

TSR: Actividades del Tema 2

Este boletín contiene varias actividades. Podemos clasificarlas, en función de su relación con los objetivos a cubrir en este Tema 2, en estas clases:

- a) Imprescindibles (aspectos importantes del Tema 2 que deben dominarse): 2, 6 y 10.
- b) Recomendadas (otros aspectos importantes del Tema 2): 1, 3, 4, 5, 11, 12 y 13.
- c) Complementarias (aspectos accesorios del Tema 2, generalmente no evaluables): 7, 8 y 9.

ACTIVIDAD 1

OBJETIVO: Profundizar en la gestión del paso de argumentos en la invocación de funciones en JavaScript.

ENUNCIADO: Tomando como base el siguiente código¹:

```
1 function table(x) { // Prints column x of a (1..10) multiplication table
2     for (let j=1; j<11; j++)
3         console.log("%d * %d = %d", x, j, x*j)
4     console.log("")
5 }
6
7 function allTables() {
8     for (let i=1; i<11; i++)
9         table(i)
10 }
11
12 table(5, 4, 1)
```

- a) Describa cuál es la salida proporcionada por el programa anterior. Justifique si tiene o no algún efecto el pasar más de un argumento en la línea 12.

- b) Suponga que ahora reemplazamos la línea 12 original del programa anterior por la siguiente, ¿qué salida proporciona el programa en este caso? ¿por qué?

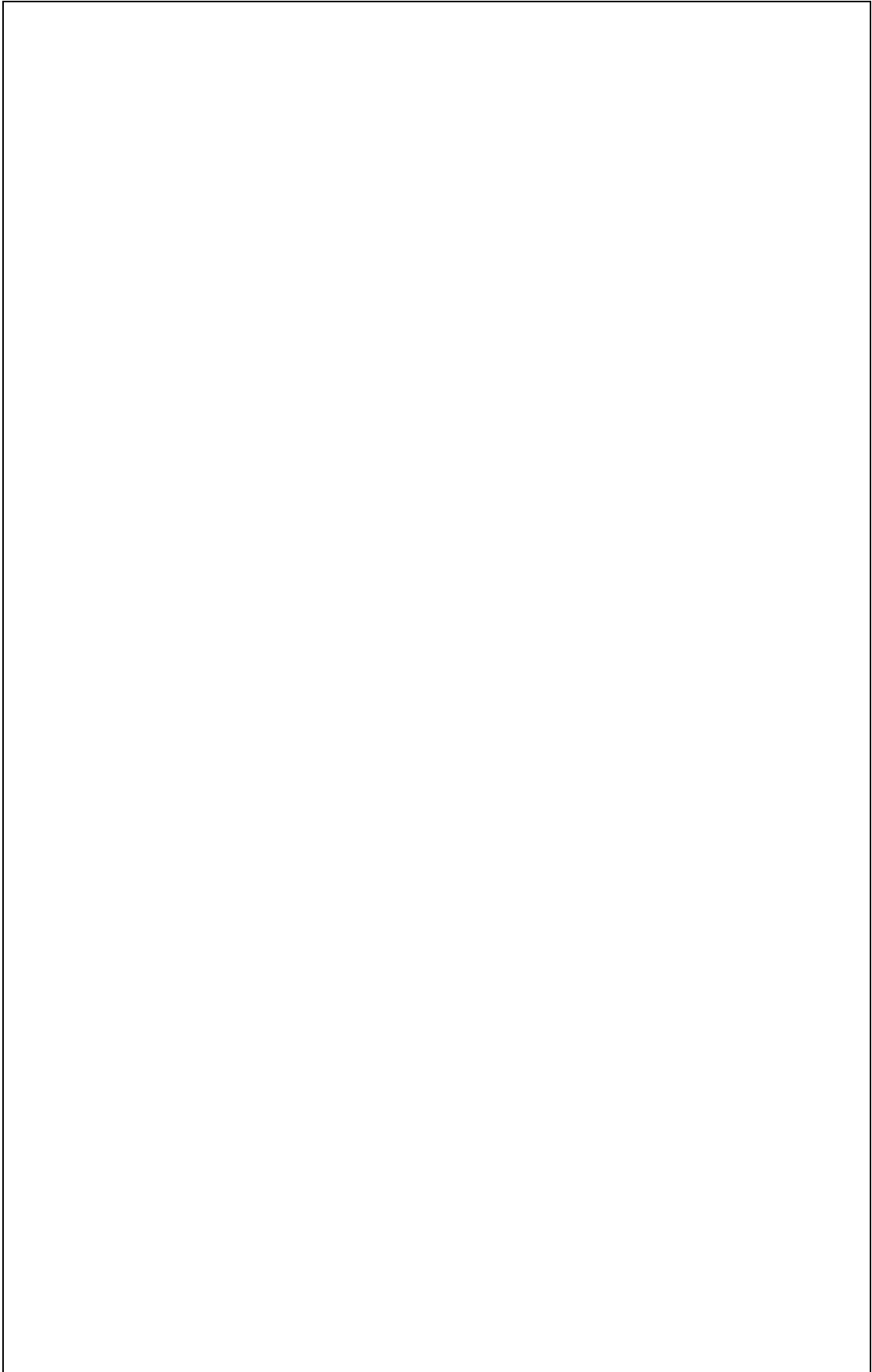
¹ Todos los ficheros fuente listados o mencionados en este documento se encuentran en un archivo "acts.zip" disponible en PoliformaT.

12	<code>table(table(2))</code>
----	------------------------------

- c) Suponga que ahora reemplazamos la línea 12 original del programa por la siguiente, ¿qué salida proporciona el programa ahora? ¿por qué?

12	<code>allTables(table(30),table(20),table(10))</code>
----	---

- d) A partir de los resultados obtenidos en los apartados anteriores, justifique si JavaScript acepta y puede tener algún efecto que se pasen más argumentos de los que espera una función determinada.



ACTIVIDAD 2

OBJETIVO: Adquirir soltura en la programación de eventos de JavaScript.

ENUNCIADO: Sea el programa siguiente:

```
1const ev = require('events')
2const emitter = new ev.EventEmitter
3const e1 = "print"
4const e2 = "read"
5const books = [ "Walk Me Home", "When I Found You", "Jane's Melody", "Pulse" ]
6
7// Function that creates the intended event listeners.
8function createListener( eventName ) {
9    let num = 0
10    return function (arg) {
11        let book = ""
12        if (arg)
13            book = ", now with book title '" + arg + "',"
14        console.log("Event " + eventName + book + " has " +
15            "happened " + ++num + " times.")
16    }
17}
18
19// Listeners are registered in the event emitter.
20emitter.on(e1, createListener(e1))
21emitter.on(e2, createListener(e2))
22// There might be more than one listener for the same event.
23emitter.on(e1, () => console.log("Something has been printed!!"))
24
25function emitE2() {
26    let counter=0
27    return function () {
28        // This second argument provides the argument for the "e2" listener.
29        emitter.emit(e2,books[counter++ % books.length])
30    }
31}
32// Generate the events periodically...
33// First event generated every 2 seconds.
34setInterval( () => emitter.emit(e1), 2000 )
35// Second event generated every 3 seconds.
36setInterval( emitE2(), 3000 )
```

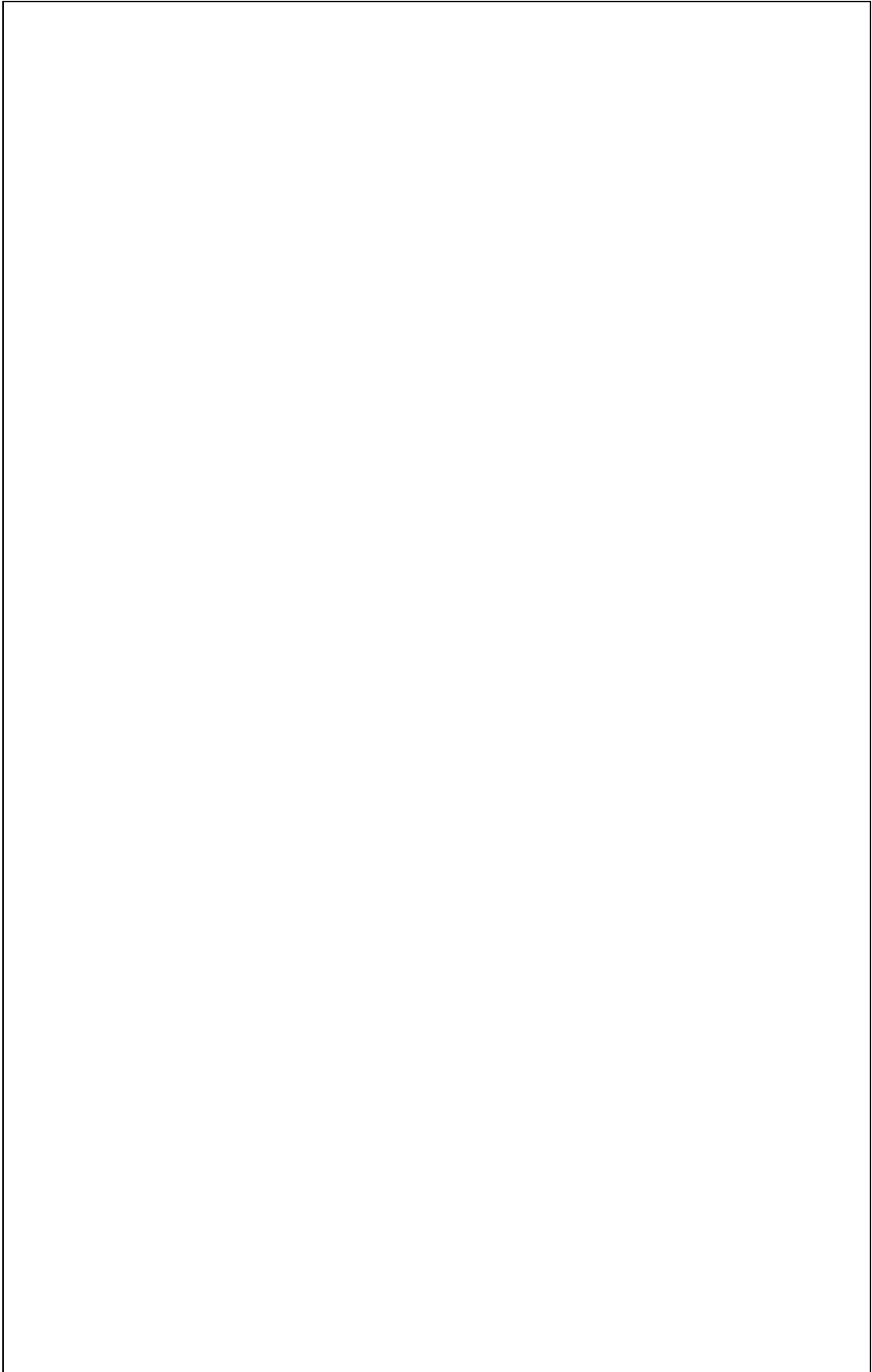
Este programa se comporta como un emisor de eventos e ilustra algunos aspectos más:

