not a number
fibonacci(40) = 165580141
fibonacci(pep) ??? : pep is not a number
```

Las funciones “asíncronas” definidas hasta el momento lo son sólo formalmente, por cuanto reciben un callback como argumento, pero no lo son en su comportamiento: la asincronía se debe a su invocación como argumento de la función ***setTimeout***.

Una forma de generar versiones realmente asíncronas consiste en utilizar la función ***nextTick*** de ***process***[3]. Esta función permite retrasar la ejecución de una acción hasta la siguiente

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

iteración del bucle de eventos. Considérese la siguiente modificación del programa teniendo en cuenta esto:

```
 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:
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('Iniciando ejecución...')
34: fact_async(10, show_back)
35: fact_async('ana', show_back)
36: console.log('lanzados cálculo factorial...')
37: fibo_async(40, show_back)
38: fibo_async('pep', show_back)
39: console.log('lanzados cálculo fibonacci...')
```

La salida obtenida al ejecutar el programa ***fibo4.js*** es la misma que al ejecutar ***fibo3.js***. Obsérvese, en las líneas 33 a 39, que no se requiere el uso de ***setTimeout***. Obsérvese también que se ha modificado el orden de las instrucciones: las invocaciones a ***fact_async*** (líneas 34-35) se hacen antes que las invocaciones a ***fibo_async*** (líneas 37-38), dado que ahora no se establece un temporizador explícito a cada función, y se quiere seguir mostrando antes el resultado del cálculo de factorial.

Supóngase ahora que se deseara calcular un cierto número de valores resultado de ambas funciones y que dichos valores se almacenaran en sendos arrays. Un programa que hiciera esto sería el siguiente:

```
 1: // ***  fibo5.js
 2:
 3: // *** funciones
 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('Iniciando ejecución...')
```

```
48:
49: let facts = []
50: for (let i=0; i<=10; i++)
51:     fact_async(i, show_fibo_back)
52: console.log('lanzado cálculo de factoriales...')
53:
54: let fibs = []
55: for (let i=0; i<=20; i++)
56:     fibo_async(i, show_fact_back)
57: console.log('lanzado cálculo de fibonaccis...')
```

En este programa, se ha modificado la implementación del callback. Ahora hay dos funciones callback (una para cada función asíncrona), que almacenan el resultado del cálculo en la posición indicada del array correspondiente, además de mostrarlo en consola.

La salida obtenida al ejecutar el programa ***fibo5.js*** (no se muestra la salida completa, las líneas con puntos suspensivos representan la salida no reproducida explícitamente) es:

```
Iniciando ejecución...
lanzado cálculo de factoriales...
lanzado cálculo 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 continuación, considérese ampliar el programa para que proporcione la suma de todos los factoriales calculados y la suma de todos los términos de Fibonacci calculados. Podría juzgarse que una posible solución sería la siguiente:

```
 1: // ***  fibo6a.js
 2:
 3: // *** funciones:  misma implementación 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('Iniciando ejecución...')
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('lanzado cálculo de factoriales...')
24:
25: for (let i=0; i<n; i++)
26:    fibo_async(i, show_fibo_back)
27: console.log('lanzado cálculo de fibonaccis...')
28:
29: console.log('suma factoriales ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
30: console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
```

Sin embargo, la salida obtenida al ejecutar ***fibo6a.js*** es:

```
Iniciando ejecución...
lanzado cálculo de factoriales...
lanzado cálculo de fibonaccis...
…
… /fibo6a.js:29
   console.log('suma factoriales ['+0+'..'+(n-1)+'] =', facts.reduce(suma))

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

Este resultado (error al intentar aplicar la función ***reduce*** a un array vacío, no inicializado) se produce porque las líneas 29 y 30 se ejecutan antes que las funciones asíncronas y sus callbacks.

Una solución sencilla a este problema es envolver las instrucciones de las líneas 29 y 30 con una función que pase su ejecución a la cola de turnos de Node. Por ejemplo:

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

O también mediante:

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

Cualquiera de estas dos soluciones, proporciona la siguiente salida al ejecutar el programa:

```
Iniciando ejecución...
lanzado cálculo de factoriales...
lanzado cálculo de fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
suma factoriales [0..9] = 409114
suma fibonaccis [0..9] = 143
```

Otra posibilidad es modificar los callbacks de las funciones asíncronas para que, una vez calculadas todas las componentes del array, calculen las sumas. Esta solución sería la siguiente:

```
 1: // *** fibo6b.js
 2:
 3: // *** funciones:  misma implementación que en fibo6a.js, excepto 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:          let 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:          let s = facts.reduce(suma)
30:          console.log('suma factoriales ['+0+'..'+(n-1)+'] =', s)
```

