NodeJS Problemes

1. Create a function **f1** that takes one parameter **a** and writes its value to the console by using the function **log** in the **console** object.

   Usage example:
   ```
   f1(3)
   ```
   Result:
   ```
   3
   ```

2. Create a function **f2** that takes one parameter **a** and returns `2 * a` when `a >= 0` and returns `-1` otherwise.

   Usage example:
   ```
   f2(3)
   ```
   Result:
   ```
   6
   ```
   Usage example:
   ```
   f2(-2)
   ```
   Result:
   ```
   -1
   ```

3. Create a function **f3** that receives a list as first parameter and that returns a list.

   ```
   list2 = f3(list)
   ```

   Each element **y** in **list2** must be the result of applying

   ```
   f2(x) + 23
   ```

   to each element **x** in **list**, where **f2** is the function from the previous exercise.

   Usage example:
   ```
   f3([1,2,3])
   ```
   Result:
   ```
   [25,27,29]
   ```

4. Add a new function named **printaki** to the **console** object. The function must print “aqui” on the console when called.

   Usage example:
   ```
   console.printaki()
   ```
   Result:
   ```
   aqui
   ```

5. First, create a function **f4** that adds two numbers (i.e., `f4(a,b)` $\mapsto$ `a + b`), and create the following list.

   ```
   listA = [1,2,3,4]
   ```

   Then, use `listB = listA.map(...)` to add `23` to each element in **listA**, Employ **f4** to calculate the addition. Note: an auxiliary function is needed, given that **f4** cannot be used directly.

   Usage example:
   ```
   listB
   ```
   Result:
   ```
   [24,25,26,27]
   ```

6. Create a function **f5** that takes one object and two functions as parameters, named **a**, **b**, and **c**, respectively.

   ```
   f5 = function(a, b, c) {
           // a is an object, b is a function, and c is a function.
   }
   ```

   Function **f5** must apply function **b** to object **a**. The result of **f5** must be passed to **c**. Function **c** must be a *callback* function of a single parameter (i.e., this function must be called once **f5** returns, and the returned value must be its parameter).

   Usage example:
   ```
   f5(1, f2, function(r) { console.log(r) })
   ```
   Result:
   ```
   2
   ```

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

7. Add a new **printaki2** function to the **console** object that prints on the console: “aqui 1”, “aqui 2”, “aqui 3”, etc. That is it prints “aqui {previous number + 1}” each time it is called. Do not use a global variable (nor a variable of a global object) to store the counter; use a closure.

   Usage example:
   ```
   console.printaki2()
   ```
   Result:
   ```
   aqui 1
   ```
   Usage example:
   ```
   console.printaki2()
   ```
   Result:
   ```
   aqui 2
   ```

8. Create a function **f6** with two parameters: a list of file names, and a callback function.

   ```
   f6 = function(list, final_callback) { ... }
   ```

   Function **f6** must read the contents of the files in **list** and must create a new list, called **result**, with the content of these files. I.e., each element in **result** must be the contents of one of the files.

   Use **.forEach(...)** and **fs.readFile(filename, callback)**.

   Once *all* files have been *read*, the callback function **final_callback**, that **f6** takes as parameter, must be called with the resulting list.

   Note all files have been completely read only when all the callback functions given to **fs.readFile(...)** have been called. Note also that the last callback called will not necessarily be the one given to **fs.readFile** on the last iteration of **.forEach**.

   It is not necessary that **result** preserves the order of **list** (it can depend on when each **fs.readFile** finalizes).

   Usage example:
   ```
   f6(['a1.txt','a2.txt'], function (result) { console.log(result) })
   ```
   Result:
   ```
   ['contents of a2.txt', 'contents of a1.txt']
   ```

9. Modify function **f6** from the previous exercise so that the order of list **result** matches that of **list**. That is that **result[i]** must correspond to the contents of the file in **list[i]**. Name the modified function as **f7**.

   Use **list.forEach(function (element, index) {})**.