- El evento “read” en este caso se genera con un argumento adicional (el título de la novela a leer). El código necesario para ello se muestra en las líneas:
 - 36: Para determinar cada cuánto se genera ese evento (3 segundos en este ejemplo).
 - 25-31: Mediante una clausura que mantiene el contador del número de eventos generados, se determina qué mensaje acompaña al evento como argumento y se emite el evento.

- 12-14: El “listener” para ese evento debe gestionar un parámetro.

Tomando ese programa como base, se pide que el alumno desarrolle otro programa en el que:

- Se generen los eventos siguientes:
 - “uno”: Cada tres segundos. Sin argumentos.
 - “dos”: Inicialmente cada dos segundos. Sin argumentos.
 - “tres”: Cada diez segundos. Sin argumentos.
 - Haya un *listener* para cada evento generado. Para cada evento, esas funciones tendrán que hacer lo siguiente:
 - “uno”: Escribir la cadena “Evento uno.” En su salida estándar.
 - “dos”: Escribir la cadena “Evento dos.” En su salida estándar si el número de eventos “dos” recibidos hasta ahora es igual o superior al número de eventos “uno”. Cuando eso ya no suceda, escribirá “Hay más eventos de tipo ‘uno’”.
 - “tres”: Escribir un mensaje “Evento tres.” Por salida estándar. Además, con cada ejecución de este manejador, se triplicará la duración del intervalo entre dos eventos consecutivos de tipo “dos”, hasta que dicho valor sea 18 segundos. A partir de ese momento, los eventos de tipo “dos” se programarán cada 18 segundos.
- La operación “setInterval()” retorna un objeto que debe ser empleado como único argumento de “clearInterval()”. Para modificar la frecuencia de un evento, conviene utilizar “clearInterval()” antes de establecer la nueva frecuencia. También puede utilizarse “setTimeout()” en lugar de “setInterval()”, reprogramando al final del listener la próxima emisión del mismo evento con la duración adecuada.



ACTIVIDAD 3

OBJETIVO: Entender las clausuras y el paso de funciones como argumento en JavaScript.

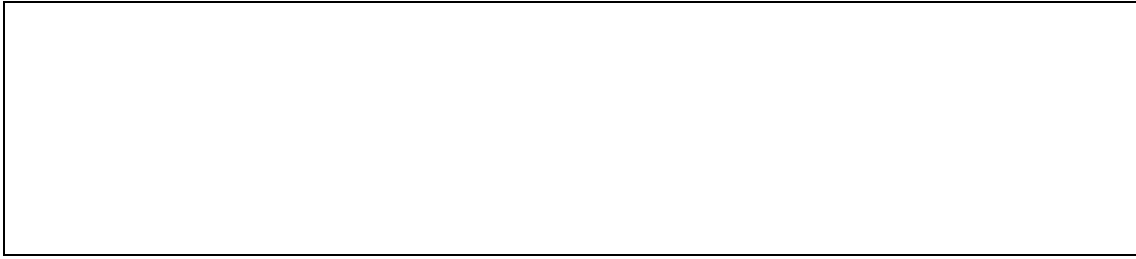
ENUNCIADO: Sea el programa siguiente:

```
1 function a3(x) {  
2     return function(y) {  
3         return x*y  
4     }  
5 }  
6  
7 function add(v) {  
8     let sum=0  
9     for (let i=0; i<v.length; i++)  
10         sum += v[i]  
11     return sum  
12 }  
13  
14 function iterate(num, f, vec) {  
15     let amount = num  
16     let result = 0  
17     if (vec.length<amount)  
18         amount=vec.length  
19     for (let i=0; i<amount; i++)  
20         result += f(vec[i])  
21     return result  
22 }  
23  
24 let myArray = [3, 5, 7, 11]  
25 console.log(iterate(2, a3, myArray))  
26 console.log(iterate(2, a3(2), myArray))  
27 console.log(iterate(2, add, myArray))  
28 console.log(add(myArray))  
29 console.log(iterate(5, a3(3), myArray))  
30 console.log(iterate(5, a3(1), myArray))
```

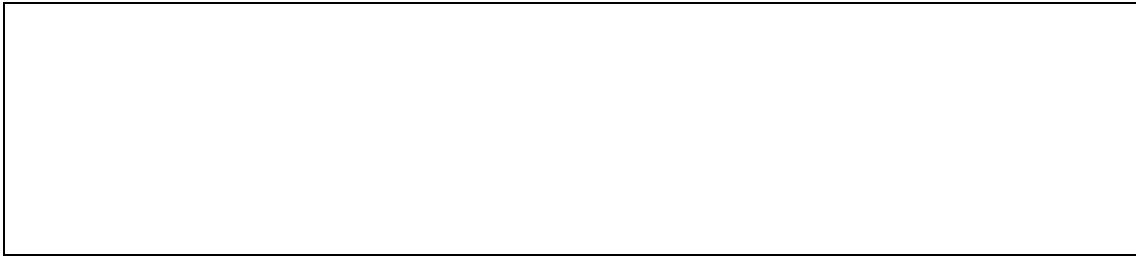
Ejecute ese programa y diga cuál es el resultado de la ejecución de cada una de las líneas siguientes, justificando por qué se da en cada caso:

a) línea 25.

b) línea 26.



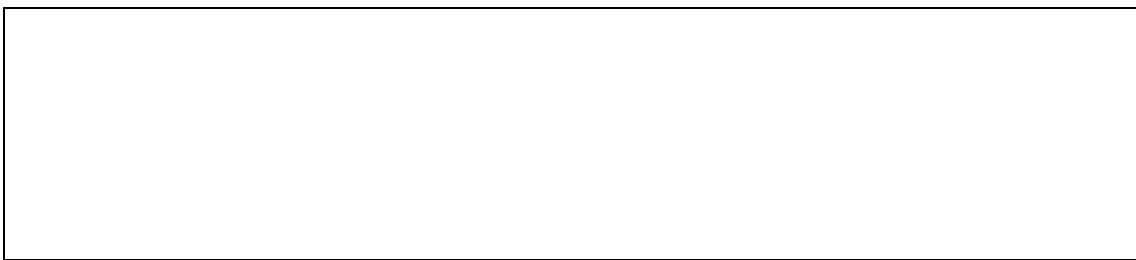
c) línea 27.



d) línea 28.



e) línea 29.



f) línea 30.



ACTIVIDAD 4

OBJETIVO: Gestionar adecuadamente la ejecución de operaciones asincrónicas.

