Sistemes i Tecnologies Web
Name: Joan Mata Pàrraga
NIU: 1533089

Problemes Node

Exercici 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**.

```
1 f1 = (a) => console.log(a)
2 f1(3)
```

Mostra per pantalla:

```
3
```

Exercici 2:

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

```
1 f2 = (a) => a ≥ 0 ? 2 * a : -1
2 f2(3)
```

Mostra per pantalla:

```
6
```

```
1 f2(-2)
```

Mostra per pantalla:

```
-1
```

Exercici 3:

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

llista2 = f3(llista)

Cada element **y** de **llista2** ha de ser el resultat d'aplicar

f2(x) + 23

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

```
1 f3 = (llista) => llista.map(x=> f2(x)+23)
2 f3([1,2,3])
```

Mostra per pantalla:

```
[ 25, 27, 29 ]
```

Exercici 4:

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

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

Mostra per pantalla:

```
aqui
```

Exercici 5:

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

$$\textbf{llistaA} = \textbf{[1,2,3,4]}$$

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

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

Mostra per pantalla:

```
[ 24, 25, 26, 27 ]
```

Exercici 6:

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

```
f5 = function(a, b, c) {
    //a és un objecte, b és una funció, i c és una funció.
}
```

La funció **f5** ha d'aplicar la funció **b** a l'objecte **a**. El resultat se li ha de passar a **c**. La funció **c** ha de ser una funció *callback* amb un paràmetre (i.e., aquesta funció s'ha de cridar quan la feina que faci **f5** s'hagi acabat, i el resultat de la feina se li ha de passar com a paràmetre).

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

Mostra per pantalla:

```
2
```

Exercici 7:

Afegeix una nova funció **printaki2** a l'objecte **console** que imprimeixi per cosola "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 comptador; fes servir una clausura.

```
1 printaki2 = function() {
2     var i = 0
3     return function() { console.log("aqui " + i); i++ };
4 }
5 console.printaki2 = printaki2()
6 console.printaki2()
```

Mostra per pantalla:

```
aqui 0
```

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

Mostra per pantalla:

```
aqui 1
```

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

Mostra per pantalla:

```
aqui 2
```

Exercici 8:

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

```
f6 = function(llista, callbalck_final) { ... }
```

La funció **f6** ha de llegir els arxius anoments a **llista** i ha de crear una nova llista **resultat** amb el contigut 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 llegit *tots* els arxius, s'ha de cridar la funció callback que **f6** rep com a paràmetre (i.e., **callback_final**) amb la llista resultant. Fixat en que quan s'han cridat l'última funció callback passada a **fs.readFile(...)** és quan realment s'han acabat de llegir totos els arxius. Fixa't també en que l'última callback de **fs.readFile** que s'executa no té perquè ser la que se li ha donat al **fs.readFile** a l'última iteració del **.forEach**.
Nota: el resultat no té perquè seguir l'ordre correcte (depèn de quan vagin acabant els **fs.readFile**).

```
fs = requiere('fs')
callback_final = (resultat) => console.log(resultat)

f6 = function(llista, callback_final) {
    var resultat = []
    var callback = function(err, data) {
        if (err) { throw data }
        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)

//has de crear els archius a la carpeta que compiles amb algun text dins.
```

Mostra per pantalla:

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

Exercici 9:

Modifica la funció **f6** de l'exercici anterior perquè l'ordre de la llista **resultat** coincideixi amb órdre original de **llista**. És a dir que a cada posició **resultat[i]** hi ha d'hacer el contigut de l'arxiu anomenat a **llista[i]**. Anomena aquesta fucnió modificada com **f7**. Fes servir **llista.forEach(function (element, index))**.

```
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)
            }
        }
        fs.readFile(element, 'utf-8', callback)
    }
    llista.forEach(f7)
}

llista = ['a1.txt', 'a2.txt']
f6(llista, function(resultat) {console.log(resultat)});
```

Mostra per pantalla:

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

Exercici 10:

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

```
llista.forEach(function (element, index) { /* ... */ } )
```

féssim això altre

```
var index = 0
llista.forEach(function (element) { /* ... */ index += 1})
```

Indicació: ...suposant que fem servir **index** al callback de **fe.readFile**.

Resposta

> Perquè al ser una funció asíncrona podríem estar modificant el valor del índex abans d'entrar a la funció o després de fer-ho. El utilitzar el forEach(), evitem aquest possible canvi de valor perquè només depen de la funció.

