Problemes NodeJS

Question 1 :

Crea una funció **f1** que rebi un paràmetre **a** i que l'escrigui a la consola fent servir la funció **log** de l'objecte **console**.

Solution:
El codi empleat es:

```
f1 = function(a){
    console.log(a);
}
f1(3);
```

Mostra per pantalla:
> 3

Question 2 :

Crea una funció **f2** que rebi un paràmetre **a** i que retorni $2 * a$ si **a** $>= 0$ i -1 en cas contrari.

Solution:
El codi empleat es:

```
f2 = function(a){
    if(a>=0){
        return 2*a;
    }
    else{
        return -1;
    }
}
f2(3)
```

Mostra per pantalla:
> 6

I si executem el següent codi:

```
f2(-2)
```

Mostrarà per pantalla:

> -1

Question 3 :

Crea una funció **f3** que rebi una llista com a primer paràmetre i retorni una llista.

$$llista2 = f3(llista)$$

Cada element **y** de la **llista2** ha de ser el resultat de:

$$f2(x) + 23$$

a cada element **x** de **llista**, on **f2** és la funció de l'exercici anterior.

Solution:
El codi empleat es:

```
llista1 = [1,2,3];
f3 = function(llista){
        var llista2 = [];
        for(i=0; i<llista.length; i++){
                llista2.push(f2(llista[i])+23);
        }
        return llista2
}
llista2 = f3(llista1)
```

Mostra per pantalla:

> [25, 27, 29]

Question 4 :

Afegeix una nova funció **printaki** a l'objecte **console** que imprimeixi .ªqui"per consola.

Solution: El codi empleat es:

```
printaki = function(){console.log("aqui")}

console.printaki = printaki
console.printaki()
```

Mostra per pantalla:
> aqui

Question 5 :

Primer fes una funció **f4** que sumi dos números (i.e., **f4 (a,b) -> a + b**), i fes una llista

$$llistaA = [1, 2, 3, 4]$$

Fes servir llistaB = llistaA.map(...) per sumar 23 a cada element de la **llistaA**, i fes servir **f4** per fer la suma. Indicació: et caldrà fer una funció addicional, ja que no es pot fer servir **f4** directament.

Solution: El codi empleat es:

```
llistaA = [1,2,3,4]
f4 = function(a,b){return a+b}
llistaB = llistaA.map(x=>f4(23,x))
```

Mostra per pantalla:

> [24, 25, 26, 27,]

Question 6 :

Crea una funció **f5** que agafi un objecte i dues funcions per paràmetres, anomenats respectivament **a**, **b**, **c**:

Solution: El codi empleat es:

```
b = f2;
c = function(r){console.log(r)};
f5 = function (a,b,c){
        c(b(a));
}
f5(1, b, c);
```

Mostra per pantalla:

> 2

Question 7 :

Afegeix una nova funció **printaki2** a l'objecte **console** que imprimeixi per consola .ªqui 1", .ªqui 2", .ªqui 3", etc. És a dir .ªqui número anterior + 1". No facis servir cap variable global (ni cap objecte global) per guardar el contador, fes servir una clausura.

Exemple d'ús:

$$console.printaki2()$$

Resultat:

$$aqui\ 1$$

Exemple d'ús:

$$console.printaki2()$$

Resultat:

$$aqui\ 2$$

Solution: El codi empleat es:

```
printaki2 = function(){ var i = 0; return function() { i++; console.
   log("aqui " + i);};
console.printaki2 = printaki2()

console.printaki2()
console.printaki2()
```

Mostra per pantalla:

>aqui 1
>aqui 2

Question 8 :

Crea una funció **f6** que tingui dos paràmetres: una llista de noms d'arxiu i una funció callback.

$$f6 = function(llista, callback\_final)$$

La funcio **f6** ha de llegir els arixus anoments a **llista** i ha de crear una nova llista **resultat** amb el contingut d'aquests arxius. I.e., cada element de resultat ha de ser el contigut d'un dels arxius.

Fes servir **.forEach(...)** i **fs.ReadFile(filename, callback)**.

Quan la funció hagi acabat de llegir *tots* els arxius, s'ha de cridar la funció callback que **f6** rep com a paràmetre (i.e., **callback_final** amb la llista resultat. Fixa't en que quan s'ha cridat l'última funció callback passada a **fs.readFile(...)** es quan realment s'han acabat de llegir tots els arxius. Fixa't també en que l'última callback del **fs.readFile** que s'executa no té perquè ser la que se li ha donat al **fs.readFile** a l'ultima iteració del **.forEach**.