   Usage example:
   ```
   f7(['a1.txt','a2.txt'], function (result) { console.log(result) })
   ```
   Result:
   ```
   ['contents of a1.txt', 'contents of a2.txt']
   ```

10. Explain why, in the previous exercise, it is an issue that we replace...

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

    by...

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

    (hint) ...given that we use **index** in the callback of **fs.readFile**.

11. Implement the function **asyncMap**. This function takes parameters as follows.

    ```
    function asyncMap(list, f, callback2) {...}

    function callback2(err, resultList) {...}
    function f(a, callback1) {...}
    function callback1(err, result) {...}
    ```

    Note that **f(...)** has the same parameters as **fs.readFile(...)**.

    Function **asyncMap** applies **f** to each element in **list** and calls **callback2** once all applications of **f** have been completed. Function **callback2** must be called, either

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

- with **err** being not `null` (being equal to the first non-null **err** received in **callback1**) and a `null` **resultList**; or
- with **err** being `null` and **resultList** having the correct result, and in the right order.

Hint: realize how similar this exercise is to the previous ones. Try to understand what is an asynchronous map.

Usage example:
```
asyncMap(['a1.txt'], fs.readFile, (a,b)=>console.log(`error: ${a} data:${b}`))
```
Result:
```
['content 1']
```

```
asyncMap(['a1.txt', 'a2.txt', 'notExists1.txt', 'notExists2.txt'], fs.readFile,
  (a,b)=>console.log(`error: ${a} data:${b}`))
```

12. Create an object **o1** with three properties: a counter **count**, a function **inc** that increments the counter, and a variable **notify** that will either be `null` or a single-parameter function. Make the counter "notify" (call) the function stored in the variable **notify** (if not `null`) when the counter is increased.

Usage example:
```
o1.notify = null; o1.inc()
```
Result:

Usage example:
```
o1.count = 1; o1.notify = function() { console.log("notified") }; o1.inc()
```
Result:
```
notified
```

12.+ Improve the previous exercise. Ensure that when making the notification, the new value of the counter is specified to the function stored in the notify property.

Usage example:
```
o1.count = 1; o1.notify = function(a) { console.log(a) }; o1.inc()
```
Result:
```
2
```

13. Repeat the previous exercise but now employ the *module pattern* to hide the value of the counter and the variable **notify**. Create a setter for the **notify** function. Name this new object as **o2**.

Here you have an example of the module pattern that you can reuse:

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

Usage example:
```
o2.setNotify(function (a) { console.log(a) }); o2.inc()
```
Result:
```
2
```

14. Repeat the previous exercise but employ a class constructor instead of a module pattern. Create an object o3. How are these cases different?

Here you have an example of a class constructor that you can reuse:

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

new Counter();
```

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

Usage example:
```
o3.setNotify(function (a) { console.log(a) }); o3.inc()
```
Result:
```
2
```

14.⁺ Modify the previous Counter class so that it only 'reveals' the methods **inc** and **setNotify**.

15. Create a new class, **DecreasingCounter**, that extends the previous one through inheritance and where the **inc** method decreases the counter (instead of increasing it).

16. Let us define an object of "future" type as an object with two fields as follows:

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

This object represents the result of an operation that could have finished or could still be executing. The field **isDone** indicates whether the operation is completed or not; its value is initially **false**, and becomes **true** when the operation completes. The field **result** is **null** while **isDone == false** and contains the result of the operation once it is complete.

For this exercise, create the function **readIntoFuture(filename)**. This function reads the contents of the file **filename** by employing **fs.readFile**, but returns an object of "future" type (using **return**) regardless of whether the read operation has been completed. The returned object, must be updated once the file has been read.

Usage example:
```
future = readIntoFuture('a1.txt'); console.log(future)
```
Result:
```
{ isDone: false, result: null }
```

Usage example:
```
future = readIntoFuture('a1.txt');
setTimeout(function() { console.log(future) }, 1000)
```
Result:
```
{ isDone: true, result: 'contents 1' }
```

17. Suppose that function **f** has the same calling convention as **fs.readFile** (this means that **f** could be **fs.readFile**).

Generalize the previous exercise as follows. Create a function **asyncToFuture(f)** that "converts" the function **f** into a new equivalent function but that returns a future, as in the previous exercise, instead of employing a callback function. **asyncToFuture** will return this function (**g**) wich will "convert"(wrap) the function **f**.

Note that **asyncToFuture(fs.readFile)** must be equivalent to **readIntoFuture**.

Usage example:
```
readIntoFuture2 = asyncToFuture(fs.readFile);
future = readIntoFuture2('a1.txt');
setTimeout(function() { console.log(future) }, 1000)
```
Result:
```
{ isDone: true, result: 'contents 1' }
```

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

18. Create the function **asyncToEnhancedFuture**, which works as the one in the previous exercise, but that returns an object of "enhanced future" type. Such "enhanced futures" are objects with three fields as follows:

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

The first two fields operate as in the plain "future" type. In addition, "enhanced futures" have a third field named **registerCallback**. This field receives a callback function as parameter (i.e.,

enhancedFuture.registerCallback(callback)) and has a similar behavior as the **setNotify** function seen previously, i.e. you might need a new future's property (notify) that you will initialize through the **registerCallback** function.

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

When a function **cb** is registered by calling **registerCallback(cb)**, the object **enhancedFuture** employs **cb** to signal that a change has been produced in **isDone**. If **isDone** is already **true** when a callback is registered through **registerCallback(cb)**, then **cb** must be called directly.

Function **cb** must receive one parameter to be able to access the fields of **enhancedFuture**. In fact:

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

receives a parameter which is the same **enhancedFuture** object that has called **cb** via the **notify** function.

Usage example:
```
const utfReadFile = (f, c) => fs.readFile(f, 'utf-8', c);
const readIntoEnhancedFuture = asyncToEnhancedFuture(utfReadFile);
enhancedFuture = readIntoEnhancedFuture('a1.txt');
enhancedFuture.registerCallback( function(ef) {console.log(ef) } )
```
Result:
```
{ isDone: true, result: 'contingut 1', registerCallback: [Function] }
```

19. Given a function **f1(callback)** that takes one callback parameter, and a function **f2(error, result)** that we would use as callback function in **f1**, create the function **when** as follows.

    Function **when** separates a callback function from the function where it is used. It employs the following syntax:

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

    Which would be the same as:

    ```
    f1(f2)
    ```

    Usage example:
    ```
    f1 = function(callback) { fs.readFile('a1.txt', callback) }
    f2 = function(error, result) { console.log(result) }
    when(f1).do(f2)
    ```
    Result:
    ```
    'contents 1'
    ```

    Note that **when** returns an object with a single field that is named do.

20. Modify the previous exercise so that **when** now works as follows:

    ```
    when(f1).and(f2).do(f3)
    ```

    In this case, **f1** i **f2** are both functions of a single callback parameter, and follow the same calling convention as **fs.readFile** (i.e., the callback has two parameters: **error** and **result**). Function **f3** has four parameters: **error1**, **error2**, **result1** i **result2**.

    Usage example:
    ```
    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)
    ```
    Result:
    ```
    'contents 1 contents 2'
    ```

21. Create the **composer** function. It receives two functions of a single parameter.

    ```
    composer = function(f1, f2) { ... }
    ```

    The result of calling **composer** must be a third function **f3** that is the composition of **f1** and **f2**. That is that **f3(x)** must be the same as **f1(f2(x))**.

    Usage example:

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

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

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

Result:

```
5
11
```

22. Convert your solution to the previous exercise into an asynchronous composer as follows. Create the function **`asyncComposer`**, which receives two functions: **`f1`** i **`f2`**.

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

    Functions **`f1`** and **`f2`** are functions following the same calling convention as **`fs.readFile`**: the first parameter is an arbitrary value, while the second one is a callback function, which has two parameters **`error`** and **`result`**.

    The result of calling **`composer`** must be a third function **`f3`**, having the same calling convention as **`f1`** and **`f2`**, and that is the composition of **`f1`** and **`f2`**. That is that the callback of **`f2`** must call **`f1`**, and the callback of **`f1`** must call to the one of **`f3`**.

    Usage example:

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

    Result:

    ```
    5
    ```

    If **`error`** is not `null` for **`f2`**, then the callback for **`f3`** must return this error directly.

    Usage example:

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

    Result:

    ```
    'error'
    ```

23. Execute **`p.then(x => console.log(x))`** for each one of the following promises **`p`**. Guess and justify the output.

    (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. Build the function **`antipromise`** that receives a promise as a parameter and returns a different promise. The returned promise must resolve when the original promise is rejected, and vice versa.

    Usage example:

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

    Result:

    ```
              0
    ```

    Usage example:

    ```
    antipromise(Promise.resolve(1)).catch(console.log);
    ```

    Result:

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