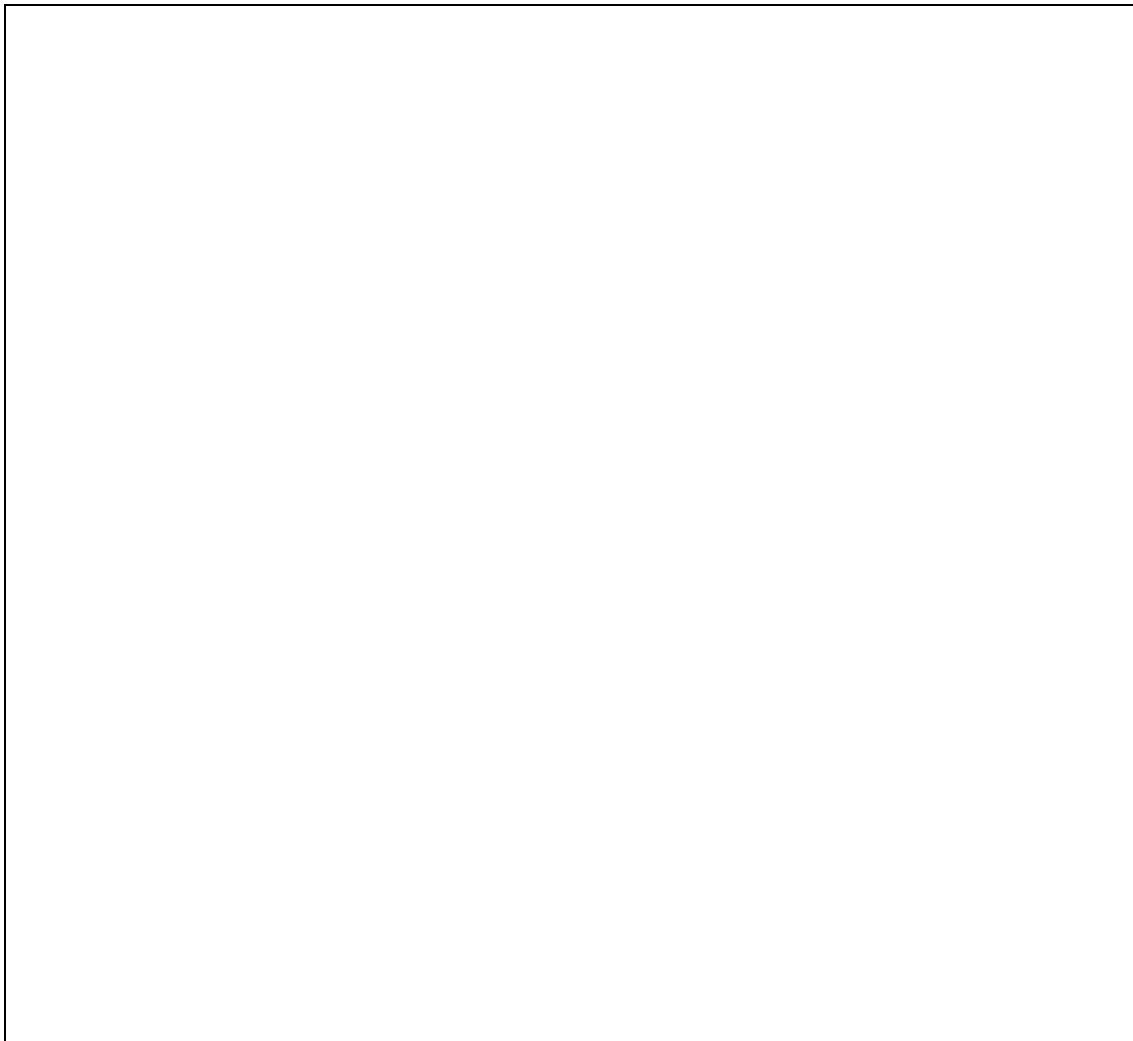
ENUNCIADO: Debemos desarrollar una versión sencilla de la orden `cat` de UNIX. Esa orden muestra en la salida estándar el contenido de una secuencia de ficheros. Los nombres de esos ficheros se reciben como argumentos en la línea de órdenes. No hay límite para el número de nombres de fichero que podrán recibirse en una ejecución de esta orden.

Asumiremos que los ficheros contienen siempre texto. La salida proporcionada debe respetar el orden en el que los nombres de fichero se hayan especificado en la línea de órdenes. Eso implica que la salida del fichero i -ésimo no debe ser mostrada mientras no se hayan visualizado ya todas las líneas del fichero $i-1$.

Desarrollaremos dos versiones de este programa:

- La primera utilizará la operación `fs.readFileSync()`.
- La otra utilizará `fs.readFile()`.

Debe compararse el rendimiento de ambas versiones en caso de que se procese una gran cantidad de ficheros.



ACTIVIDAD 5

OBJETIVO: Entender que los “callbacks” no siempre son asíncronos.

ENUNCIADO: Sólo hay un hilo de ejecución en Node.js. Esto implica que no tendremos por qué preocuparnos sobre la protección de variables compartidas con locks o cualquier otro mecanismo de control de concurrencia.

Sin embargo, hay algunos casos en los que necesitaremos ser cuidadosos.

Un buen principio para razonar sobre la lógica de un programa asíncrono es considerar que TODOS sus callbacks se ejecutarán en un turno posterior al que ahora ejecuta el código que los pasa como argumentos.

Considere el código que se muestra a continuación:

```
1  const fs = require('fs')
2  const path = require('path')
3  const os = require('os')
4  var rolodex={}
5
6  function contentsToArray(contents) {
7      return contents.split(os.EOL)
8  }
9  function parseArray(contents,pattern,cb) {
10     for(let i in contents) {
11         if (contents[i].search(pattern) > -1)
12             cb(contents[i])
13     }
14 }
15
16 function retrieve(pattern,cb) {
17     fs.readFile("rolodex", "utf8", function(err,data){
18         if (err) {
19             console.log("Please use the name of an existant file!!")
20         } else {
21             parseArray(contentsToArray(data),pattern,cb)
22         }
23     })
24 }
25
26 function processEntry(name, cb) {
27     if (rolodex[name]) {
28         cb(rolodex[name])
29     } else {
30         retrieve( name, function (val) {
31             rolodex[name] = val
32             cb(val)
33         })
34     }
35 }
```

```

36
37 function test() {
38     for (let n in testNames) {
39         console.log ('processing ', testNames[n])
40         processEntry(testNames[n], function generator(x) {
41             return function (res) {
42                 console.log('processed %s. Found as: %s', testNames[x], res)
43             }(n))
44         }
45     }
46
47     const testNames = ['a', 'b', 'c']
48     test()

```

Cuando sea ejecutado², esperaremos esta salida:

```

processing a
processing b
processing c
processed a...
processed b...
processed c...

```

TODOS los mensajes “processed” aparecen tras TODOS los mensajes “processing”, tal como se esperaba (los “callbacks” parecen ser llamados en turno futuro).

Sin embargo, considere esta variación:

Sustituya la línea 4 del código anterior (var rolodex={};) por la siguiente:

```

4 var rolodex={a: "Mary Duncan 666444888"};

```

La salida que obtendríamos sería:

```

processing a
processed a...
processing b
processing c
processed b...
processed c...

```

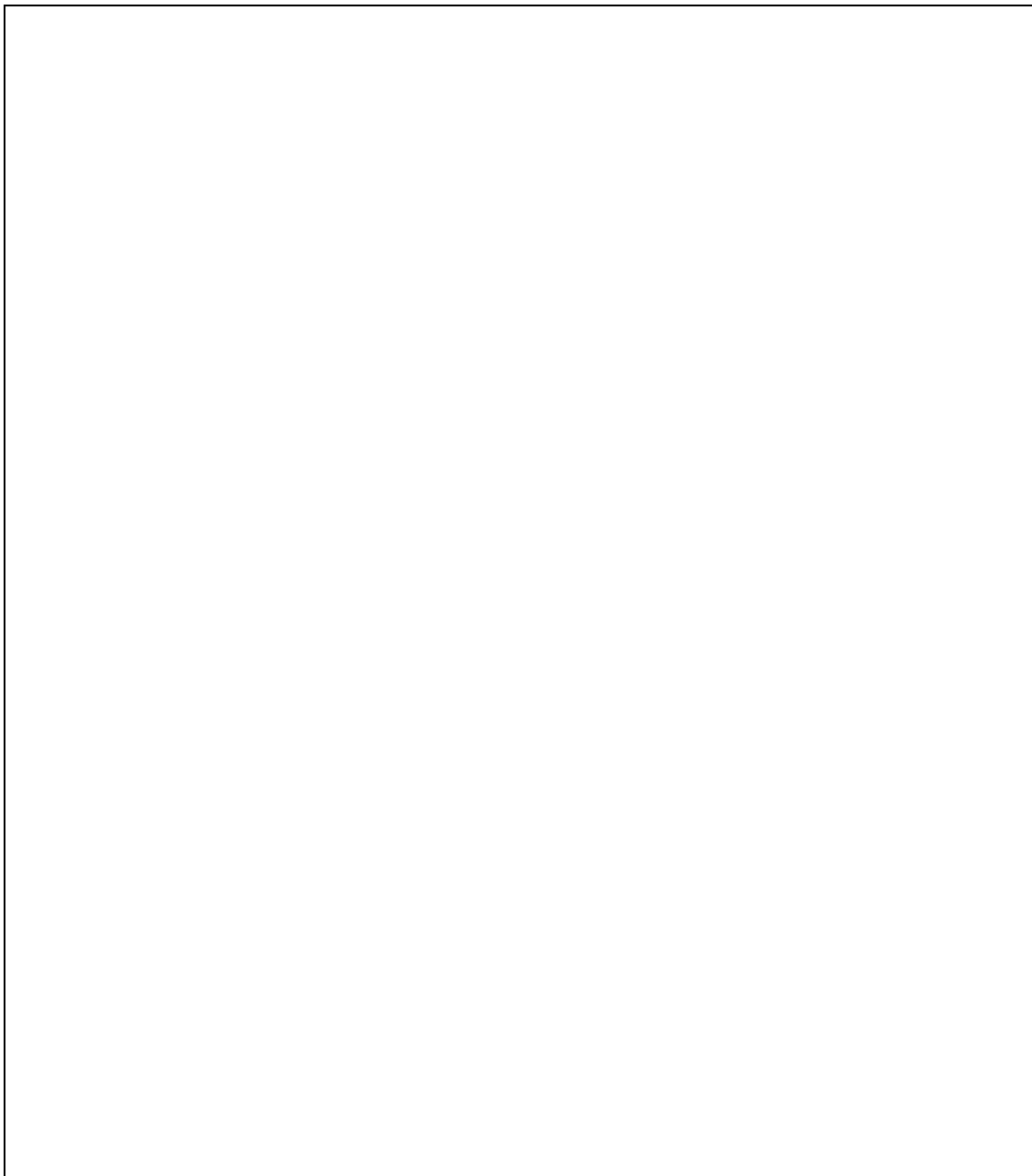
² Para ejecutar el programa correctamente deberá existir un fichero llamado “rolodex” en el mismo directorio. Ese fichero debe contener algunas líneas de texto. En ellas se buscarán las cadenas contenidas en el vector “testNames”.