Nota: el resultat no té perquè seguir l'ordre correcte (depèn de quan vagin acabant els **fs.readFile**)

Exemple d'ús:

$$fs([\text{'}a1.txt\text{'},\text{'}a2.txt\text{'}], function(result)(console.log(result)))$$

Resultat:

$$[\text{'}continguta2.txt\text{'},\text{'}continguta1.txt\text{'}]$$

Solution: El codi empleat es:

```
fs = require('fs')

callback_final = function(resultat){
    console.log(resultat)
}

f6 = function(llista, callback_final){
    var resultat = [];

    var callback = function(err, data){
        if (err){throw err}
            else{resultat.push(data)}
            if (resultat.length == llista.length)
                callback_final(resultat)

```

```
    }

    llista.forEach(filename => fs.readFile(filename, 'utf-8',
        callback))
}

llista = ['a1.txt', 'a2.txt'];
f6(llista, callback_final);
```

Mostra per pantalla:

>['contingut a1.txt', 'contingut a2.txt']

Question 9 :

Modifica la funció **f6** de l'exercici anterior perquè l'ordre de la llista **resultat** coincideix amb l'ordre original de **llista**. És a dir que a cada posició **resultat[i]** hi ha d'haver el contingut de l'arxiu anomenat a **llista[i]**. Anomena aquesta funció modificada com **f7**.

Fes servir **llista.forEach (function (element, index)()**).

Exemple d'ús:

$$fs([\prime a1.txt\prime,\prime a2.txt\prime], function(result)(console.log(result)))$$

Resultat:

$$[\prime continguta1.txt\prime,\prime continguta2.txt\prime]$$

Solution: El codi empleat es:

```
fs = require('fs')

f6 = function(llista, callback_final){
    var resultat = []
    llista.forEach(element => {
        resultat.push(0)
    });

    var f7 = function(element, index){
       var callback = function(err, data){
            resultat[index] = data;
            if (!resultat.includes(0)){
                callback_final(resultat)}
```

```
14        }
15        fs.readFile(element, 'utf-8', callback)
16    }
17
18    llista.forEach(f7)
19}
20
21llista = ['a1.txt', 'a2.txt'];
22f6(llista, function (resultat) { console.log(resultat) });
```

Mostra per pantalla:

>['contingut a1.txt', 'contingut a2.txt']

Question 10 :

Explica perquè, a l'exercici anterior, hi podria haver problemes si en comptes de fer això

$$llista.forEach(function(element, index)(/*...*/))$$

féssim això altre

$$varindex = 0$$
$$llista.forEach(function(element)(/*...*/index+ = 1))$$

Solution: Perque al ser una funció asincrona, podriem estar avançant l'index abans d'haver llegit l'arxiu i al final del bucle anar afegint totes al final, ja que index=llista.length

Per aquest motiu, s'utiliza millor el foreach, amb index i element, per asignar per parelles a cada element el seu index.

Question 11 :

Implementa la funció **asyncMap**. Aquesta funció té la següent convenció d'ús:

function **asyncMap** (list, f, callback2) {...}
- function **callback2** (err, resultList){...}
- function **f** (a, callback1) {...}
- function **callback1** (err, result) {...}

Fixa't en que **f** {...} té la mateixa forma que **fs.readFile(...)**.

La funció **asyncMap** aplica f a cada element de **list** i crida **callback2** s'ha de cridar, o bé amb el primer **err** $!= null$ que se li hagi passat a **callback1** o bé amb **resultList** contenint el resultat de la feina feta per **asyncMap** (en l'ordre correcte).

Indicació: fixa't en els paral·lelismes entre aquest exercici i els anteriors. Intenta entre que vol dir fer un map asíncron.

Solution:

```
fs = require('fs')

function asyncMap (llista, f, callback2)
{
    var resultat = [];
    var callback = function(err , data){
        if (err){throw err}
        else{resultat.push(data)}
        if (resultat.length == llista.length)
            callback2(resultat)
    }

    llista.forEach(filename => f(filename , 'utf-8', callback))
}
asyncMap(['a1.txt', 'a2.txt'], fs.readFile, function(resultat){
   console.log(resultat)});
```

['contingut a2.txt', 'contingut a1.txt']

Mostra per pantalla:

>['contingut a1.txt', 'contingut a2.txt']

Question 12 :

Fes un objecte **o1** amb tres propietats: un comptador **count**, una funció **inc** qye incrementi el comptador, i una variable **notify** que contindrà **null** o bé una funció d'un paràmetre. Feu que el comptador "notifiqui"la funció guardad a la propietat **notify** cada cop que el comptador s'incrementi.