```
            1
```

In case you encounter an **UnhandledPromiseRejectionWarning**:

It means that the promise has been resolved before the `.catch` has been executed. That is,
```
let p = Promise.reject(0);
setTimeout(() => p.catch(console.log));
```
is not the same as
```
Promise.reject(0).catch(console.log);
```
Therefore, if you do

```
let p = Promise.reject(0)
p.then(x => 'this callback will not be called')
p.catch(x => 'this callback will be called')
```

the result of the **then** will be a rejected promise which you are not catching (see exercise 23).

25. Create the function **promiseToCallback** that converts the function **f** into the function **g**, where **f** returns the result as a promise, while **g** returns the result with a callback.

```
let g = promiseToCallback(f);
```

The function **f** is a function that takes a single parameter and returns a promise. The function **g** is a function that takes two parameters, the first is the same parameter received by the function **f** and the second is a callback function that will be invoked with the result. The callback function must follow the usage convention of **callback(err, res)**.

Usage example:
```
const isEven = x => new Promise(
(resolve, reject) => x % 2 ? reject(x) : resolve(x)
);
const 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));
```
Result:
```
OK
null 2
KO
3 null
```

26. Create the function **readToPromise(file)** that reads the file **file** using **fs.readFile** and returns the result as a promise.

Usage example:
```
readToPromise("a1.txt").then(x => console.log("Contents: ", x))
.catch(x => console.log("Error: ", x));
```
Result:
```
Contents:  <Buffer 74 65 73 74 0a>
```

Usage example:
```
readToPromise("notfound.txt").then(x => console.log("Contents: ", x))
.catch(x => console.log("Error: ", x));
```
Result:
```
Error:  { [Error: ENOENT: no such file or directory, open 'notfound.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'notfound.txt' }
```

27. Generalize the previous exercise in the same way as **asyncToFuture** generalizes the function **readIntoFuture**.

    That is, create a function **callbackToPromise** with the following signature:

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

```
callbackToPromise = (f) => h
```

where **f** is a function with two parameters following the same usage convention as **fs.readFile**, and **h** is a function with the following signature:

```
h(x) => Promise
```

The function **h** must do the same as **f** but return the result with a promise that will resolve to the result of executing **f**. If **f** encounters an error, the promise will reject.

Usage example:

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

Result:

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

28. Create the function **enhancedFutureToPromise** that, given an object **enhancedFuture**, converts it into a promise:

```
enhancedFutureToPromise = (enhancedFuture) => Promise
```

That is, when the **notify** function of the **enhancedFuture** is called, the promise should be resolved with the value of the property **result** of the **enhancedFuture**.

Hint 1: Note that the **notify** function is initialized with a callback. When we call the **notify** function, we are actually calling this callback. Now, in addition to calling the callback, we need to resolve a promise.

Hint 2: You may need to override the **registerCallback** method in order to change the functionality of the **notify** function. Remember that **registerCallback** receives a callback as a parameter, which receives an **enhancedFuture** object as a parameter:

```
registerCallback(<callback(<enhancedFuture>)>)
```

Note: You can import the code from exercise 18 with the code:

```
const readIntoEnhancedFuture = require('./18').readIntoEnhancedFuture;
```

Also, you need to export the function **readIntoEnhancedFuture** in the file **18.js** following the design pattern **revealing module pattern**:

```
module.exports = {readIntoEnhancedFuture}
```

Usage example:

