Problemes NodeJS

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**.

   Exemple d'us:
   ```
   f1(3)
   ```
   Resultat:
   ```
   3
   ```

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

   Exemple d'us:
   ```
   f2(3)
   ```
   Resultat:
   ```
   6
   ```

   Exemple d'us:
   ```
   f2(-2)
   ```
   Resultat:
   ```
   -1
   ```

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.

   Exemple d'us:
   ```
   f3([1,2,3])
   ```
   Resultat:
   ```
   [25,27,29]
   ```

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

   Exemple d'us:
   ```
   console.printaki()
   ```
   Resultat:
   ```
   aqui
   ```

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

   ```
   llistaA = [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: et caldrà fer una funció addicional, ja que no es pot fer servir **f4** directament.

   Exemple d'us:
   ```
   llistaB
   ```
   Resultat:
   ```
   [24,25,26,27]
   ```

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).

   Exemple d'us:
   ```
   f5(1, f2, function(r) { console.log(r) })
   ```
   Resultat:
   ```
   2
   ```

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 comptador; fes servir una clausura.

   Exemple d'us:
   ```
   console.printaki2()
   ```
   Resultat:
   ```
   aqui 1
   ```

   Exemple d'us:
   ```
   console.printaki2()
   ```
   Resultat:
   ```
   aqui 2
   ```

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 funció **f6** ha de llegir els arxius anomenats 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 contingut 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 resultant. Fixa't en que quan s'ha cridat l'última funció callback passada a **fs.readFile(...)** és 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'última iteració del **.forEach**.

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

   Exemple d'us:
   ```
   f6(['a1.txt','a2.txt'], function (result) { console.log(result) })
   ```
   Resultat:
   ```
   ['contingut a2.txt', 'contingut a1.txt']
   ```

9. Modifica la funció **f6** de l'exercici anterior perquè l'ordre de la llista **resultat** coincideixi 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'us:
   ```
   f7(['a1.txt','a2.txt'], function (result) { console.log(result) })
   ```
   Resultat:
   ```
   ['contingut a1.txt', 'contingut a2.txt']
   ```

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

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

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

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

    ```
    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` quan acaba d'aplicar `f` a tots els elements de la llista. La funció `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 entendre que vol dir fer un map asíncron.

Exemple d'us:
```
asyncMap(['a1.txt'], fs.readFile, function (a, b) { console.log(b) })
```
Resultat:
```
['contingut 1']
```

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.

Exemple d'us:
```
o1.notify = null; o1.inc()
```
Resultat:

Exemple d'us:
```
o1.count = 1; o1.notify = function(a) { console.log(a) }; o1.inc()
```
Resultat:
```
2
```

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; }
        };
})();
```

Exemple d'us:
```
o2.setNotify(function (a) { console.log(a) }); o2.inc()
```
Resultat:
```
2
```

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();
```

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

16. Definim un objecte de "tipus future" com 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 acabat 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 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 **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.

Exemple d'us:
```
future = readIntoFuture('a1.txt'); console.log(future)
```
Resultat:
```
{ isDone: false, result: null }
```

Exemple d'us:
```
future = readIntoFuture('a1.txt');
setTimeout(function() { console.log(future) }, 1000)
```
Resultat:
```
{ isDone: true, result: 'contingut 1' }
```

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**.

    Exemple d'us:
    ```
    readIntoFuture2 = asyncToFuture(fs.readFile);
    future = readIntoFuture2('a1.txt');
    setTimeout(function() { console.log(future) }, 1000)
    ```
    Resultat:
    ```
    { isDone: true, result: <Buffer 63 6f 6e 74 69 6e 67 75 74 20 31 0a> }
    ```

    Exemple d'us:
    ```
    statIntoFuture = asyncToFuture(fs.stat);
    future = statIntoFuture('a1.txt');
    setTimeout(function() { console.log(future) }, 1000)
    ```
    Resultat:
    ```
    { isDone: true, res : Stats { dev: 64769, mode: 33188, /*...*/ } }
    ```

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 tres 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. 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.

    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) } )
    ```
    Resultat:

```
{ isDone: true, result: 'contingut 1', registerCallback: [Function] }
```

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)
    ```

    És a dir que ha de fer el mateix que:

    ```
    f1(f2)
    ```

    Exemple d'us:

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

    Resultat:

    ```
    'contingut 1'
    ```

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

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`.

    Exemple d'us:

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

    Resultat:

    ```
    'contingut 1 contingut 2'
    ```

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))`.

    Exemple d'us:

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

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

    Resultat:

    ```
    5
    11
    ```

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

    ```
    asyncComposer = function(f1, 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**.

Exemple d'us:
```
f1 = function(a, callback) { callback(null, a + 1) }
f3 = asyncComposer(f1, f1)
f3(3, function(error, result) { console.log(result) } )
```
Resultat:
```
5
```

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

Exemple d'us:
```
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) } )
```
Resultat:
```
'error'
```

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.

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

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

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

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

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

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.

    Exemple d'us:
    ```
    antipromise(Promise.reject(0)).then(console.log);
    ```
    Resultat:
    ```
    0
    ```

    Exemple d'us:
    ```
    antipromise(Promise.resolve(0)).catch(console.log);
    ```
    Resultat:
    ```
    0
    ```

    Indicació: si us apareix un **UnhandledPromiseRejectionWarning**, 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);
    ```.

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** 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 la **f** i el segon és una funció callback que cridarà amb el resultat. La funció de callback ha de fer servir la convenció d'ús **callback(err, res)**.

    Exemple d'us:

```
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));
```

Resultat:

```
OK
null 2
KO
3 null
```

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

Exemple d'us:

```
readToPromise("a1.txt").then(x => console.log("Contents: ", x))
  .catch(x => console.log("Error: ", x));
```

Resultat:

```
Contents:  <Buffer 74 65 73 74 0a>
```

Exemple d'us:

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

Resultat:

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

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.

Exemple d'us:

```
readToPromise2 = callbackToPromise(fs.readFile);
readToPromise2("a1.txt").then(x => console.log("Contents: ", x))
  .catch(x => console.log("Error: ", x));
```

Resultat:

```
Contents:  <Buffer 74 65 73 74 0a>
```

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**.

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.

Exemple d'us:

```
mergedPromise(Promise.resolve(0)).then(console.log);
mergedPromise(Promise.reject(1)).then(console.log);
```

Resultat:

```
0
1
```

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.

    Exemple d'us:

    ```
    var f1 = x => new Promise((resolve, reject) => resolve(x + 1));
    functionPromiseComposer(f1, f1)(3).then(console.log);

    var f3 = x => new Promise((resolve, reject) => reject('always fails'));
    functionPromiseComposer(f1, f3)(3).catch(console.log);
    ```

    Resultat:

    ```
    5
    always fails
    ```

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.

    Exemple d'us:

    ```
    var p2 = parallelPromise(Promise.resolve(0), Promise.resolve(1));
    p2.then(console.log);
    ```

    Resultat:

    ```
    [0, 1]
    ```

    Indicació: no cal gestionar els errors.

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.

    La idea és que podem usar les funcions que retorna **promiseBarrier** per sincronitzar diferents cadenes de promeses.

    Exemple d'us:

    ```
    var [f1, f2] = promiseBarrier(2);

    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(x => { console.log("c1 s4"); return x; })
      .then(f1)

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

    Resultat:

```
c1 s1
c1 s2
c1 s3
c1 s4
c2 s1
c2 s2
```

Exemple d'us:

```
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; })
```

Resultat:

```
c1 s1
c2 s1
c1 s2
c1 s3
c2 s2
c3 s1
c3 s2
c1 s4
c3 s3
```

Indicació: la funció que se li passa a `new Promise` s'executa just quan se li passa.