NodeJS introduction

But first:

JavaScript Basics

1. How to install it: **`apt-get install nodejs`** or get it from its website `https://nodejs.org/`.
2. What is **`node`**: (1) An asynchronous event-driven JavaScript server-side runtime environment, (2) a javascript interpreter, in particular Google's V8 engine and (3) a non-blocking I/O API

   Let us do a few examples to get up to speed with Javascript basics:

3. Primitive data types: number, bigint, string, boolean, undefined, null, and symbol.

```
> 3 + 3.5; // number (IEEE 754 doubles)
6.5
> 0.1 + 0.2
0.30000000000000004 //0.3 has not an exact representation as a base 2 fraction
> 0/0
NaN
> 1/0
Infinity // Shouldn't this be +/- Infinity !? ...
> 1/-0
-Infinity // ...zero has sign.
> 3456347568934756893467983456n // bigint (has low adoption)
3456347568934756893467983456n
> 3456347568934756893467983456 // see Number.MAX_SAFE_INTEGER
3.4563475689347566e+27
> "hola" + 2.5; // string
'hola2.5'
> true || false; // boolean
true
> undefined; // undefined (unknown variable)
undefined
> null; // null (variable not pointing anywhere)
null
> Symbol("hola") == Symbol("hola"); // symbol (unique identifier)
false
```

4. Template strings: Used to insert/interpolate vars and expressions into Strings

```
> a = 'foo'
> b = `bar`
> `new
line` // multi-line syntax
'new\nline'
> st2=`new \
line`
'new line'
> 'test ' + a + ' ' + b
> `test ${a} ${b}` // interpolation
> `test ${3+3}`
> `test ${f1(3)*3}`
```

5. Variables and the `typeof` operator.

```
> typeof false;
'boolean'
> typeof 0;
'number'
> typeof NaN;
'number'
> foo = 0;
> typeof foo; // note that typeof is an operator, not a function.
'number'
> typeof foo2;
'undefined'
> foo = null;
null
> typeof foo; // Here we would expect 'null'...
'object'
// ...but javascript is full of inconsistencies.
```

6. Equality and Identity (Strict Equality),

```
> 0 == 0;
true
> 0 == false;
true
> 0 === false; // equal value and equal type
false
> (true && false) === false;
true
> true !== false; //try out 0!=false and 0!==false
true
```

7. Truthy and Falsy:

```
> 3 && true; // Values are Truthy if "true in a boolean context".
true
> 0 || false; // And Falsy otherwise.
false
// Falsy values are: false, null, undefined, 0, NaN, "", document.all.
// Everything else is Truthy.
// Good for if(value) { ... } else {...}, but careful here...
> 0 && true;
0
> null && true;
null
> null == false;
false
> NaN == NaN;
false
```

8. Short-circuit evaluation.

```
> false && destroyWorld() // Short-circuit evaluation
false                       || --> returns the first truthy value
> true && destroyWorld()    && --> returns the first falsy one
ReferenceError: destroyWorld is not defined
```

9. Ternary operator.

```
> true ? 1 : 2;
1
> false ? 1 : 2;
2
> (false ? 1 : 2) * 2; // This is an expression.
4
// We cannot do this with an if statement (if is an instruction).
```

We normaly use it to set a variable's value depending on a condition:

```
> isEven = (3 % 2 == 0) ? true : false; //seria equivalent isEven = !(3 % 2) ? true :
false
```

10. Nullish Coalescing Operator (??). returns the right-hand value only if the left-hand value is null or undefined

```
> a = null
null
> a = a ?? 1
1
> b = a ?? 2
1
```

11. Nullish Coalescing Operator assignment (??=): assigns a value only if the variable is null or undefined

```
> a = null
undefined
> b = 1
> a??=b
1
```

12. Simple Objects.

```
> l = [1,2,3,4]; // a 'list'
> t = { a : "hi", b : "bye" } // a simple object (dictionary).
> t.a
> t.b
> t.c = 33
33
> t
{ a: 'Hi', b: 'Bye', c: 33 }
```

13. Built in Objects > console

```
> console.log('Hello World')
Hello World
undefined

> console.log(3*2)
6
```

14. RegExps (test, exec, match)

```
> e = /Pol/ // (c) perl-like regular expression (regex)
> e.test('Hi Pol')
> e = /Pol.*bye$/
> e.test('Hi Pol bye')
> e = /Pol/g
> e.exec('Hi Pol, bye Pol')
> e.lastIndex
> e.exec('Hi Pol, bye Pol')
> e.lastIndex
> e = /Name: (.*),(.*)/    //grouping: extracting data
> res = e.exec('Name: Mar Blau,Pol Nord');
> res[1]
> res[2]
> res = 'Name: Mar Blau,Pol Nord'.match(e)
> res[1]
> res[2]
```