```
31:         }
32:     }
33: }
34:
35: // *** programa principal
36: console.log('Iniciando ejecución...')
37:
38: const n = 10
39: let facts = []
40: let fibs = []
41:
42: for (var i=0; i<n; i++)
43:     fact_async(i, show_fact_back)
44: console.log('lanzado cálculo de factoriales...')
45:
46: for (var i=0; i<n; i++)
47:     fibo_async(i, show_fibo_back)
48: console.log('lanzado cálculo de fibonaccis...')
```

La salida obtenida al ejecutar ***fibo6b.js*** es:

```
Iniciando ejecución...
lanzado cálculo de factoriales...
lanzado cálculo de fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
suma factoriales [0..9] = 409114
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
suma fibonaccis [0..9] = 143
```

Esta salida es correcta y sólo difiere (respecto a las soluciones de envolver las llamadas a ***reduce*** con ***setTimeout*** o ***nextTick***) en el orden en el que se presentan los resultados en la consola.

2.5.2. Promesas

Hay otra forma de realizar ejecuciones asíncronas en JavaScript: mediante promesas. Aunque durante los últimos años ha habido varias propuestas para introducir promesas en JavaScript, soportadas mediante diferentes módulos que podían incluirse en los programas que desarrolláramos, en el estándar ECMAscript 6 únicamente se ha aceptado la variante basada en constructores, proporcionando el objeto Promise para este fin.

Esta sección aplica esa variante al ejemplo de cálculo de los términos de la sucesión de Fibonacci, presentado en la sección anterior.

El programa a utilizar es el siguiente:

```
 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: let elems = [25, '5', true]
28:
29: for (let i in elems)
30:    fibo_promise(elems[i]).then( onSuccess(elems[i]), onError )
```

En la función ***fibo_promise*** se construye y devuelve un objeto promesa, con una función que proporciona el tratamiento debido. Esa función necesita tener dos parámetros. El primero es el callback a utilizar cuando la promesa genere un resultado correcto. Como vemos en este ejemplo, ese callback recibe como argumento el resultado de la invocación a la función que se pretenda convertir en promesa. En este caso ha sido la función fibo(). Por su parte, el segundo es el callback que se invocará cuando la promesa genere un error o excepción.

Tanto la segunda como la tercera componente del vector “elems” generan un error al invocar a la función fibo(), mostrando de inmediato un error. A su vez, fibo(25) necesita algún tiempo para completar su cálculo. Por ello, en la salida del programa anterior se observará en primer lugar el mensaje de error para la cadena “5”, seguido por un mensaje similar para el valor booleano “true” y, por último, se mostrará el resultado de fibo(25), cuando se haya completado su cómputo.

En el siguiente programa se considera de nuevo el problema de calcular factoriales y términos de Fibonacci almacenándolos en arrays. Una implementación usando promesas es la siguiente:

```
 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: }
```

La salida obtenida al ejecutar el programa ***fibo8.js*** es:

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

Considérese ahora el mismo problema, y además el problema de sumar las componentes de los arrays resultado. Un programa correcto usando promesas es el siguiente:

```
 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) )
```

La salida obtenida al ejecutar el programa ***fibo9.js*** es:

```
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
```

Como se acaba de comprobar, el uso de arrays de promesas resuelve satisfactoriamente el problema de calcular la suma de componentes de un array. El manejador de éxito asociado al resultado del array de promesas, función ***sumAll***, se ejecuta solamente después de que todas las promesas han sido satisfechas, es decir, después de que se hayan calculado todos los valores que se deseaba almacenar en el array.

El encadenamiento de las promesas usando la función ***then*** facilita al programador establecer la secuencia adecuada de acciones. Como se puede apreciar, dicha secuencia es más fácil de leer en el código (respecto a la solución basada en callbacks).

En los programas ***fibo8.js*** y ***fibo9.js***, se puede apreciar la extrema similitud de las funciones ***fibo_promise*** y ***fact_promise***. En el siguiente programa se proporciona otra implementación equivalente, con menos código:

```
 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 documentación oficial sobre NodeJS (https://nodejs.org/en/docs/) facilita toda la información necesaria en este segundo bloque del tema. Puede consultarse si es necesario. Podemos distinguir dos clases de documentación:

- Una referencia para su API. Desde la página inicial de documentación se puede acceder a esa información para la actual última edición y para la LTS. Sin embargo, habrá una versión más antigua en los laboratorios. De hecho, la edición que primero cubrió el estándar ECMAScript 6 fue la NodeJS 6.x.x. Su referencia está aquí: https://nodejs.org/docs/latest-v6.x/api/. No es necesario un conocimiento exhaustivo de ese conjunto de operaciones.
- Guías complementarias. Disponibles en: https://nodejs.org/en/docs/guides/. Hay tres tipos de guías: (1) general, (2) conceptos importantes y (3) relativas a módulos. El segundo tipo describe algunos conceptos centrales del "runtime" de NodeJS, tales como los temporizadores, el bucle de gestión de eventos, las diferencias entre operaciones bloqueantes y no bloqueantes, etc. Estas guías del segundo tipo son una lectura recomendable.