Exercici 11:

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

```
function asyncMap(llist, 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** quan acava d'aplicar **f** a totos els elements de la llista. La funció **callback2** s'ha de cridar, o bé amb el pirmer **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 el paral·lelismes entre aquest exercici i els anteriors. Intenta entendre que vol dir fer un map asíncron.

```
1  fs = require('fs')
2
3  asyncMap = function(llist, f, callback2) {
4      var result = []
5      var callback = function(err, data) {
6          if (err) {throw data}
7          else {result.push(data)}
8
9          if (result.length == llist.length) {
10             callback2(err, result)
11         }
12     }
13     llist.forEach(filename => f(filename, 'utf-8',  callback))
14 }
15
16 asyncMap(['a1.txt'], fs.readFile, function(a,b) { console.log(b) })
```

Mostra per pantalla:

```
[ 'contingut 1' ]
```

Exercici 12:

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

```
oi = {
    count : 0,
    inc : function() {
        oi.count++;
        if (oi.notify instanceof Function) {
            oi.notify(oi.count);
        }
    },
    notify : null,
};

oi.notify = null
oi.inc()
```

Mostra per pantalla:

```
" "
```

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

Mostra per pantalla:

```
2
```

Exercici 13:

Fes el mateix que a l'exercici anterior però fes servir el *module pattern* per amagar el valor del comptador i la funció especificada a **notify**. Fes un setter per la funció triada per l'usuari. Anomena l'objecte com **o2**.
El següent codi és un exemple de module pattern que pots reaprofitar:

```
var testModule = (function() {
    var count = 1;
    return {
        inc: function() {count++;},
        count: function() {return count;}
    };
})();
```

```
var o2 = (function() {
    var _count = 1;
    var _notify = null;
    return {
        inc : function() {
            _count++;
            if (_notify instanceof Function) {
                _notify(_count);
            }
        },
```

```
        count : function() { return _count; },
        setNotify : function(a) { _notify = a; }
    };
})();

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

Mostra per pantalla:

```
2
```

Exercici 14:

Converteix l'exemple anterior en una classe i assigna'l a un objecte **o3**. En que es diferencien els dos exemples?
El següent codi és un exemple d'una classe que pots reaprofitar:

```
function Counter() {
    this.a  = 1;
    this.inc = function() { this.a++;},
    this.count = function() {return this.a;}
}
new Counter();
```

```
function Counter() {
    this.count = 1;
    this.notify = null;
    this.inc = function() {
        this.count++;
        if (this.notify instanceof Function) {
            this.notify(this.count);
        }
    };
    this.setNotify = function(a) { this.notify = a }
};

var o3 = new Counter();
o3.setNotify(function(a) { console.log(a) })
o3.inc()
```

Mostra per pantalla:

```
2
```

Resposta:

A nivell de resultats, en res. Ambdòs cassos donen el mateix resultat. Però en nivell de codi, es que no el repetim. Aquí inicialitzem amb un "new Conter()" en lloc d'haver de posar cadascun dels objectes en la mateixa funció.

a 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()

Exercici 15:

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

```
function DecreasingCounter(){
    this.inc = function() {
        this.count--;
        if (this.notify instanceof Function) {
            this.notify(this.count);
        }
    };
    this.setNotify = function(a) { this.notify = a }
};

DecreasingCounter.prototype = new Counter();
var o4 = new DecreasingCounter;
o4.setNotify(function(a) { console.log(a) })
o4.inc()
```

Mostra per pantalla:

```
0
```

Exercici 16:

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

```
function = { isDone: false, result: null }
```

Aquest objecte representa el resultat d'una operació que pot haver acabt o estar-se executant encara. El camp **isDone** ens indica si l'operació ja ha acabt; inicialment és **false** quan l'operació a ser **true** quan l'operació ha acabat. 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 **readOntoFuture(filename)**. Aquesta fucnió 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.

```
fs = require('fs')

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

    fs.readFile(filename, 'utf-8', (err, data) => {
        if (err) {
            throw data;
        }
        else {
            variable.result = data;
            variable.isDone = true;
        }
    })

    return variable;
}

console.log('a: ')
future = readIntoFuture('a1.txt')
console.log(future)
```