```
const enhancedFuture = readIntoEnhancedFuture('a1.txt');
const promise = enhancedFutureToPromise(enhancedFuture);
promise.then(console.log)
enhancedFuture.registerCallback(console.log)
```

Result:

```
{
  isDone: true,
  result: 'content 1',
  notify: [Function (anonymous)],
  registerCallback: [Function (anonymous)],
  cb: [Function: log]
}
'content 1'
```

29. Create the function **mergedPromise** that, given a promise, returns another promise. This second promise should always resolve and never reject to the value that the original promise resolves or rejects to.

Usage example:

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

Result:

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

```
0
1
```

30. Given two functions, **f1** and **f2**, where both functions take a single parameter and return a promise, create the function **promiseComposer** that takes these two functions as parameters and returns the function **f3**.

```
const f3 = promiseComposer(f1, f2);
```

The function **f3** should return a promise that resolves with the result of composing the functions **f1** and **f2**. That is, **f3(x)** should yield the same result as **f1(f2(x))** if none of the three functions were returning promises. Hint: the functionality is the same as the **composer** function seen earlier, but in this case, the functions **f1**, **f2**, and **f3** return promises. Usage example:

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

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

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

Result:

```
5
always fails
7
```

Note: the order of the results in the usage example may vary depending on the implementation.

31. Create the function **parallelPromise** that takes two promises as parameters and returns a third promise as the result. This third promise will be the result of dealing with the two promises passed as parameters in parallel and should resolve to an array fulfilled with the value the parameter promises resolve/reject to. Those result values have to be placed in order (the value resolved by the promise passed as the first parameter is placed in the first position). In other words, it is the exercise of `when(f1).and(f2).do(f3)` but with promises.

Usage example:

```
let p1 = parallelPromise(Promise.resolve(0), Promise.resolve(1));
p1.then(console.log);
```

Result:

```
[0, 1]
```

Usage example:

```
let plast = new Promise((resolve, reject) =>
setTimeout(() => resolve('left'), 200));
let pfirst = new Promise((resolve, reject) =>
setTimeout(() => resolve('right'), 100));

let p2 = parallelPromise(plast, pfirst);
p2.then(console.log);
```

Result:

```
[ 'left', 'right' ]
```

Usage example:

```
let plast = new Promise((resolve, reject) =>
setTimeout(() => reject('left rejected'), 200));
let pfirst = new Promise((resolve, reject) =>
setTimeout(() => reject('right rejected'), 100));

let p2 = parallelPromise(plast, pfirst);
p2.then(console.log);
```

Result:

```
[ 'left rejected', 'right rejected' ]
```

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

32. Create the function **promiseBarrier**. This function takes an integer strictly greater than zero as a parameter and returns a list with as many functions as indicated by the parameter:

```
let list = promiseBarrier(3);
```

where **list** is **[f1, f2, f3]**.

Each of the functions in the resulting list will receive a single parameter and return a promise that resolves to the value of the parameter. That is, in

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

the variables **x1** and **x2** will always take the same value.

The important detail is that, regardless of when each of the functions in the list returned by **promiseBarrier** is called, they will only resolve their promises once all the functions have been called.

That is, following the previous example, the callback function passed to **f1(x1).then(cb)** will only be executed once **f1**, **f2**, and **f3** have been called.

The idea is that we can use the functions returned by **promiseBarrier** to synchronize different promise chains.

Hints:

- The executor function passed to **new Promise** executes just when it is passed.
- You will likely need to store the parameters of the executor functions *outside* of the executor functions.

Note: the order of the results may vary in the following two examples, but lines ending in **a** should precede those ending in **b**.

Usage example:

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

Promise.resolve(0)
.then(f1)
.then(x => { console.log("c1 s1 b"); return x; })
.then(x => { console.log("c1 s2 b"); return x; })

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

Result:

```
c2 s1 a
c2 s2 a
c2 s3 a
c2 s4 a
c1 s1 b
c1 s2 b
```

Usage example:

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

```
let [f1, f2, f3] = promiseBarrier(3);