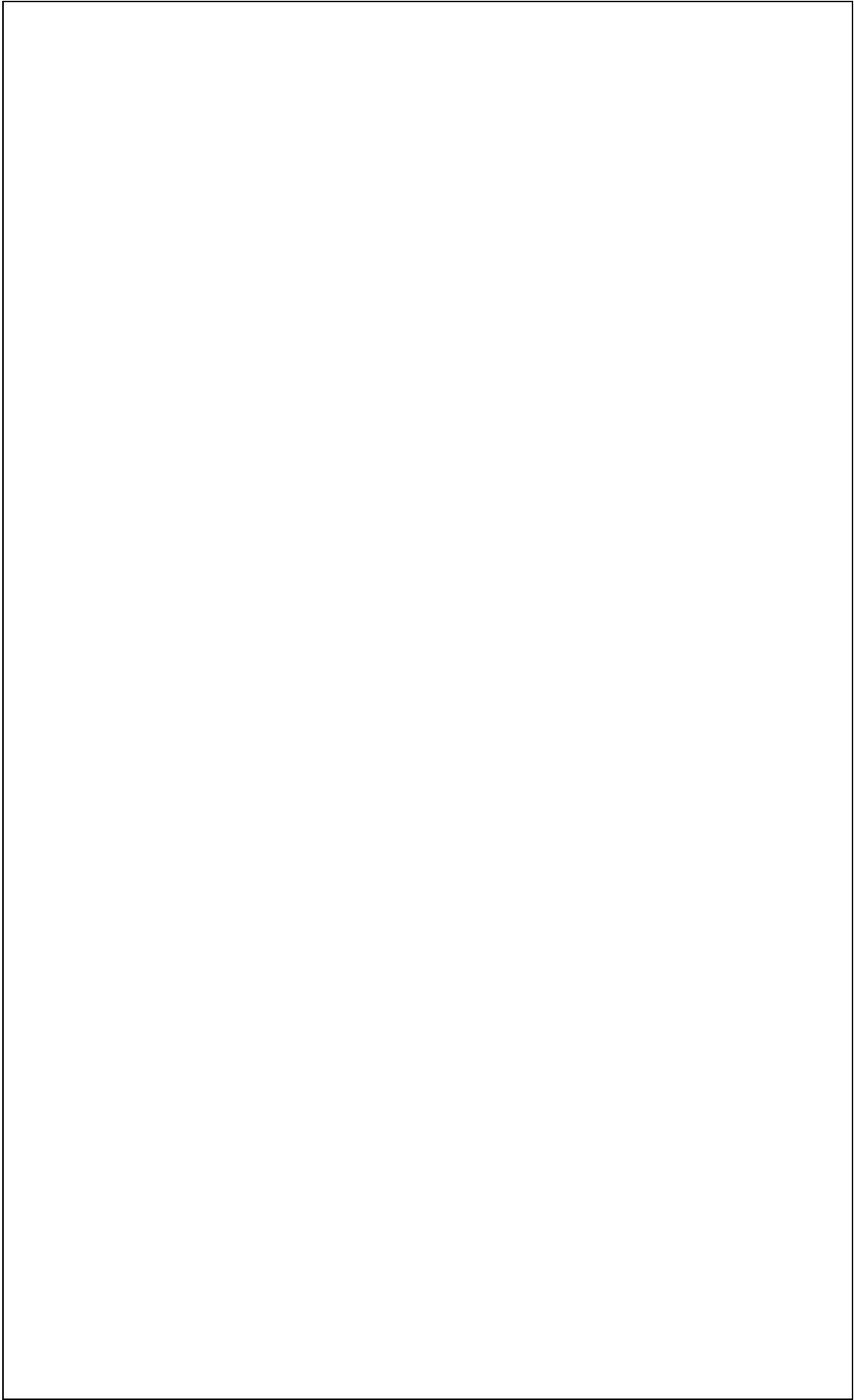
Observe que ahora NO TODOS los mensajes “processed” se muestran tras TODOS los mensajes “processing”. La razón es que uno de los callbacks ha sido ejecutado EN EL MISMO turno que la función que lo pasó.

Dependiendo de la situación, esto podría introducir problemas difíciles de percibir, en caso de que el código que establece los callbacks y uno o más de los callbacks interfirieran en su acceso a una misma parte del estado del proceso, dejándolo inconsistente.

Esta situación (inconveniente) siempre aparecerá si los callbacks confían en que su invocador preparará sus contextos antes de que ellos inicien su ejecución.

Modifique el programa anterior, utilizando promesas (pues los callbacks que procesan la resolución de una promesa siempre lo hacen en un turno posterior), para garantizar que se respete el orden de ejecución esperado.





ACTIVIDAD 6

OBJETIVO: Utilizar adecuadamente “callbacks” y clausuras.

ENUNCIADO: Para acceder a ficheros, Node.js proporciona el módulo “fs”. Escriba un programa que reciba un número variable de nombres de ficheros desde la línea de órdenes y que escriba en pantalla el nombre del fichero más grande de todos ellos, así como su longitud en bytes. Para ello, utilice la función `fs.readFile()` del módulo “fs”, cuya documentación está disponible en https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback. No utilice las variantes sincrónicas de esa función u otras funciones del módulo “fs”.

Para gestionar los argumentos recibidos desde la línea de órdenes tendrá que utilizar el vector `process.argv` (https://nodejs.org/api/process.html#process_process_argv).

Si se reciben varios argumentos, necesitará utilizar clausuras para que el “callback” acceda correctamente a la posición adecuada del vector `process.argv`.

ACTIVIDAD 7

OBJETIVO: Gestionar adecuadamente una secuencia de operaciones asincrónicas. Usar clausuras cuando resulte necesario.

ENUNCIADO: Vamos a desarrollar una versión sencilla de la orden **grep** de los sistemas UNIX. Para ello debemos escribir un programa Node.js que acepte al menos dos argumentos desde la línea de órdenes. Su primer argumento será la palabra a buscar y todos los demás argumentos serán los nombres de múltiples ficheros de texto en los que se buscará la palabra mencionada. Cada vez que encontremos esa palabra a buscar, nuestro programa deberá mostrar en su salida estándar una línea con la información siguiente:

nombre-de-fichero : número-de-línea : contenido-de-esa-línea

El programa debe aceptar una lista de nombres de fichero de cualquier longitud y debe utilizar `fs.readFile()` para leer cada fichero.

NOTA: Usa la operación “[split\(\)](#)” para dividir el contenido del fichero en un vector de líneas de texto y la operación “[includes\(\)](#)” para averiguar si la palabra a buscar está contenida en una línea. Observa que ambas operaciones son métodos de la clase [String](#) en JavaScript.

ACTIVIDAD 8

OBJETIVO: Ejecutar código de manera interactiva en un servidor remoto usando los módulos “net” y “repl”.

ENUNCIADO: EL módulo REPL (Read-Eval-Print Loop) representa el shell de Node. El shell se puede activar directamente en la línea de comandos de una terminal escribiendo:

```
> node
```

Además, el módulo “repl”, si se usa en el código de un programa Node, permite invocar el shell y ejecutar sentencias interactivamente. Considere el siguiente programa:

```
1  /* repl_show.js */
2
3  const repl = require('repl')
4  let f = function(x) {console.log(x)}
5  repl.start('$> ').context.show = f
```

Un ejemplo de ejecución de este programa:

```
> node repl_show
$> show(5*7)
35
undefined
$> show('juan '+'luis')
juan luis
undefined
```

Como se aprecia, en el contexto del shell invocado con “repl” existe una función referenciada como “show” que es equivalente a “console.log”. En esta ejecución también se advierte la aparición del valor “undefined”, esto se puede evitar estableciendo una propiedad del módulo (consultar <https://nodejs.org/api/repl.html>).

La shell del módulo “repl” puede ser invocada remotamente. Considere el código de los programas, repl_client.js y repl_server.js, que se muestran a continuación:

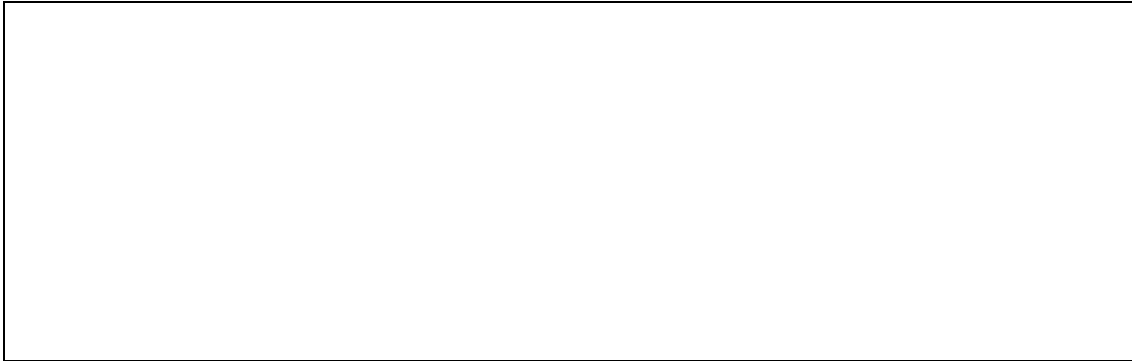
```
1  /* repl_client.js */
2
3  const net = require('net')
4  const sock = net.connect(8001)
5
6  process.stdin.pipe(sock)
7  sock.pipe(process.stdout)
```

```
1  /* repl_server.js */
2
3  const net = require('net')
4  const repl = require('repl')
5
6  net.createServer(function(socket){
7    repl
8    .start({
9      prompt: '>',
10     input: socket,
11     output: socket,
12     terminal: true
13   })
14   .on('exit', function(){
15     socket.end()
16   })
17 }).listen(8001)
```