Solution:

```
var o1 =  {
    count : 0,
    notify : null,
    inc : function(){
        this.count++;
        if(this.notify){
            this.notify(this.count)
```

```
        }
    }

}

o1.notify = null;
o1.inc()

o1.count = 1
o1.notify = function(a) {console.log(a)}
o1.inc()
```

Mostra per pantalla:

>2

Question 13 :

Fes el mateix que a l'exercici anterior però fes servir el *modulepattern* per amagar el valor del comptador i la funció especificada a **notify**. Fes un setter per la funció triada per l'usuari. Anomena l'objecta com **o2**.

El següent codi és un exemple de module pattern que pots reaprofitar:

Solution:

```
var o2 = (function() {
    var count = 1;
    var notify = null;
    return {
    inc : function(){
        count++;
        if(notify){
            notify(count)
        }
    },
    setNotify: function(b){notify=b}
    };
})();

o2.setNotify(function (a) { console.log(a) });
o2.inc()
```

Mostra per pantalla:

>2

Question 14 :

Converteix l'exemple anterior en una clase i assigna'l a un objecte **o3**. En que es diferencien els dos exemples?

Solution:

```
function Counter() {
    this.count = 0;
    this.notify = null
    this.inc = function(){
        this.count++;
        if(this.notify){
            this.notify(this.count)
        }
    }

}

o3 = new Counter();
o3.notify = null;
o3.inc()

o3.count = 1
o3.notify = function(a) {console.log(a)}
o3.inc()
```

Mostra per pantalla:

>2

Responent a la pregunta:

- La diferencia entre aquest codi i la del exercici 13, es que d'aquesta manera ens evitem repetir codi. En l'exercici 13 hauriem de posar per cadascun dels objectes la mateixa funció. En canvi en aquest exercici només hem de inicialitzar amb *new Conter*()

Question 15 :

Fes una nova classe, **DecreasingCounter**, que estengui l'anterior per herència i que faci que el mètode **inc** en realitat decrementi el comptador.

Solution:

```
function Counter() {
    this.count = 1;
```

```
    this.notify = null
    this.inc = function(){
        this.count++;
        if(this.notify){
            this.notify(this.count)
        }
    }

}

function DecreasingCounter(){
    this.inc = function(){
        this.count--;
        if(this.notify){
            this.notify(this.count)
        }
    }
}
DecreasingCounter.prototype = new Counter();
o4 = new DecreasingCounter;
o4.notify = function(a) {console.log(a)}
o4.inc()
```

Mostra per pantalla:

>0

Question 16 :

Definim un objecte de "tipus futureçom un objecte de dos camps tal i com es mostra a continuació:

- future = $\{isDone : false, result : null\}$

Aquest objecte representa el resultat d'una operació que pot haver accabar o estar-se executant encara. El camp **isDone** ens indica si l'operació ja ha acabat; inicialment és **false**, i passa a ser **true** quan l'operació ha acabar. El camp **result** és **null** mentre **isDone** == **false** i conté el resultat de l'operació quan aquesta ja ha acabat.

Es demana que implementis la funció **readIntoFuture(filename)**. Aquesta funció ha de llegir l'arxiu **filename** fent servir **fs.readFile**, però ha de retornar un objecte de tipus future (amb un **return**) encara que l'operació de lectura no hagi acabat. L'objecte retornat s'actualitzarà quan l'arxiu s'hagi llegit.

Solution:

```
fs = require("fs")

function readIntoFuture(filename){
    var future = { isDone: false, result:null }

    var callback = function(err , data){
        if (err){throw err}
        else{future.isDone=true; future.result = data}
    }
    fs.readFile(filename, 'utf-8', callback);

    return future;
}

future = readIntoFuture('a1.txt');


setTimeout(function()
{console.log(future)}, 1000)
```

Mostra per pantalla:

$>\{isDone : true, result :' continguta1.tx'\}$

Question 17 :

Suposem que tenim una funció **f** amb la mateixa convenció de crida que **fs.readFile** (això vol dir que **f** podria ser **f** ha de ser equivalent a **read.File**).