Mostra per pantalla:

```
a: { isDone: true, result: null }
```

```
console.log('b: ')
future = readIntoFuture('a1.txt')
setTimeout(function() { console.log(future) }, 1000)
```

Mostra per pantalla:

```
b: { isDone: true, result: 'contingut a1.txt' }
```

Exercici 17:

Suposem que tenim una funció **f** amb la mateixa convenció de crida que **fs.readFile** (això vol dir que **f** podria ser **fs.readFile**). Generalitza l'exercici anterior de la següent manera. Fes una funció **asyncToFuture(f)** que "converteixi" la funció **f** en una nova funció equivalent però que retorni un future tal que l'exercici anterior. Noteu que **asyncToFuture(fs.readFile)** ha de ser equivalent a **readIntoFuture**.

```
fs = require('fs')

asyncToFuture = function(f) {

    readIntoFuture = function(filename) {
        let variable = {isDone : false, result : null};

        f(filename, (err, data) => {
            if (err) {
                throw data;
            }
            else {
                variable.result = data;
                variable.isDone = true;
            }
        })
        return variable;
    }

    return readIntoFuture;
}

console.log('a: ')
readIntoFuture2 = asyncToFuture(fs.readFile);
future2 = readIntoFuture2('a1.txt')
setTimeout(function() { console.log(future2) }, 1000)
```

Mostra per pantalla:

```
a: {
    isDone: true,
    result: <Buffer 63 6f 6e 74 69 6e 67 75 74 20 61 31 2e 74 78 74>
}
```

```
console.log('b: ')
statIntoFuture = asyncToFuture(fs.stat);
future2 = statIntoFuture('a1.txt')
setTimeout(function() { console.log(future2) }, 1000)
```

Mostra per pantalla:

```
b:
{
  isDone: true,
  result: Stats {
    dev: 16777223,
    mode: 33188,
    nlink: 1,
    uid: 501,
    gid: 20,
    rdev: 0,
    blksize: 4096,
    ino: 25168369,
    size: 16,
    blocks: 8,
    atimeMs: 1647603650564.416,
    mtimeMs: 1645806365374.132,
    ctimeMs: 1646672167242.6658,
    birthtimeMs: 1645806123456.7698,
    atime: 2022-03-18T11:40:50.564Z,
    mtime: 2022-02-25T16:26:05.374Z,
    ctime: 2022-03-07T16:56:07.243Z,
    birthtime: 2022-02-25T16:22:03.457Z
  }
}
```

Exercici 18:

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

```
enhancedFuture = { isDone: false, result: null, registerCallback: [Function] }
```

Els dos primers camps funcionen igual que amb un tipus future dels anteriors. Adicionalment, els onbjectes de tipus enhanced future tenne 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. La funció **f** rep un paràmetre per poder accedir als camps d'**enhancedFuture**. De fet:

```
function f(enhancedFuture) { ... }
```

té un paràmetre on rep l'enhancedFuture que la crida (una especie de **this**, però no un **this** perquè no us funcionarà).

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

```
fs = require('fs')

asyncToEnhancedFuture = function(f) {

    f2 = function(filename) {
        var variable = {isDone : false, result : null, registerCallback : null};

        f(filename, (err, data) => {
            if (err) {
                throw data;
            }
            else {
                variable.result = data;
                variable.isDone = true;
            }
        })
        return {
            registerCallback : function(f3) {
                variable.registerCallback = f3
                f3(variable)
            }
        }
    }
    return f2
}
```

Mostra per pantalla:

```
{ isDone: false, result: null, registerCallback: [Function] }
```

Exercici 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** epara una funció de callback de la funció que la crida. Fa server la següent sintaxi:

```
when(f1).do(f2)
```

És a dir que ha de fer el mateix que:

```
f1(f2)
```

Fixa't en que **when** retorna un objecte amb un sol camp de nom **do**.

```
fs = require('fs')

function when(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:

```
a1.txt
```

Exercici 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**.

```
fs = require('fs')

function when(f1) {
    let err1, err2, res1, res2;
    return {
        do : function(f2){
            f1(f2)
        },
        and : function(f2){
            return {
                do : function(f3){
                    f1(function(err1, res1){
                        f2(function(err2, res2){
                            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
```

Exercici 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 funciío **f3** que sigui la composició de **f1** i **f2**. És a dir que **f3(x)** fa el mateix que **f1(f2(x))**.