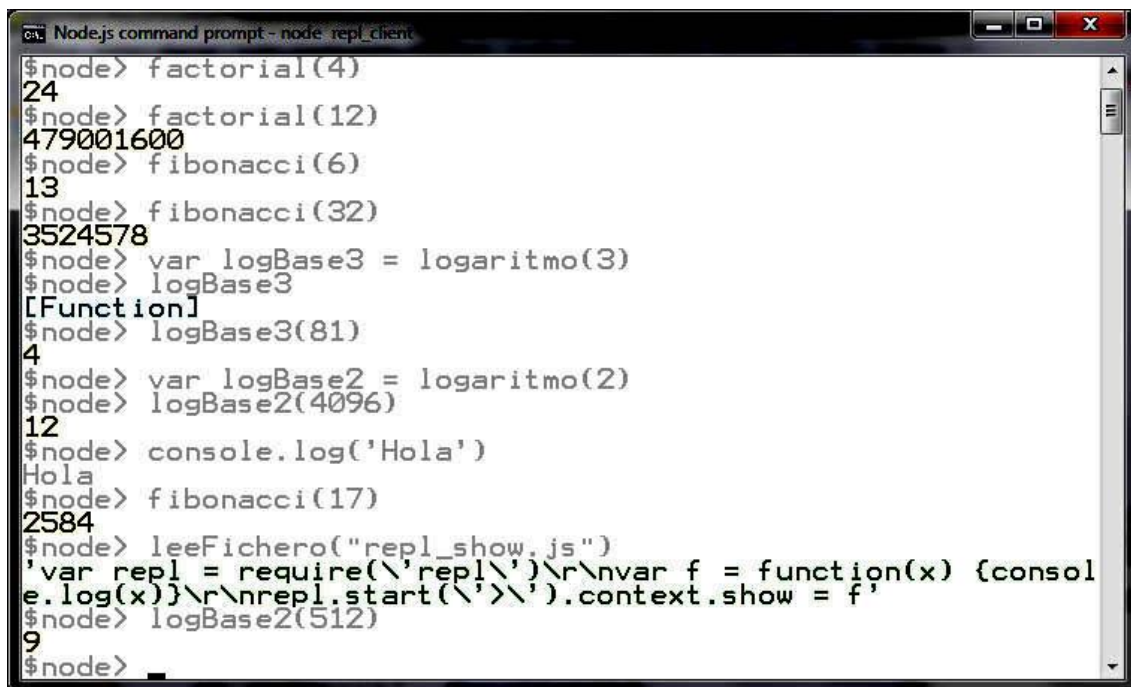
Se pide estudiar el funcionamiento de estos programas, consultando, si es preciso, la API de los módulos utilizados, y ejecutando los programas, interactuando con el shell remoto accesible desde el programa cliente.

- a) Explique el funcionamiento de cliente y servidor, precisando cómo fluye la información y dónde se realiza el procesamiento.

- b) Si en la terminal donde se ejecuta el cliente se escribe `“console.log(process.argv)”`, ¿qué se obtiene?, ¿por qué?



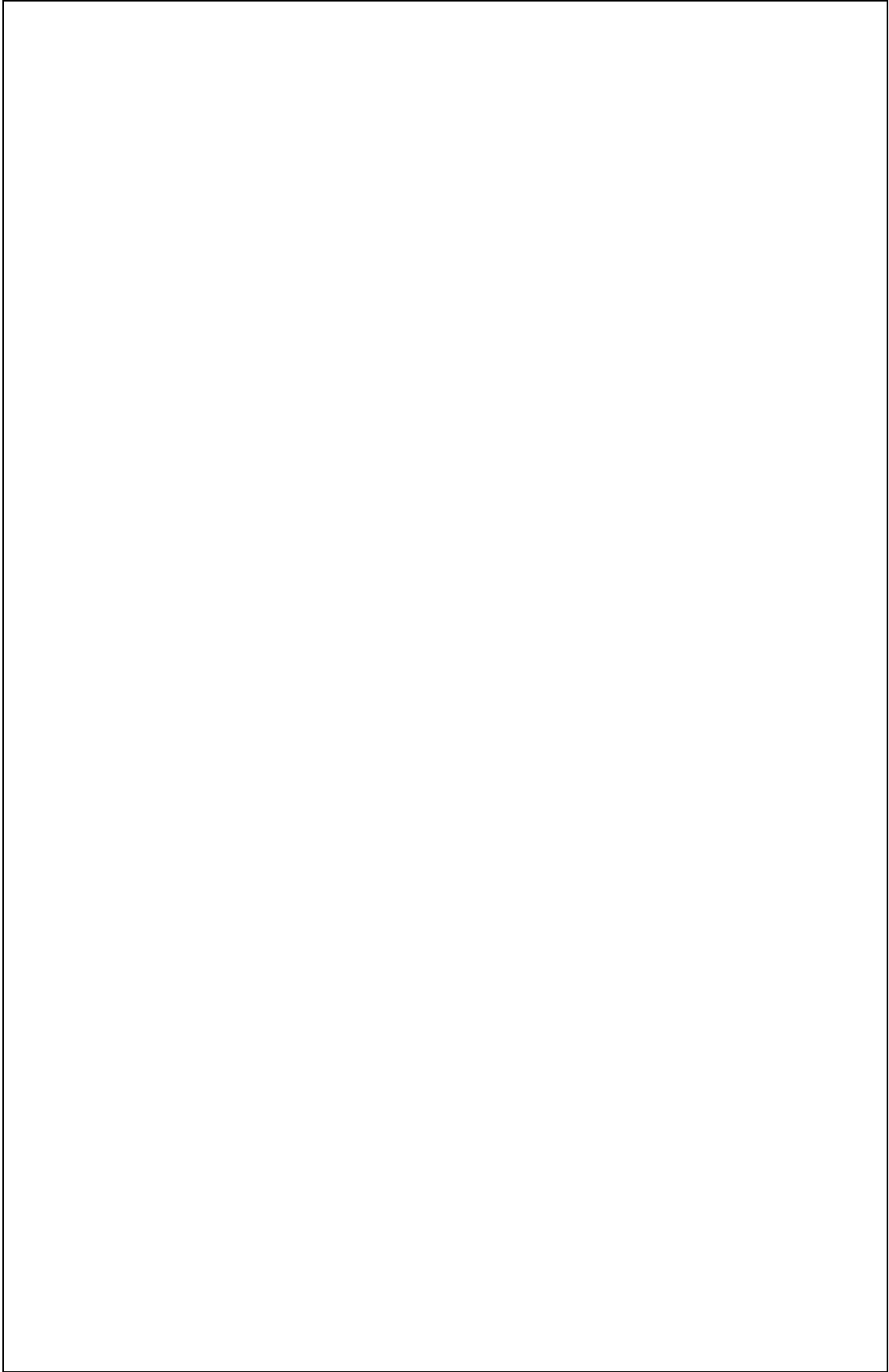
- c) Modifique el programa `“repl_server.js”` para que, desde una terminal donde se ejecute el programa `repl_client` se pueda desarrollar una sesión interactiva como la mostrada en la siguiente figura:



```
Node.js command prompt - node repl_client
$node> factorial(4)
24
$node> factorial(12)
479001600
$node> fibonacci(6)
13
$node> fibonacci(32)
3524578
$node> var logBase3 = logaritmo(3)
$node> logBase3
[Function]
$node> logBase3(81)
4
$node> var logBase2 = logaritmo(2)
$node> logBase2(4096)
12
$node> console.log('Hola')
Hola
$node> fibonacci(17)
2584
$node> leeFichero("repl_show.js")
'var repl = require('\repl\')\r\nvar f = function(x) {consol
e.log(x)}\r\nrepl.start('\>').context.show = f'
$node> logBase2(512)
9
$node> _
```

No se debe modificar el programa cliente. En el servidor se tendrán que modificar los parámetros necesarios (cambio del prompt, supresión de “undefined”, activación de colores) así como añadir a su contexto las funciones necesarias:

- **factorial**: función tal que dado n devuelve $n!$
- **fibonacci**: función tal que dado n devuelve el término n -ésimo de la sucesión de Fibonacci.
- **logaritmo**: función tal que dado n devuelve una función para calcular el logaritmo en base n (consultar el apartado 2.4.3, Clausuras, de la guía de estudio de este tema).
- **leeFichero**: función tal que dado un nombre de fichero, si existe el fichero, devuelve su contenido como String (ayuda: usar la función `“readFileSync”` del módulo `“fs”`). Si el fichero no existe, se generará el error `ENOENT`.



ACTIVIDAD 9

OBJETIVO: Comunicación en una aplicación multiusuario en red, y uso del módulo “Socket.IO”. En particular, se pretende comunicar mediante sockets todas las instancias de una sencilla aplicación ejecutable en red (cada instancia corresponde a un usuario activo) de manera que todos los usuarios de la aplicación puedan colaborar en un objetivo común.