Generalitza l'exercici anterior de la següent manera. Fes una funció **asyncToFuture(f)** que çonverteixi"la funció **f** en una nova funció equivalent però que retorni un future tal que l'exercici anterior.

Noteu que **asyncToFuture(fs.readFile)** ha e ser equivalent a **readIntoFuture**.

Solution:

```
fs = require("fs")

function asyncToFuture(f){

    return f2 = function(filename){
```

```
6         var future = { isDone: false, result:null }
7
8         var callback = function(err , data){
9             if (err){throw err}
10            else{future.isDone=true; future.result = data}
11        }
12        f(filename, callback);
13
14        return future;
15    }
16
17}
18
19readIntoFuture2 = asyncToFuture(fs.readFile)
20future2a = readIntoFuture2('a1.txt')
21setTimeout(function(){console.log(future2a)}, 1000)
22
23
24statIntoFuture = asyncToFuture(fs.stat);
25future2b = statIntoFuture('a1.txt');
26setTimeout(function() { console.log(future2b) }, 1000)
```

Mostra per pantalla:

$>\{isDone : true, result :' continguta1.tx'\}$
$>\{isDone : true, result : Stats\}$

Question 18 :

Fes la funció **asyncToEnhancedFuture** que faci el mateix que la funció anterior, peró que retorni un objecte de tipus enhanced future. Els objectes d'aquest tipus tenen els tres camps que es mostren a continuació:

Els dos primers camps funcionen igual que amb un tipus future dels anteriors. Addicionalment, els objectes de tipus enhanced future tenen un tercer camp **registerCallback**. Aquest camp és una funció que rep un callback per paràmetre (i.e., **enhancedFuture.registerCallback(callback)**) i té un funcionament similar a la funció **setNotify()** vista a exercicis anteriors.

Quan es registra una funció **f** cridant a **registerCallback(f)**, l'objecte **enhancedFuture** la fa servir per notificar quan s'ha produit un canvi a **isDone**. Si **isDone** ja es **true** quan es registra el callback amb **registerCallback(f)**, s'ha de cridar **f** directament.

Solution:

```
const { throws } = require("assert")

fs = require("fs")

function asyncToEnhancedFuture(f){

    return f2 = function(filename){

        var future = { isDone: false, result:null,  registerCallback:
            function(fp)
            {
                if(this.isDone){
                    fp(this)
                }

            }
        }

        var callback = function(err , data){
            if (err){throw err}
            else{future.isDone=true; future.result = data}
        }
        f(filename, 'utf-8', callback);

        return future;
    }

}


readIntoEnhancedFuture = asyncToEnhancedFuture(fs.readFile);
enhancedFuture = readIntoEnhancedFuture('a1.txt');
setTimeout(function() { enhancedFuture.registerCallback( function(ef)
    {console.log(ef) } )}, 1000)
```

Mostra per pantalla:

$>\{isDone : true, result :' continguta1.tx', registreCallback : [Function : registerCallback]\}$

Question 19 :

Tenim que **f1(callback)** és una funció que rep un paràmetre de callback, i **f2(error, result)** és la funció de callback que faríem servir a **f1**, aleshores volem fer la funció **when** que ha de

funcionar de la següent manera. La funció **when** separa una funció de callback de la funció que la crida. Fa servir la següent sintaxi:

- **when(f1).do(f2)**

Solution:

```
fs = require('fs')

when = function(f1){
    return{
        do : function(f2){
            f1(f2)

        }
    }
}
f1 = function(callback){fs.readFile('a1.txt', 'utf-8', callback)}
f2 = function(error, result){console.log(result)}
when(f1).do(f2)
```

Mostra per pantalla:

>contingut a1.txt

Question 20 :

Modifica la solució de l'exercici anterior perquè funcioni així:

- **when(f1).and(f2).do(f3)**

En aquest cas, f1 i f2 són funcions amb un sol callback per paràmetre, i que segueixen la mateixa convenció que **fs.readFile** (i.e., el callback té dos paràmetres, **error** i **result**). La funció **f3** rep quatre paràmetres: **error1, error2, result1 i result2**.