Promise.resolve(0)
.then(x => { console.log("c1 s1 a"); return x; })
.then(x => { console.log("c1 s2 a"); return x; })
.then(x => { console.log("c1 s3 a"); return x; })
.then(f1)
.then(x => { console.log("c1 s4 b"); return x; })

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

Promise.resolve(0)
.then(f3)
.then(x => { console.log("c3 s1 b"); return x; })
.then(x => { console.log("c3 s2 b"); return x; })
.then(x => { console.log("c3 s3 b"); return x; })
```

Result:

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

Usage example:

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

Promise.resolve(1).then(f1).then(console.log)
Promise.resolve(2).then(f2).then(console.log)
```

Result:

```
1
2
```

32.$^+$ Enhance the previous exercise. Create the function **timedPromiseBarrier** that allows limiting the maximum time the barrier will be active.

```
let list = timedPromiseBarrier(n, t);
```

This function should do the same as **promiseBarrier**, but if the **n** functions have not been called once **t** milliseconds have elapsed, the following will occur:

- the promises associated with the functions that have already been called will be resolved, and
- the promises associated with the functions that have not been called should be resolved as soon as they are called.

The timer should start at the first time one of the functions returned by **timedPromiseBarrier** is called (i.e., not from when **timedPromiseBarrier** is executed).

33. Create the constructor function **PromisedPriorityQueue**, which ensures the resolution of promises by priority.

```
let ppq = new PromisedPriorityQueue();
```

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

A **PromisedPriorityQueue** can have a promise **p** added to it using the **decorate** method, and this method will return a second promise **q**.

```
let p = new Promise(/*...*/)
let q = ppq.decorate(p, priority)
```

When adding **p** to the queue, a numeric priority must be indicated.

If **p** is the promise with the highest priority of all those in the queue, **q** must be resolved with the same value as **p** when **p** is resolved, or immediately if **p** has already been resolved. Once **q** is resolved, **p** must be removed from the queue, and the next promise with the highest priority must be processed.

The priority of a promise is given by the value **priority**. The higher the value, the higher the priority.

Attention: it may happen that **p** is the promise with the highest priority, but it ceases to be so while waiting for its resolution. In this case, you must wait for the promise that has become of higher priority to be resolved first.

Hint 1: Since you don't know when a promise **q** will be resolved, you may need to store its resolution function.

Hint 2: You may need to use recursion for the process of resolving a promise and finding the next highest priority to resolve.

Note: To remove an item from an array, you can use the function: `Array.splice`

Usage example:

```
let ppq = new PromisedPriorityQueue()

p1 = new Promise((resolve, reject)=>{
        setTimeout(()=>{resolve(1)}, 1000)
})

p2 = new Promise((resolve, reject)=>{
        setTimeout(()=>{resolve(2)},2000)
})

p3 = new Promise((resolve, reject)=>{
        setTimeout(()=>{resolve(3)},3000)
})

(q1 = ppq.decorate(p1,1)).then(console.log)
(q2 = ppq.decorate(p2,2)).then(console.log)
(q3 = ppq.decorate(p3,3)).then(console.log)
```

Result:

```
3
2
1
```

33.+ Enhance the previous exercise. Ensure that if **p** is rejected, **q** is immediately rejected as well.

Usage example:

```
let ppq = new PromisedPriorityQueue()

//use case
p1 = new Promise((resolve, reject)=>{
        setTimeout(()=>{reject('rejected 1')}, 1000)
})

p2 = new Promise((resolve, reject)=>{
        setTimeout(()=>{resolve(2)},2000)
})

p3 = new Promise((resolve, reject)=>{
        setTimeout(()=>{resolve(3)},3000)
})
```

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.

```
(q1 = ppq.decorate(p1,1)).then(console.log).catch(console.log)
(q2 = ppq.decorate(p2,2)).then(console.log).catch(console.log)
(q3 = ppq.decorate(p3,3)).then(console.log).catch(console.log)
```

Result:

```
'rejected 1'
2
1
```

© Universitat Autònoma de Barcelona, 2024.
All rights reserved. No publication or reproduction is allowed without prior written permission from Universitat Autònoma de Barcelona.