```
composer = function(f1, f2) {
    f = function(x){
        return f1(f2(x))
    }
```

```
    return f
}

f1 = function(a) { return a + 1 }
f3 = composer(f1, f1)
console.log(f3(3))
```

Mostra per pantalla:

```
5
```

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

Mostra per pantalla:

```
11
```

Exercici 22:

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

```
asyncComposer = fucnion(f1, f2) { ... }
```

En aquesta cas, **f1** i **f2** són funcions de dos paràmetres qeu segueixen la mateixa convenció que **fs.readFile**: el primer paràmetre és un valor qualsevol, i el segons és u 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 a **f3**.

```
asyncComposer = function(f1, f2) {
    f3 = function(data, callback){
        return f2(data, function(err, f) {
            if(!err) {
                f1(f, callback)
            } else {
                callback(err)
            }
        })
    }
    return f3
}

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

Mostra per pantalla:

```
5
```

Si **error** és diferent de null al resultat de **f2**, aleshores el callback de **f3** ha de tornar directament aquest error.

```
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:

'error'

Exercici 23:

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

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

Al estar "resolve" fa tots els then(), llavors: 1+2+4=7

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

Com esta en "reject" fa primer el catch() i després el then() consecutiu, llavors: 2+4=6

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

Al estar "resolve" fa tots els then() (es salta el catch(), llavors: 1+2+8=11

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

Com esta en "reject" fa primer el catch() i després el then() consecutiu, llavors: 4+8=12

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

Al ser una promesa "reject" només fa el primer catch(), llavors 2.

Exercici 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 rebitjada (reject) i viceversa.

```
antipromise = function(p) {
    return new Promise((resolve, reject)=>{
        p.then(reject, resolve)
    })
}
antipromise(Promise.reject(0)).then(console.log);
```

Mostra per pantalla:

```
0
```

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

Mostra per pantalla:

```
0
```

Indicació: si us apareix un **UnhandlePromiseRejectionWarning**, vol dir que la promesa s'ha resolt abans que el **.catch** s'hagi executat. És a dir que:

```
var p = Promise.reject(0);
setTimeout(() => p.catch(console.log));
```

no és el mateix que:

```
Promise.reject(0).catch(console.log);
```

Exercici 25:

Fes la funció **promiseToCallbcak** 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** és una funció que pren un sol paràmetre i retorna una promesa. La funció **g** és una funció que pren dos paràmetres, el primer és el mateix paràmetre que rep la funció **f** i el segon és una funció callback que cridarà amb el resultat. La funció de callback ha de fer servir la conversió d'ús **callback(err, res)**.

```
promiseToCallback = function(f) {
    return function(x, callback) {
        f(x).then(a => {callback(null,a)}, b => {callback(b,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));
```

Mostra per pantalla:

```
OK
null 2
```

```
isEven(3).then(() => console.log("OK"), () => console.log("KO"));
isEvenCallback(3, (err, res) => console.log(err, res));
```

Mostra per pantalla:

```
KO
3 null
```

Exercici 26:

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

```
readToPromise = function(file) {
    const fsPromises = require('fs').promises;
    return fsPromises.readFile(file);
}

readToPromise("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\
```

```
readToPromise("notfound.txt").then(x => console.log("Contents: ", x)).catch(x => ...
    console.log("Error: ", x));
```

Mostra per pantalla:

```
Error:  [Error: ENOENT: no such file or directory, open 'notfound.txt'] {
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'notfound.txt'
}
```

Exercici 27:

Generalitza l'exercici anterior de la mateixa manera com **asyncToFuture** generalitza la funció **readIntoFuture**. És a dir, donada una funció f de dos paràmetres 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** però retornant el resultat amb una promise. La funció **g** ha de rebre un paràmetre, que serà el primer paràmetre que rep **f** i ha de retornar una promise que s'assentarà (will settle) al resultat d'executar **f**. Si **f** dona error la promesa serà rebutjada (rejected) apropiadament.