Solution:

```
fs = require('fs')

when = function(f1){
    let err1, err2, res1, res2;
    return{
        do : function(f2){
            f1(f2)

        },
```

```
        and : function(f2){
            return {
                do : function(f3){
                    f1(function(err, res){
                        err1 = err;
                        res1 = res;

                    });
                    f2(function(err, res){
                        err2 = err;
                        res2 = res;
                        f3(err1, err2, res1, res2)
                    });

                }
            }
        }
    }
}

f1 = function(callback) { fs.readFile('a1.txt', 'utf-8', callback) }
f2 = function(callback) { fs.readFile('a2.txt', 'utf-8', callback) }
f3 = function(err1, err2, res1, res2) { console.log(res1, res2) }
when(f1).and(f2).do(f3)
```

Mostra per pantalla:

>contingut a1.txt contingut a2.txt

Question 21 :

Fes la funció **composer** que rebi dues funcions d'un sol paràmetre.

- **composer = function(f1, f2)**

El resultat d'executar **composer** ha de ser una tercera funció **f3** que sigui la composició de **f1** i **f2**. És a dir que **f3(x)** fa el mateix que **f1(f2(x))**.

Solution:

```
composer = function(f1, f2)
{
    return function(a)
    {
        b = f2(a)
```

```
        c = f1(b)
        return c
    }
}

f1 = function(a) { return a + 1 }
f3 = composer(f1, f1)
//m=f3(3)


f4 = function(a) { return a * 3 }
f5 = composer(f3, f4)
f5(3)
```

Mostra per pantalla:

>5

>11

Question 22 :

Converteix l'exercici anterior en asíncron tal com s'explica a continuació. Fes la funció **asyncComposer** que rebi dues funcions **f1** i **f2**.

En aquest cas, **f1** i **f2** són funcions de dos paràmetres que segueixen la mateixa convenció que **fs.readFile**: el primer paràmetre és un valor qualsevol, i el segón és un **callback** (que té dos paràmetres: **error** i **result**). El resultat d'executar asyncComposer ha de ser una tercera funció **f3**, que tingui la mateixa convenció de crida que **f1** i que **f2**, i que sigui la composició de **f1** i **f2**. És a dir el **callback** de **f2** ha de cridar a **f1**, i el **callback** de **f1** ha de cridar al de **f3**.

Solution:

```
fs = require('fs')

asyncComposer = function(f1, f2)
{
    return function(a, f4)
    {
        f2(a,function(err, b){
            if(!err)
            {
                f1(b, f4)
            }
            else{
```

```
                f4(err)
            }
        })

    }
}

f1 = function(a, callback) {callback(null, a + 1) }
f3 = asyncComposer(f1, f1)
f3(3, function(error, result) { console.log(result) } )


f1 = function(a, callback) { callback(null, a + 1) }
f2 = function(a, callback) { callback("error", "") }
f3 = asyncComposer(f1, f2)
f3(3, function(error, result) { console.log(error, result) } )
```

Mostra per pantalla:

>5

>error

Question 23 :

Fes un **p.then(x =>console.log(x))** per cadascuna de les promisses **p** que es mostren a continuació. Digues què s'imprimeix per pantalla i el perquè.

1. **p = Promise.resolve(0).then(x =>x + 1).then(x =>x + 2).then(x =>x + 4);**

Solution:

```
p = Promise.resolve(0).then(x=>x+1).then(x=>x+2).then(x=>x+4);
p.then(x=>console.log(x))
```

Mostra per pantalla:

>7

Responent a la pregunta:

>El motiu es perque afegim el 0 en resolve i cada vegada que fem .then(...) estem creant una nova promesa on el resolve (en el primer punt) equival al resolve anterior i fa la funció que diu .then(...)

2. **p = Promise.reject(0).then(x =>x + 1).catch(x =>x + 2).then(x =>x + 4);**

Solution:

```
p = Promise.reject(0).then(x => x + 1).catch(x => x + 2).then(x
   => x + 4);
p.then(x=>console.log(x))
```

Mostra per pantalla:

>6

Responent a la pregunta:

>El motiu es que al haver un reject no executarà ni un sol then fins arribar a un catch(...). En aquest cas es salta el primer then.

3. **p = Promise.resolve(0).then(x =>x + 1).then(x =>x + 2).catch(x =>x + 4).then(x =>x + 8);**

Solution:

```
p = Promise.resolve(0).then(x => x + 1).then(x => x + 2).catch(x
   => x + 4).then(x => x + 8);
p.then(x=>console.log(x));
```

Mostra per pantalla:

>11

Responent a la pregunta:

>El motiu es que al no haver un reject no executarà els catchs i continuira amb la linea seqüencial del promise.