3.2. Asincronía. Cola de eventos

El *runtime* de NodeJS tiene una cola de eventos (también conocida como "cola de turnos"). Esto se debe a que JavaScript es un lenguaje que no soporta múltiples hilos de ejecución. Por ello, cuando en alguna parte de un programa se genere otra actividad, esa actividad se modela como un nuevo "evento" y es añadida a la cola de eventos o turnos. Esos turnos se sirven en orden FIFO. Para iniciar el servicio de un nuevo turno habrá terminado la ejecución del turno anterior.

Analicemos este ejemplo:

```
 1: function fibo(n) {
 2:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1);
 3: }
 4: console.log("Iniciando ejecución...");
 5: // Esperar 10 ms para escribir un mensaje.
 6: // Implica generar nuevo evento.
 7: setTimeout( function() {
 8:   console.log( "M1: Quiero escribir esto..." );
 9: }, 10 );
10: // Más de 5 segs. para ejecutar fibo(40)
11: let j = fibo(40);
12: function otroMensaje(m,u) {
13:   console.log( m + ": El resultado es: " + u );
14: }
15: // M2 se escribe antes que M1 porque el
16: // hilo "principal" no se suspende...
17: otroMensaje("M2",j);
18: // M3 ya se escribe tras M1.
19: // Es otro evento (a ejecutar tras 1 ms).
20: setTimeout( function() {
21:  otroMensaje("M3",j);
22: }, 1 );
```

La función setTimeout() se utiliza en JavaScript para programar la ejecución de la función recibida en su primer parámetro al cabo del número de milisegundos especificados en su segundo parámetro.

Sigamos una traza de este programa:

- La ejecución empieza en las líneas 1 a 3, donde se declara una función recursiva "fibo()" que recibirá un argumento entero y calculará el número de Fibonacci asociado a ese valor.
- Llegamos ahora a la línea 4 que escribe el mensaje "Iniciando ejecución..." en la pantalla.
- En las líneas 7 a 9 se utiliza setTimeout() para programar la escritura de otro mensaje ("M1: Quiero escribir esto...") dentro de 10 ms. De momento eso no tiene ningún efecto.
- Llegamos a la línea 11 donde se invoca la función "fibo()" con el valor 40. Su ejecución necesitará algunos segundos. Durante ese plazo habrán transcurrido los 10 ms mencionados en el punto anterior. Eso implica que la función que debe escribir el

mensaje M1 por pantalla está ya ubicada en la cola de eventos, esperando su turno. Ese turno no empezará mientras no termine el hilo principal.

- El hilo principal llegará al poco tiempo a las líneas 12 a 14, donde se declara la función otroMensaje() que utilizaremos posteriormente para escribir el número de Fibonacci obtenido.
- Con ello llegamos ya a la línea 17. En ella se escribe por primera vez el resultado. En pantalla se mostrará lo siguiente: "M2: El resultado es: 165580141"
- Y finalmente llegamos a las líneas 19 a 21, donde se programa de nuevo la invocación de otroMensaje() (mensaje M3) tras una espera de 1 ms. Con esto termina la ejecución del hilo principal.
- En este momento en la cola de eventos solo encontramos un turno activo. Iniciamos su ejecución. En este turno se imprimirá por pantalla este mensaje: "M1: Quiero escribir esto..." Durante esa escritura habrá finalizado la espera de 1 ms iniciada en el punto anterior. Con ello, se deposita un nuevo contexto en la cola de eventos, cuyo objetivo es escribir el mensaje M3.
- Una vez mostrado el mensaje M1, el primer turno que habíamos creado finaliza. Se revisa la cola de eventos y se observa que en él existe todavía otro contexto. Se inicia su ejecución, que muestra por pantalla el mensaje: "M3: El resultado es: 165580141".
- Con esto termina la ejecución del programa. La salida completa proporcionada es:

```
Iniciando ejecución...
M2: El resultado es: 165580141
M1: Quiero escribir esto...
M3: El resultado es: 165580141
```

Como puede observarse, las dos primeras líneas han sido impresas por el hilo principal. Por su parte, las líneas que empiezan con M1 y M3 han sido impresas utilizando dos eventos. Aunque el primero de esos eventos fue generado muy pronto (mientras se estaba ejecutando la función fibo(); bastante antes de que se pretendiera escribir el mensaje M2) su resultado no pudo mostrarse entonces por pantalla. Ha tenido que terminar antes el hilo principal.