ENUNCIADO: Considérese la siguiente aplicación de dibujo en una página web, constituida por un fichero HTML, “index.html”, donde se carga un fichero js, “script.js”. Los ficheros son:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <title>Node.js Multiuser Drawing Game</title>
6   </head>
7   <body>
8     <div id="cursors">
9       <!-- The mouse pointers will be created here -->
10    </div>
11    <canvas id="paper" width="800" height="400"
12      style="border:1px solid #000000;">
13      Your browser needs to support canvas for this to work!
14    </canvas>
15    <hgroup id="instructions">
16      <h1>Draw anywhere inside the rectangle!</h1>
17      <h2>You will see everyone else who's doing the same.</h2>
18      <h3>Tip: if the stage gets dirty, simply reload the page</h3>
19    </hgroup>
20    <!-- JavaScript includes. -->
21    <script src="http://code.jquery.com/jquery-1.8.0.min.js"></script>
22    <script src="script.js"></script>
23  </body>
24 </html>
```

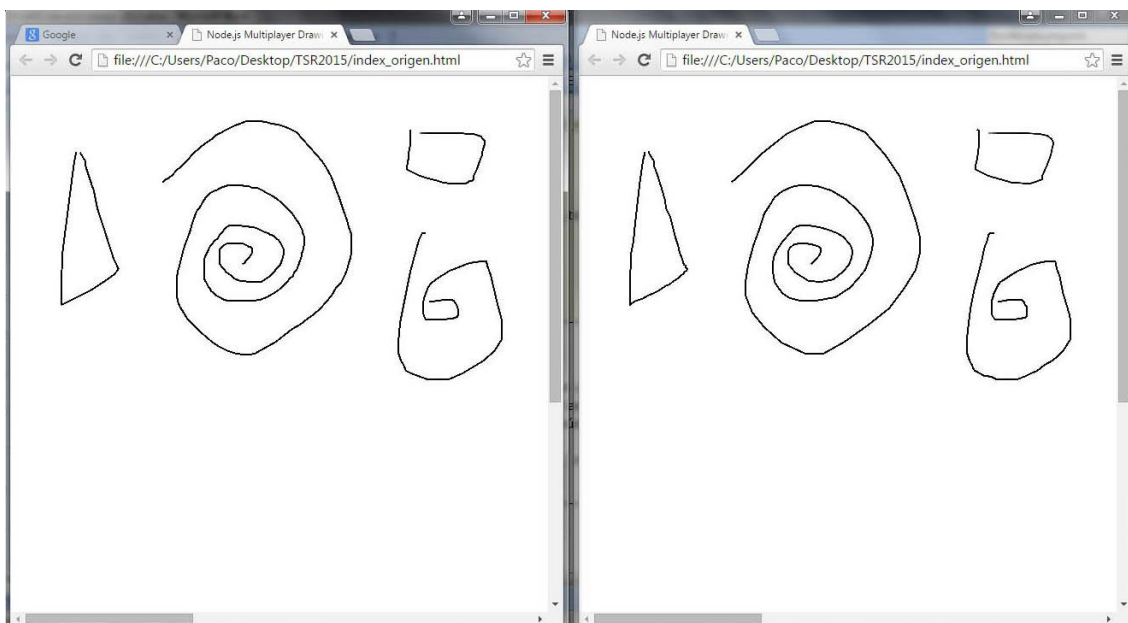
```
1 $(function(){
2   // This demo depends on the canvas element
3   if(!('getContext' in document.createElement('canvas'))){
4     alert('Sorry, it looks like your browser does not support canvas!');
5     return false;
6   }
7   let doc = $(document),
8       win = $(window),
9       canvas = $('#paper'),
10      ctx = canvas[0].getContext('2d'),
11      instructions = $('#instructions'),
12      id = Math.round($.now()*Math.random()), // Generate an unique ID
13      drawing = false, // A flag for drawing activity
14      clients = {},
15      cursors = {},
16      prev = {};
```

```

17 canvas.on('mousedown',function(e){
18     e.preventDefault();
19     drawing = true;
20     prev.x = e.pageX;
21     prev.y = e.pageY;
22 });
23 doc.bind('mouseup mouseleave',function(){
24     drawing = false;
25 });
26 doc.on('mousemove',function(e){
27     // Draw a line for the current user's movement
28     if(drawing){
29         drawLine(prev.x, prev.y, e.pageX, e.pageY);
30         prev.x = e.pageX;
31         prev.y = e.pageY;
32     }
33 });
34 function drawLine(fromx, fromy, tox, toy){
35     ctx.moveTo(fromx, fromy);
36     ctx.lineTo(tox, toy);
37     ctx.stroke();
38 }
39 });

```

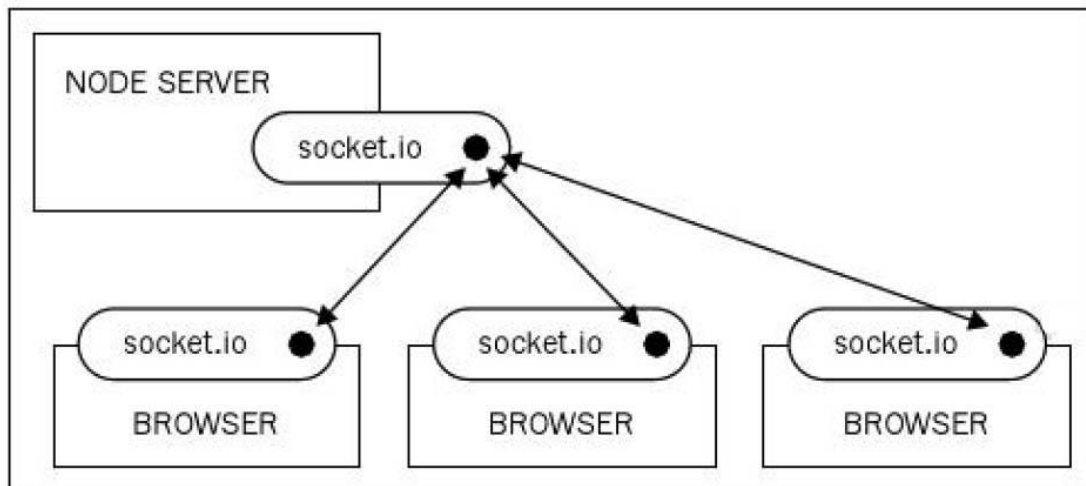
Esta aplicación, usando un objeto canvas y dando respuesta a los eventos de ratón, permite al usuario dibujar con trazo libre (véase, como ejemplo, la siguiente figura), pero es una aplicación monousuario. El objetivo de esta actividad es modificarla de forma que múltiples usuarios puedan cargarla en sus navegadores y el dibujo sea compartido (es decir, los trazos efectuados por cualquier usuario en su lienzo de dibujo se muestren inmediatamente en los lienzos del resto de usuarios conectados). La figura muestra un ejemplo con dos usuarios, pero la solución al problema no estará limitada a un número máximo de usuarios.



En la solución considerada se utilizará el módulo “Socket.IO”. Como no es un módulo estándar, preinstalado, se tendrá que instalar, mediante la orden:

```
> npm install socket.io
```

“Socket.IO” proporciona sockets bidireccionales, y se integra adecuadamente con los diferentes navegadores de Internet. Un diseño adecuado para la aplicación multiusuario considerada se muestra en la siguiente figura:



Es un diseño adecuado dado que se requiere que cada “browser” (navegador de cliente) comunique con un “node server” (servidor) al que envía sus acciones (trazos de dibujo) y del que recibe las acciones de los demás clientes. Como se aprecia, el cometido del servidor será retransmitir los mensajes recibidos de cada cliente al resto de clientes. Este tipo de comunicación se conoce como “broadcasting”.

Usando el módulo “Socket.IO”, para retransmitir mensajes hay que añadir el flag “broadcast” en las llamadas a los métodos “emit” y “send”. Por ejemplo, el siguiente fragmento de código correspondería a un servidor que retransmitiera mensajes a todos, excepto al socket que los envió:

```
1 const io = require('socket.io').listen(8080);  
2  
3 io.on('connection', function (socket) {  
4   socket.broadcast.emit('user connected');  
5 });
```

Para la solución de la actividad, es necesario implementar un nuevo módulo, el servidor, y modificar el fichero cliente, “script.js”, y la página web que lo carga. En esta última, “index.html” solamente se necesita añadir la siguiente línea en la sección de includes:

```
<script src="socket.io.js"></script>
```

Suponiendo que el fichero “socket.io.js” esté en el mismo directorio que “index.html”.

El servidor estará a la escucha en un determinado puerto, al que se conectará el socket (tipo “socket.io”) de cada script cliente. Las modificaciones a efectuar en “script.js” serán las siguientes:

- Declarar el socket y conectarlo al servidor.
- Modificar la función de callback del documento (variable “doc”) cuando se produce el evento “mousemove” para que, además de seguir dibujando el trazo del usuario local, envíe la información de dicho trazo a través del socket. La información a transmitir sería:

```
{ 'x': e.pageX, 'y': e.pageY, 'drawing': drawing, 'id': id }
```

Y el evento asociado a este envío podría ser “mousemove” (si se elige también este nombre de evento para ser escuchado en el servidor).

- Por otra parte, el socket del cliente deberá estar a la escucha de los envíos del servidor. Estos envíos corresponderán a notificaciones de trazos de dibujo hechos por otros usuarios. Tendrán que tener un nombre de evento, por ejemplo, “moving”. La función de callback asociada a este evento “moving” en el socket del cliente deberá procesar adecuadamente los datos recibidos (el correspondiente objeto “data” tendrá las propiedades “x”, “y”, “drawing” e “id”, dado el formato indicado antes).

En este callback, en primer lugar, deberá comprobarse si los datos proceden de un usuario nuevo para, en tal caso, registrarlo:

```
if ( !(data.id in clients) )  
    cursors[data.id] = $('<div class="cursor">').appendTo('#cursors');
```

A continuación, se dibujará el trazo correspondiente a los datos recibidos (desde la última posición del usuario, “clients[data.id]”, hasta su nueva posición, “data”):

```
if ( data.drawing && clients[data.id] )  
    drawLine(clients[data.id].x, clients[data.id].y, data.x, data.y);
```

Por último, el callback actualizará el estado del usuario:

```
clients[data.id] = data;
```

Se pide implementar en el script del cliente, “script.js”, las modificaciones descritas:

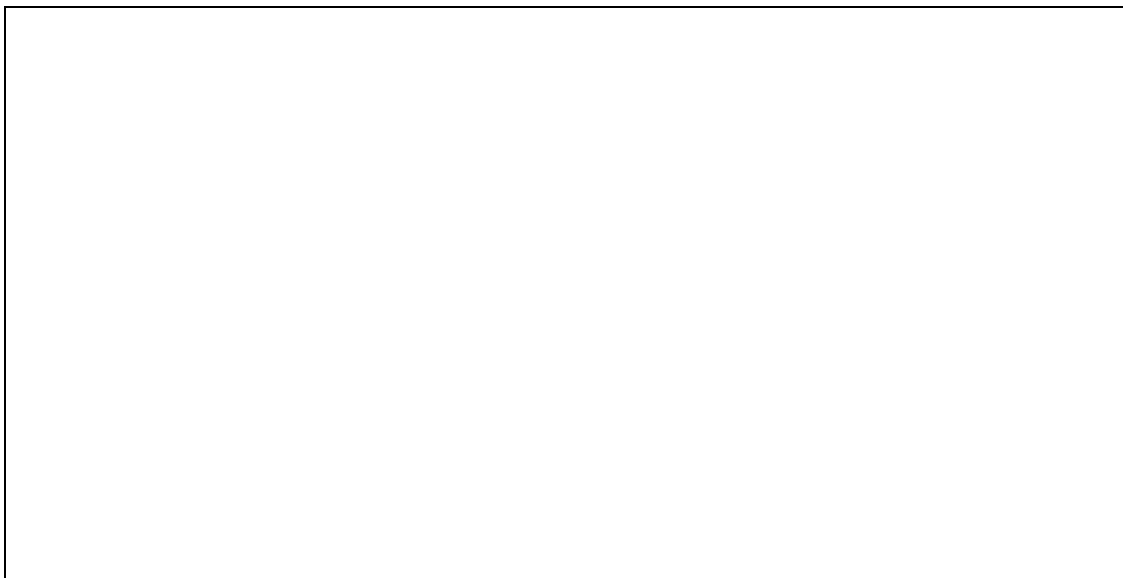




En el servidor se ha de escribir el código necesario para:

- Establecer un socket (tipo “socket.io”) que escuche en el mismo puerto indicado en el script cliente.
- Para el objeto “io.sockets”, implementar un callback que responda al evento “connection” por parte de algún cliente.
- En este callback, para el socket identificado del cliente, implementar otro callback que responda al evento “mousemove”.
- En respuesta a este evento “mousemove”, se transmitirá (con *broadcasting*) un mensaje con el evento “moving” y los mismos datos que acompañen al evento “mousemove”.

Se pide implementar el servidor:



ACTIVIDAD 10

OBJETIVO: Introducir una gestión asíncrona en programas que inicialmente tenían un comportamiento sincrónico. Utilizar adecuadamente *callbacks* (o, alternativamente, promesas) para este fin.

ENUNCIADO: En este ejercicio aplicaremos los conceptos y las construcciones relacionadas con la asincronía en NodeJS. El programa a desarrollar, en una primera aproximación, debe leer el contenido de un fichero de texto (usando el módulo fs), ordenar alfabéticamente sus líneas, y escribir el resultado en un nuevo fichero. Se incorporarán gradualmente características relacionadas con el sincronismo y el número de ficheros a ordenar. Los aspectos o combinaciones complejas se resuelven a partir de otras soluciones más sencillas; por ello se organiza el ejercicio en 5 etapas:

1. Versión **síncrona**: $Ls \rightarrow O \rightarrow Es$ (Ls =Lectura síncrona, O =Ordenación, Es =Escritura síncrona). Es el punto de partida, para un solo fichero.
2. Versión "**semisíncrona**": $Ls \rightarrow O \rightarrow E \rightarrow F$ donde E es asíncrono y se requiere una operación F final.
3. Versión **asíncrona**: $L \rightarrow O \rightarrow E \rightarrow F$ donde L también es asíncrono. Puede proponerse una alternativa orientada a eventos.
4. **Asíncrona con dos ficheros**. Habrá 2 ramas $Li \rightarrow Oi \rightarrow Ei \rightarrow Fi$ ($1 \leq i \leq 2$) independientes entre sí. Añadir un **mensaje** adicional cuando **ambos terminen**.
5. **Generalizar** para un número indeterminado de ficheros.

El enunciado completo (en la página siguiente) añade algunos detalles y puntualizaciones. Para no desviarnos del tema de interés, no se contempla la posibilidad de error. La ordenación del fichero tampoco es preocupante (usar la función `sort_lines()` cuyo código se encuentra en el ejemplo de solución del primer apartado).

Etapas

1. (ya resuelto) Versión **síncrona**: **Ls->O->Es** (**Ls**=Lectura síncrona, **O**=Ordenación, **Es**=Escritura síncrona). Se toman las variantes síncronas de la lectura y escritura en archivos. E no comienza hasta que finaliza O, O no comienza hasta que finaliza L.

Solución:

```
01: // función sort_lines
02: var fs=require('fs')
03: fich=process.argv[2]
04:
05: function sort_lines(mystring)
06: { return mystring.toString().split("\n").sort().join("\n")}
07:
08: var r=fs.readFileSync(fich, 'utf-8')
09: fs.writeFileSync(fich+"2", sort_lines(r), 'utf-8')
```

2. Versión "**semisíncrona**": La escritura será asíncrona, de manera que, para saber cuándo ha finalizado, añadiremos una operación F al final console.log("Fin");), tras la escritura. No usamos sufijo para asíncrono. La ordenación será **Ls->O->E->F**.
3. Versión **asíncrona**: **L->O->E->F** La lectura también será asíncrona, de forma que la relación de precedencia obliga a que (O+E) sea callback de L, y que F siga siendo callback de E. Esta transformación de secuencialidad en anidamiento puede ser un problema.
 - Como *alternativa*, puede orientarse a **eventos**, y que la finalización de L se represente como evento que active a (O+E), que la finalización de E sea un evento que active a F.
4. **Dos ficheros**. Ahora habrá 2 ramas **Li->Oi->Ei->Fi** ($1 \leq i \leq 2$). El planteamiento supone que una rama NO tiene dependencia respecto a la otra, y esa independencia debe ser aprovechada por la solución. Esto supone que, p.ej. una ordenación **L1-L2-O2-E2-O1-F2-E1-F1** es tan válida como **L2-L1-O2-O1-E2-E1-F2-F1** o cualquier otro intercalado que respete las relaciones expresadas para cada rama. La solución no puede limitar esto porque perdería rendimiento; ¿puedes argumentarlo?, ¿y poner contraejemplos?
Añadir un **mensaje adicional de finalización** cuando ambas ramas finalicen (habrá un "Fin 1", "Fin 2" y "Fin de todos").
5. **Generalizar** para un número indeterminado de ficheros. Si en el apartado anterior has creado una solución para dos ficheros exactamente, posiblemente hayas optado por una versión "artesana". Cambiar a 3, 4 ó 5 ficheros puede poner a prueba esa artesanía, pero generalizar de la forma indicada supone cambiar de planteamiento. ¿Un vector de funciones ...? Hay que imaginarlo antes de escribirlo. Recuerda que aún necesitas el mensaje de finalización. Alternativamente puede ser resuelto mediante promesas.

ACTIVIDAD 11

OBJETIVO: Utilizar adecuadamente las operaciones del módulo **net**.

ENUNCIADO: Considérense los tres ficheros siguientes:

```
1 // file: proxy.js
2 const net = require('net')
3
4 const LOCAL_PORT = 8000
5 let remotePort = process.argv[3] || 8001
6 let remoteIP = process.argv[2] || '127.0.0.1'
7
8 const server = net.createServer(function (socket) {
9   const serviceSocket = new net.Socket()
10  serviceSocket.connect(parseInt(remotePort),
11    remoteIP, function () {
12      socket.on('data', function (msg) {
13        serviceSocket.write(msg)
14      })
15      serviceSocket.on('data', function (data) {
16        socket.write(data)
17      })
18    })
19 }).listen(LOCAL_PORT)
20 console.log("TCP server accepting connection on port: " + LOCAL_PORT)
```

```
1 // File: worker.js
2 const net = require('net')
3
4 const server = net.createServer(
5   function(c) { //connection listener
6     console.log('server: client connected')
7     c.on('end',
8       function() {
9         console.log('server: client disconnected')
10      })
11     c.on('data',
12       function(data) {
13         c.write(parseInt(data+"")*3+"")
14       })
15   })
16
17 server.listen(parseInt(process.argv[2]) || 8001,
18   function() { //listening listener
19     console.log('server bound')
20   })
```