4. **p = Promise.reject(0).then(x =>x + 1).then(x =>x + 2).catch(x =>x + 4).then(x =>x + 8);**

Solution:

```
p = Promise.reject(0).then(x => x + 1).then(x => x + 2).catch(x
   => x + 4).then(x => x + 8);
p.then(x=>console.log(x));
```

Mostra per pantalla:

>12

Responent a la pregunta:

>El motiu es que al haver un reject no executarà ni un sol then fins arribar a un catch(...). En aquest cas es salta dos then i encadena un then després.

5. **p = Promise.reject(0).then(x =>x + 1, null).catch(x =>x + 2) .catch(x =>x + 4);**

Solution:

```
p = Promise.reject(0).then(x => x + 1, null).catch(x => x + 2).
   catch(x => x + 4);
p.then(x=>console.log(x));
```

Mostra per pantalla:

>2

Responent a la pregunta:

>El motiu es que al haver un reject no executarà ni un sol then fins arribar a un catch(...) i no s'executará més que un catch. Fins que no hi hagi més rejects.

Question 24 :

Fes una funció **antipromise** que rebi una promise per paràmetre i retorni una promise diferent. La promise retornada s'ha de resoldre (resolve) quan la promise original es rebutjada (reject) i viceversa.

Solution:

```
antipromise = function (p_def)
{
    return new Promise((resolve, reject)=>{
        p_def.then(reject, resolve)
    })
    //return p_def.then(function(p_def){throw p_def}, x => x)
}

antipromise(Promise.reject(0)).then(console.log);
antipromise(Promise.resolve(8)).catch(console.log);
```

Mostra per pantalla:

>0

>8

Question 25 :

Fes la funció **promiseToCallback** que converteixi la funció **f** en la funció **g**, on **f** retorna el resultat en forma de promesa, mentre que **g** retorna el resultat amb un callback.

- **var g = promiseToCallback(f);**

La funció f es una funció que pren un sol paràmetre i retorna una promesa. La funció **g** és una funció que pren dos ´ parametres, el primer és el mateix paràmetre que rep la funció la **f** i el segon es una funció **callback** que cridar a amb el resultat. La funcio de callback ha de fer servir la convenció d'ús **callback(err, res)**.

Solution:

```
promiseToCallback = function(f)
{
    var a, b;
    return function(par_f, callback){
        p2 = f(par_f);
        p2.then((res)=>callback(null,res), (req)=>callback(req, null)
           )
    }

}

var isEven = x => new Promise(
    (resolve, reject) => x % 2 ? reject(x) : resolve(x)
    );
var isEvenCallback = promiseToCallback(isEven);
isEven(2).then(() => console.log("OK"), () => console.log("KO"));
isEvenCallback(2, (err, res) => console.log(err, res));
isEven(3).then(() => console.log("OK"), () => console.log("KO"));
isEvenCallback(3, (err, res) => console.log(err, res));
```

Mostra per pantalla:

>OK

>null 2

>KO

>3 null

Question 26 :

Fes la funció **readToPromise(file)** que llegeixi l'arxiu **file** amb **fs.readFile** i retorni el resultat en forma de promise.

Solution:

```
const { resolve } = require('path');
fs = require('fs')
readToPromise = function(file){
    var a, b;
    var callback=function(err, result){
        if(err != null)
        {
            b(err)
        }
        a(result)
    }
    return new Promise((resolve, reject)=>{
        a = resolve;
        b = reject;
        fs.readFile(file, callback);
    })
}
readToPromise("notfound.txt").then(x => console.log("Contents: ", x))
.catch(x => console.log("Error: ", x));
```

Mostra per pantalla:

>Contents: <Buffer 63 6f 6e 74 69 6e 67 75 74 20 61 31 2e 74 78 74>

>Error: [Error: ENOENT: no such file or directory, open 'C:Node.txt'] [
errno: -4058,
code: 'ENOENT',
syscall: 'open',
path: '<path>notfound.txt']

Question 27 :

Generalitza l'exercici anterior de la mateixa manera com **asyncToFuture** generalitza la funció **readIntoFuture**. És a dir, donada una funció **f** de dos parametres amb la mateixa convenció d' ús que **fs.readFile**, en fer

- **var g = callbackToPromise(f);**

s'ha de retornar la funció **g** que ha de fer el mateix que **f** pero retornant el resultat amb una promise. La funció **g** ha de rebre un parametre, que serà el primer paràmetre que rep **f** i ha de retornar una promise que s'assentara (will settle) al resultat d'executarà **f**. Si **f** dona error la promesa sera rebutjada (rejected) apropiadament.