Revisemos este segundo ejemplo:

```
 1: /*************************************************************/
 2: /* Events1.js                                                */
 3: /*************************************************************/
 4: const ev = require('events');
 5: let emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: const e1 = "print";
 8: const e2 = "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(e1, function() {
16:    console.log( "Event " + e1 + " has " +
17:      "happened " + ++num1 + " times.")});
18:  emitter.on(e2, function() {
19:    console.log( "Event " + e2 + " has " +
20:      "happened " + ++num2 + " times.")});
21:  // There might be more than one listener
22:  // for the same event.
23:  emitter.on(e1, 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(e1);}, 2000 );
30:  // Second event generated every 3 seconds.
31:  setInterval( function() {
32:     emitter.emit(e2);}, 3000 );
```

El programa principal instancia el objeto “emitter” en la línea 5 y crea cuatro variables entre las líneas 7 y 11. Posteriormente asocia tres funciones a los dos eventos que van a crearse. Finalmente, en las líneas 28 y 29 se programa el evento “e1” (print) para que ocurra cada dos segundos y en las líneas 31 y 32 se programa el evento “e2” (read) para que ocurra cada tres segundos. Hecho eso, el hilo principal termina.

Cada vez que se genere el evento “e1” se depositarán en la cola de eventos dos funciones, según se indica en las líneas 15 a 17 (esta función imprime el número de veces que ha ocurrido ese evento) y en las líneas 23 y 24 (esta función siempre imprime el mensaje “Something has been printed!!”). Por su parte, cada vez que se genere el evento “e2” se insertará en la cola de eventos la función especificada en las líneas 18 a 20 que imprime el número de veces que ha ocurrido el evento “read”.
Como los intervalos de disparo de los eventos son muy amplios, la cola de eventos permanecerá normalmente vacía. Cuando se generan los eventos, se dejan sus funciones manejadoras en la cola de eventos y pasan a ejecutarse en seguida, vaciándose de nuevo la cola de turnos.

La parte inicial de la salida proporcionada será:

```
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.
```

Como ejercicio sencillo, justifique cuál será la salida del siguiente programa, donde se extiende ligeramente el ejemplo anterior.

```
 1: /*************************************************************/
 2: /* Events2.js                                                */
 3: /*************************************************************/
 4: const ev = require('events');
 5: let emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: const e1 = "print";
 8: const e2 = "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(e1, function() {
16:   console.log( "Event " + e1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(e2, function() {
19:   console.log( "Event " + e2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(e1, 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(e1);}, 2000 );
30: // Second event generated every 3 seconds.
31: setInterval( function() {
32:    emitter.emit(e2);}, 3000 );
33: // Loop.
34: while (true)
35:    console.log(“.”);
```

3.3. Gestión de módulos

La presentación describe cómo pueden exportarse las funciones “públicas” que vayamos a definir en un módulo, utilizando “`exports`”, y cómo deben importarse desde otros mediante “`require()`”.

Lo que no se comenta en ella es que “`exports`” es sólo un alias para “`module.exports`” y que tanto “`module.exports`” como “`module`” son objetos JavaScript. Eso implica que, al ser objetos, dispondrán de un conjunto dinámico de propiedades, que podremos ampliar o reducir a voluntad. Cada módulo tiene un objeto “module” privado. No es un objeto global común a todos los ficheros. Con él podemos especificar qué operaciones y qué variables serán exportadas por el módulo.

Para añadir una propiedad o método a un objeto basta con declararlos y asignarles un valor. Tanto “`area()`” como “`circumference()`” se utilizan como nuevas propiedades del objeto “module.exports”. Al ser de tipo función, se consigue de esa manera exportarlas como operaciones del módulo.

Para eliminar una propiedad basta con utilizar el operador “**delete**” seguido del nombre de la propiedad a eliminar.

Este mecanismo permitirá que podamos construir módulos que importen a otros y modifiquen algunas de sus operaciones, manteniendo las demás. Por ejemplo, este código...

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

delete c.circumference;

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

module.exports = c;
```

...es capaz de renombrar la operación "`circumference()`" del módulo "Circle.js", pasando a llamarla "`circunferencia()`", sin afectar al resto de operaciones del módulo original. Para conseguirlo sólo necesita hacer lo siguiente:

1. Cuando se importa un módulo mediante "`require()`", asignamos el objeto exportado a una variable de nuestro programa. En este caso es la variable "`c`", que mantendrá el objeto exportado en el fichero "Circle.js".
2. A continuación eliminamos una de las operaciones exportadas: "`circumference()`". La otra operación, "`area()`", no se ha visto afectada.
3. Posteriormente añadimos una nueva operación "`circunferencia()`" que en este caso mantiene un código similar al que tenía la operación original.
4. Como última sentencia, exportamos todo el objeto "`c`", con lo que se ofrecerán las operaciones "`area()`" y "`circunferencia()`" a quien lo importe.

Si ese código se guardara en un fichero "Círculo.js", ya tendríamos un módulo con ese nombre que exportaría esas dos operaciones.