```

1 // File: client.js
2 const net = require('net')
3
4 const client = net.connect(parseInt(process.argv[2]) || 8000,
5   function() { //connect listener
6     console.log('client connected')
7     client.write(process.pid+"")
8   })
9
10 client.on('data',
11   function(data) {
12     console.log(data.toString())
13   })
14
15 client.on('end',
16   function() {
17     console.log('client disconnected')
18   })

```

El primer fichero, proxy.js, actúa como un intermediario entre los demás. Normalmente, la comunicación sería iniciada por el proceso cliente, que enviaría un mensaje al proxy. El proxy reenviaría ese mensaje al trabajador. El trabajador procesaría esa petición y devolvería una respuesta al proxy. Finalmente, el proxy devolvería la respuesta al cliente.

En esta versión original, una vez finalicen esas interacciones el proceso cliente no finaliza, pero es incapaz de hacer nada más.

Se plantean los apartados siguientes:

1. Utilizando el código original de los programas, iniciaremos un proceso de cada tipo en este orden: trabajador, proxy y cliente. Cuando el cliente haya mostrado la respuesta recibida, eliminaremos el proxy. ¿Qué les ocurre entonces a los demás procesos?
2. Para que finalice el cliente, ese proceso deberá cerrar su conexión tras recibir la respuesta. Extienda "client.js" para que se comporte de esa manera.
3. Tras ampliar "client.js", describa qué ocurre (es decir, si la ejecución finaliza sin problemas o hay algún error; en caso de errores, describa qué error ocurre y cómo podría evitarse) cuando los tres procesos sean iniciados en estas secuencias:
 - a. Trabajador, proxy, cliente.
 - b. Proxy, trabajador, cliente.
 - c. Proxy, cliente, trabajador.
 - d. Cliente, proxy, trabajador.
4. Amplíe los tres programas para que gestionen el evento 'error' en sus conexiones. Repita el apartado 3 con esos nuevos programas y describa si esos órdenes de ejecución siguen causando los mismos problemas o no. Explique los nuevos comportamientos si hubiera cambios. Tras completar cada secuencia, inicie más clientes en cada una de ellas y compruebe si se comportan adecuadamente o no.
 NOTA: Para "serviceSocket", establezca el *listener* de su evento 'error' antes de conectar con el trabajador.
5. Las conexiones del módulo **net** son "transitorias", pues están basadas en sockets TCP. El tema 3 describe la biblioteca ZeroMQ. ZeroMQ utiliza conexiones (débilmente) "persistentes". En un canal de comunicación persistente los mensajes se pueden enviar incluso antes de que el otro extremo de la comunicación se haya conectado. Describa en líneas generales cómo podrían construirse canales "persistentes" sobre las conexiones TCP del módulo **net**.

ACTIVIDAD 12

OBJETIVO: Ejercicio sobre servidor **net**, objetos y arrays, funciones JSON, módulo **fs**, y eventos periódicos (setInterval).

ENUNCIADO: Se quiere implementar un servidor **net** que reciba resultados electorales, que le serán transmitidos mediante un número indeterminado de clientes **net**. El servidor deberá contabilizar, acumular adecuadamente, y guardar en ficheros toda la información que le sea transmitida.

Cualquier cliente **net** podrá enviar al servidor objetos que contengan los votos obtenidos por cada partido político en un determinado colegio electoral. Un par de ejemplos de esta clase de objetos:

```
{lugar:madrid, pp:3532, psoe:2056, up:3077, cs:1540}
```

```
{lugar:barcelona, pp:1056, psoe:1403, up:2056, cs:1986, ERC:2389}
```

Los objetos tendrán siempre una propiedad **lugar** que identifica la circunscripción electoral, y un número variable de propiedades tales que su identificador es el de un partido político y su valor los votos obtenidos por dicho partido.

Estos objetos, serializados con **JSON.stringify**, son los datos que recibirá el servidor **net** a implementar. El servidor ha de procesar los datos recibidos, guardándolos adecuadamente en memoria y en disco:

- En memoria debe mantener una variable de tipo array, llamémosla **votos**, que use la propiedad **lugar** (de los objetos recibidos) como índice del array, y almacene en la posición del array así indexada un objeto con todos los votos recibidos por cada partido en esa circunscripción.
- En disco, periódicamente (cada 20 segundos), debe guardar un fichero de texto (extensión txt) por cada entrada en el array **votos**. El nombre del fichero será el del índice del array (la propiedad **lugar**) y el contenido del fichero será el valor almacenado en dicha posición del array, serializado con JSON.

A modo de ejemplo, considérese que durante los primeros 20 segundos de ejecución del servidor se han recibido los siguientes datos desde varios clientes **net**:

```
{lugar:madrid, pp:3500, psoe:2000, up:3000, cs:1500}
```

```
{lugar:barcelona, pp:1000, psoe:1500, up:2000, cs:2000, ERC:3000}
```

```
{lugar:madrid, pp:2000, psoe:3000, up:1000, cs:500}
```

```
{lugar:valencia, pp:2500, psoe:1500, up:2000, cs:2500}
```

```
{lugar:madrid, pp:4000, psoe:3000, up:2000, cs:2000}
```

```
{lugar:barcelona, psoe:500, up:400, cs:200, ERC:300}
```

Entonces, la variable **votos** del servidor ha de mantener la siguiente información:

votos['madrid'] = {pp:9500, psoe:8000, up:6000, cs:4000}

votos['barcelona'] = {pp:1000, psoe:2000, up:2400, cs:2200, ERC:3300}

votos['valencia'] = {pp:2500, psoe:1500, up:2000, cs:2500}

Y en disco se habrán escrito los siguientes ficheros:

<i>Nombre fichero</i>	<i>Contenido fichero</i>
madrid.txt	{ "pp":9500, "psoe":8000, "up":6000, "cs":4000 }
barcelona.txt	{ "pp":1000, "psoe":2000, "up":2400, "cs":2200, "erc":3300 }
valencia.txt	{ "pp":2500, "psoe":1500, "up":2000, "cs":2500 }

Se pide implementar el servidor **net**, tomando como base, y manteniendo en la solución, el siguiente código:

```
1 const net = require('net')
2 const fs = require('fs')
3 var votos = {}
4
5 var server = net.createServer(function(c) {
6     c.on('data', function(data){
7         // A COMPLETAR
8     })
9 })
10
11 server.listen(9000,
12 function() { console.log('server bound')
13 })
14
15 function guardar() {
16     // A COMPLETAR
17     console.log('datos volcados a disco')
18 }
19
20 // A COMPLETAR
```

ACTIVIDAD 13

OBJETIVO: Ejercicio sobre callbacks, clausuras, objetos y arrays, funciones JSON, módulo FS, e interacción con process.stdin.

ENUNCIADO: Se quiere implementar un servidor que permita obtener resultados electorales a partir de la lectura de un conjunto de ficheros de texto (donde se guardan dichos datos) y desarrollar una sesión interactiva, con el usuario de la aplicación, para consultar resultados de cada circunscripción electoral.

Se considera que el servidor se ejecuta en un directorio donde se encuentran los ficheros de texto con los resultados electorales. Los nombres y contenidos de los ficheros siguen el formato descrito en el ejercicio anterior, así, por ejemplo:

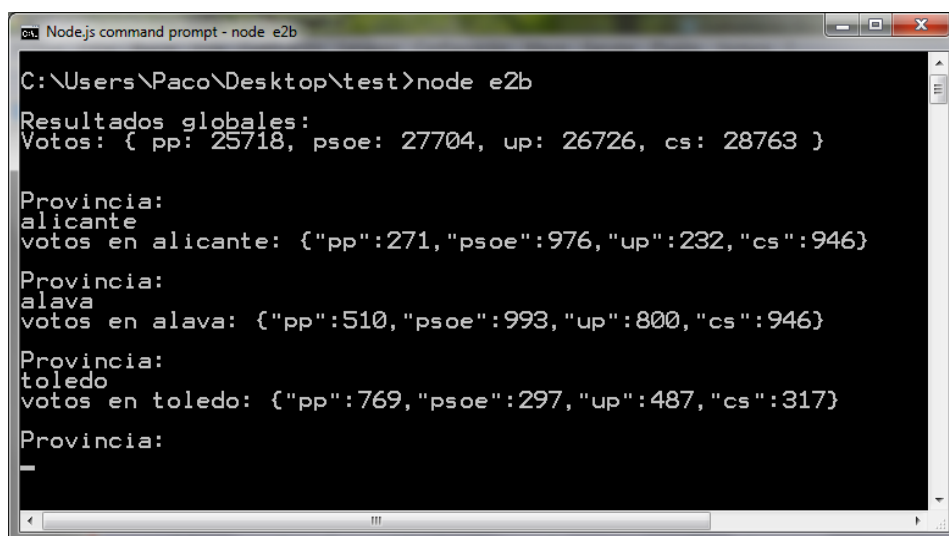
<i>Nombre fichero</i>	<i>Contenido fichero</i>
madrid.txt	{ "pp":9500, "psoe":8000, "up":6000, "cs":4000 }
barcelona.txt	{ "pp":1000, "psoe":2000, "up":2400, "cs":2200, "erc":3300 }
valencia.txt	{ "pp":2500, "psoe":1500, "up":2000, "cs":2500 }

El servidor, en primer lugar, debe leer todos los ficheros, guardando los resultados electorales en un array, llamémoslo **votos**, con el mismo criterio del ejercicio anterior.

El servidor, en segundo lugar (es decir, una vez leídos todos los ficheros), debe:

- Mostrar los resultados globales (el total de votos obtenidos por cada partido en todas las circunscripciones), guardados en otro array llamado **total_votos**.
- Iniciar una sesión interactiva, mediante **process.stdin**, para consultar los resultados en cada provincia o circunscripción.

La siguiente captura de pantalla sirve como referencia de la funcionalidad de la aplicación:



```
Node.js command prompt - node e2b
C:\Users\Paco\Desktop\test>node e2b
Resultados globales:
Votos: { pp: 25718, psOE: 27704, up: 26726, cs: 28763 }

Provincia:
alicante
votos en alicante: { "pp":271, "psOE":976, "up":232, "cs":946 }

Provincia:
alava
votos en alava: { "pp":510, "psOE":993, "up":800, "cs":946 }

Provincia:
toledo
votos en toledo: { "pp":769, "psOE":297, "up":487, "cs":317 }

Provincia:
_
```

En este ejemplo, el usuario ha escrito los nombres de 3 provincias (*alicante*, *alava*, *toledo*), y el resto de la salida ha sido generada por el servidor a implementar.

Se pide implementar el servidor, tomando como base, y manteniendo en la solución, el siguiente código:

```
1 const fs = require('fs')
2 var total_votos = {}
3
4 fs.readdir('.', function (err, files) {
5   var count = files.length
6   var votos = {}
7   for (var i=0; i < files.length; i++) {
8     // A COMPLETAR
9   }
10 })
```