```
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¿
```

Exercici 28:

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

```
fs = require('fs')

asyncToEnhancedFuture = function(f) {

    f2 = function(filename) {
        var variable = {isDone : false, result : null, registerCallback : null};

        f(filename, (err, data) => {
            if (err) {
                throw data;
            }
            else {
                variable.result = data;
                variable.isDone = true;
            }
        })
        return {
            registerCallback : function(f3) {
                variable.registerCallback = f3
                f3(variable)
            }
        }
    }
    return f2
}

enhancedFutureToPromise = function(f4) {
    var x;
    p = new Promise((result, reject) => { x = result; })
    x(f4)
    return p
}

readIntoEnhancedFuture = asyncToEnhancedFuture(fs.readFile);
var enhancedFuture = readIntoEnhancedFuture('a1.txt');
var promise = enhancedFutureToPromise(enhancedFuture);
promise.then(console.log)
```

Mostra per pantalla:

```
'contingut 1'
```

Exercici 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.

```
antipromise = function(p) {
    return new Promise((resolve, reject)=>{
        p.then(reject, resolve)
    })
}

mergedPromise = function(p){
    return p.then(x=>x, x => antipromise(p))
}

mergedPromise(Promise.resolve(0)).then(console.log);
```

Mostra per pantalla:

```
0
```

```
mergedPromise(Promise.reject(1)).then(console.log);
```

Mostra per pantalla:

```
1
```

Exercici 30:

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

```
var f3 = functionPromiseComposer(f1, f2);
```

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

```
functionPromiseComposer = function(fa, fb){
    return function(param){
        let a,b;
        p = new Promise((resolve, reject) =>{
            a = resolve;
            b = reject;
        })
        p = fb(param).then(x=>fa(x))
        return p
    }
}
var f1 = x => new Promise((resolve, reject) => resolve(x + 1));
functionPromiseComposer(f1, f1)(3).then(console.log);
```

Mostra per pantalla:

```
5
```

```
var f2 = x => new Promise((resolve, reject) => reject('always fails'));
functionPromiseComposer(f1, f2)(3).catch(console.log);
```

Mostra per pantalla:

```
always fails
```

```
var f3 = x => new Promise((resolve, reject) => setTimeout(() => resolve(x * 2), 500));
functionPromiseComposer(f1, f3)(3).then(console.log);
```

Mostra per pantalla:

```
7
```

Exercici 31:

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

```
parallelPromise = function(p1, p2) {
    let a;
    p = new Promise((resolve, reject)=> {a = resolve })
    a([p1,p2])
    return p
}
var p1 = parallelPromise(Promise.resolve(0), Promise.resolve(1));
p1.then(console.log);
```

Mostra per pantalla:

```
[ Promise { 0 }, Promise { 1 } ]
```

```
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 { <pending> }, Promise { <pending> } ]
[ Promise { 0 }, Promise { 1 } ]
```

Exercici 32:

Fes la funció **promiseBarrier**. Aquesta funció 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 rebrà un sol paràmetre i retornarà una promesa que es resoldrà al valor del paràmetre. És a dir que a

```
f1(x1).then(x2 => ...)
```

les variables **x1** i **x2** sempre prendran el mateix valor. El detall important serà que, independentment de quan és cridin cadascuna de les funcions de la llista que retorna **promiseBarrier**, aquestes només resoldran les seves promeses un cop totes les funcions s'hagin cridat. És a dir, seguint amb l'exemple anterior, només es cridarà la funció que se li passi a **f1(x1).then** un cop **f1**, **f2** i **f3** hagin estat cridades. La idea és que podem usar les funcions que retorna **promiseBarrier** per sincronitzar diferents cadenes de promeses. Indicació: la funció que se li passa a **new Promise** s'executa just quan se li passa.

```
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)
        }
    ]
}
```

Pregunta 1:

```
var [f1, f2] = promiseBarrier(2);
Promise.resolve(0)
.then(f1)
.then(x => { console.log("c1 s1"); return x; })
.then(x => { console.log("c1 s2"); return x; })

Promise.resolve(0)
.then(x => { console.log("c2 s1"); return x; })
.then(x => { console.log("c2 s2"); return x; })
.then(x => { console.log("c2 s3"); return x; })
.then(x => { console.log("c2 s4"); return x; })
.then(f2)
```

Mostra per pantalla:

```
c2 s1 c2 s2 c2 s3 c2 s4
```

Pregunta 2:

```
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 c1 s2 c1 s3 c2 s1
```

Pregunta 3:

```
var [f1, f2] = promiseBarrier(2);
Promise.resolve(1).then(f1).then(console.log)
Promise.resolve(2).then(f2).then(console.log)
```

Mostra per pantalla:

```
" "
```