Solution:

```
const { resolve } = require('path');
fs = require('fs')
callbackToPromise = function(f){
    return function(file){
        var a, b;
        var callback=function(err, result){
            if(err != null)
            {
                b(err)
            }
            a(result)
        }
        return new Promise((resolve, reject)=>{
            a = resolve;
            b = reject;
            f(file, callback);
        })
    }
}
readToPromise2 = callbackToPromise(fs.readFile);
readToPromise2("a1.txt").then(x => console.log("Contents: ", x))
.catch(x => console.log("Error: ", x));
```

Mostra per pantalla:

>Contents: <Buffer 63 6f 6e 74 69 6e 67 75 74 20 61 31 2e 74 78 74>

Question 28 :

Fes la funció **enhancedFutureToPromise** que donat un **enhancedFuture** el converteixi en una promise. Es a dir, que quan es cridi el callback registrat amb registerCallback al **enhancedFuture**, es resolgui la promesa amb el valor result de l'**enhancedFuture**.

- **var g = callbackToPromise(f);**

s'ha de retornar la funció **g** que ha de fer el mateix que **f** pero retornant el resultat amb una promise. La funció **g** ha de rebre un parametre, que serà el primer paràmetre que rep **f** i ha de retornar una promise que s'assentara (will settle) al resultat d'executarà **f**. Si **f** dona error la promesa sera rebutjada (rejected) apropiadament.

Solution:

```
fs = require('fs')
```

```
2
3 const { throws } = require("assert")
4
5 fs = require("fs")
6
7 function asyncToEnhancedFuture(f){
8     return f2 = function(filename){
9         var future = { isDone: false, result:null,  registerCallback:
10             function(fp)
11             {
12                 if(this.isDone){
13                     fp(this)
14                 }
15
16             }
17         }
18         var callback = function(err , data){
19             if (err){throw err}
20             else{future.isDone=true; future.result = data;}
21         }
22         f(filename, 'utf-8', callback);
23
24         return future;
25     }
26
27 }
28 enhancedFutureToPromise = function(enhancedFuture){
29     var a;
30     p = new Promise((result, reject) => {
31         a = result;
32     })
33     a(enhancedFuture)
34     return p
35 }
36
37 readIntoEnhancedFuture = asyncToEnhancedFuture(fs.readFile)
38 var enhancedFuture = readIntoEnhancedFuture('a1.txt');
39 var promise = enhancedFutureToPromise(enhancedFuture);
40 console.log(promise)
41 setTimeout(function(){promise.then(console.log)}, 2000)
```

Mostra per pantalla:

>Promise [
isDone: false,

result: null,
registerCallback: [Function: registerCallback

]
]
>[
isDone: true,
result: 'contingut a1.txt',
registerCallback: [Function: registerCallback]
]

Question 29 :

Fes la funció **mergedPromise** que, donada una promesa, retorni una altra promesa. Aquesta segona promesa sempre s'ha de resoldre i mai refusar (resolve and never reject) al valor que es resolgui o es refusi la promesa original.

Solution:

```
antipromise = function (p_def)
{
    return new Promise((resolve, reject)=>{
        p_def.then(reject, resolve)
    })
}
mergedPromise = function(prom){
    return prom.then(x=>x, x => antipromise(prom))
}
mergedPromise(Promise.resolve(0)).then(console.log);
mergedPromise(Promise.reject(1)).then(console.log);
```

Mostra per pantalla:

>0

>1

Question 30 :

Donades dues funcions, **f1** i **f2**, on totes dues funcions prenen un sol parametre i retornen una promesa, fes la funció functionPromiseComposer que prengui aquestes dues funcions per parametre i que retorni la funció **f3**.

- **var f3 = functionPromiseComposer(f1, f2);**

La funció **f3** ha de retornar la composicio de les funcions **f1** i **f2**. Es a dir **f3(x)** donaria el mateix resultat que **f1(f2(x))** si cap de les tres funcions retornessin promises. Indicacio: el funcionament és el mateix que el de la funció **composer** vista anteriorment, pero en aquest cas les funcions **f1, f2, i f3** retornen promises.

Solution:

```
functionPromiseComposer = function(f11, f22){
    return function(param){

        let a,b;
        p = new Promise((resolve, reject) =>{
            a = resolve;
            b = reject;
        })
        p=f22(param).then(x=>f11(x))
        return p
    }
}
var f1 = x => new Promise((resolve, reject) => resolve(x + 1));
functionPromiseComposer(f1, f1)(3).then(console.log);
var f2 = x => new Promise((resolve, reject) => reject('always fails')
   );
functionPromiseComposer(f1, f2)(3).catch(console.log);
var f3 = x => new Promise((resolve, reject) =>
setTimeout(() => resolve(x * 2), 500));
functionPromiseComposer(f1, f3)(3).then(console.log);
```

Mostra per pantalla:

>5

>always fail

>7

Question 31 :

Fes la funció **parallelPromise** que rep dues promises per parametre i retorna una tercera promise per resultat. Aquesta tercera promise sera el resultat d'executar les dues promises per paràmetre en paral·lel. És a dir, l'exercici del **when(f1).and(f2).do(f3)** pero amb promises.

Solution:

```
parallelPromise = function(p1, p2){
```

```
    let a;
    p = new Promise((resolve, reject)=>{
        a = resolve
    })
    a([p1,p2])
    return p

}
//First part
var p1 = parallelPromise(Promise.resolve(0), Promise.resolve(1));
p1.then(console.log);
//Second part
var plast = new Promise((resolve, reject) =>
setTimeout(() => resolve(0), 200));
var pfirst = new Promise((resolve, reject) =>
setTimeout(() => resolve(1), 100));
var p2 = parallelPromise(plast, pfirst);
p2.then(console.log);
setTimeout(() => p2.then(console.log), 300)
```

Mostra per pantalla:

>[Promise[0], Promise[1]]

>[Promise[<pending>], Promise[<pending>]]

>[Promise[0], Promise[1]]

Question 32 :

Fes la funció promiseBarrier. Aquesta funcio rep per paràmetre un enter estrictament més gran que zero, i retorna una llista de funcions. E.g.,

- **var list = promiseBarrier(3);**

on list és [f1, f2, f3]. Cadascuna de les funcions de la llista resultant rebra un sol paràmetre i retornarà una promesa que es resoldrà al valor del parametre. Ès a dir que a **f1(x1).then(x2 =>...)** les variables x1 i x2 sempre prendran el mateix valor. El detall important sera que, independentment de quan és cridin cadascuna de les funcions de la llista que retorna **promiseBarrier**, aquestes nomes resoldran les seves promeses un cop totes les funcions s'hagin cridat. És a dir, seguint amb l'exemple anterior, nom és cridarà a la funció que se li passi a **f1(x1).then** un cop **f1**, **f2** i **f3** hagin estat cridades. La idea es que podem usar les funcions que retorna **promiseBarrier** per sincronitzar diferents cadenes de promeses. Indicacio: la funció que se li passa a **new Promise** s'executa just quan se li passa.

Solution:

```
var promiseBarrier = function(param){
    return llista=[
        f1 = function(param){
            p = new Promise((resolt, reject)=>{

            })
            return p.then(param)
        },
        f2 = function(param){
            p = new Promise((resolt, reject)=>{

            })
            return p.then(param)
        },
        f3 = function(param){
            p = new Promise((resolt, reject)=>{

            })
            return p.then(param)
        }
    ]
}


var [f1, f2, f3] = promiseBarrier(3);
Promise.resolve(0)
.then(x => { console.log("c1 s1"); return x; })
.then(x => { console.log("c1 s2"); return x; })
.then(x => { console.log("c1 s3"); return x; })
.then(f1)
.then(x => { console.log("c1 s4"); return x; })
Promise.resolve(0)
.then(x => { console.log("c2 s1"); return x; })
.then(f2)
.then(x => { console.log("c2 s2"); return x; })
Promise.resolve(0)
.then(f3)
.then(x => { console.log("c3 s1"); return x; })
.then(x => { console.log("c3 s2"); return x; })
.then(x => { console.log("c3 s3"); return x; })
```

Mostra per pantalla:

>c1 s1

>c2 s1

>c1 s2

>c1 s3