Functions

1. Functions (which are objects too):

```
// Defining functions
> function f1(a) { return a + 1 };
> f1 = function(a) { return a + 1 }; // (d) a function
> f1.toString();
> f1(1);
> f1p = function() { };
> f1p();
undefined;
> f2 = function(x){return(x ? x : 1)} //f2 = function(x){if(x){return x}else{r
> f2p = function(x){return x || 1}
> f2p()
1
> f2p(44)
44
```

2. Functional programming > **Anonymous Functions**.

```
// anonymous functions (lambda functions)
// a.k.a. (IIFE) Immediately Invoked Function Expression`
> function(a){return a+1}
Uncaught SyntaxError: Function statements require a function name
> (function(a) { return a + 1 })(1)
```

3. Functional programming > **Arrow Functions**.

```
// arrow functions (different scope than regular functions)
> // regular function f1p = function(x){return x+1}
> // converted to an arrow
> f1p = (x) => {return x+1}
[Function: f1p]
> f1p(1)
2
> f1p = x => x+1 // automatically performs return a+1 note the lack of braces
> f1p(1)
2
> f1p = x => {x+1} // note the braces
[Function: f1p]
> f1p(1)
undefined
```

4. Functional Programming > *forEach* and *map* looping functions:

```
>//forEach

> l
[ 1, 2, 3, 4 ]
> f1 = function(x){console.log(++x)}
> //using a named function as a parameter
> l.forEach(f1)
1
2
3
4
undefined
> //using an anonymous function as a parameter
> l.forEach(function(x){console.log(++x)})

> //The same but using an arrow function
> l.forEach(x=>console.log(++x))

> //Accessing to the loop index:
> l.forEach((x,i)=>l[i]=l[i]+x) //Changing the content of l
undefined
> l
[ 2, 4, 6, 8 ]

> //The forEach function returns undefined
> l.forEach(x=>x+1)
undefined
```

MODIFY

```
> //map

> l
[ 1, 2, 3, 4 ]
> l2 = l.map(f1) //using the existing f1 function
[ 2, 3, 4, 5 ]
> l3 = l.map(function(x){return x+1}) //using an anonymous function
[ 2, 3, 4, 5 ]
```

CREATE A NEW ARRAY

```
> l4 = l.map(x=>x+1) //using an arrow function
[ 2, 3, 4, 5 ]
```

5. Variables scopes: undeclared (global), declared: *var* (function scope), *const* (block scope), *let* (block scope)

```
// (a) undeclared variable => global scope
> a = 3;
> f1 = function() { a = 4 }; f1();
> a;
4
```

```
// (b) var: declared variable => function scope
> a = 4 //undeclared variable a
> f2 = function() { var a = 2; console.log(a) }; f2();
2
> a;
4
> f3 = function() { var b = 3; console.log(b) }; f3();
3
> b;
ReferenceError: b is not defined

// Another example
f4 = ()=>{
        var a = 1
        { //private scope
                var a = 2
                console.log(a);//2
        }
        console.log(a); //2
}
f4()
2
2
```

```
// (c) let: declared variable => block scope
> a = 4 //undeclared variable a
> f5 = () => {
        let a = 1;
        {
                console.log(a);
                a = 2;
                let b = 3;
                console.log(a);
                console.log(b);
        }
        console.log(a);
        console.log(b);
}
> f_c()
1
2
3
1
Error
```

```
// (d) const: declared variable => block scope, cannot be reassigned within the block
> const a = 6; // same scope as let
> a = 7;
TypeError: Assignment to constant variable.

f6 = () => {
        const a = 1
        {
                const a = 2
                console.log(a);
        }
        console.log(a);
}

> f6();
2
1

f7 = function(){const a=1; {a=2; console.log(a);} console.log(a);}
f7()
Error

> const zoo = {}; // but careful here, only the assignment is constant
zoo.fox = 3
> zoo
{ fox: 3 }
```

6. **Inner functions**: A function defined within another.

```
>f2 = function(x){
        const aux = 1;
        const f3 = function(b) {
                return x + aux + b;
        }
        return f3(x);
}
>f2(2)
5
```

☞ Let's have a look to the NodeJS execution model.

7. **Closures**: A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment) where the function was created.

   See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

```
// The closure
> f2 = function(x){
        const aux = 1;
        const f3 = function(b){ return x + aux + b}
        return f3; //here is the closure
}
[Function: f2]
> let pepito = f2(1)
undefined
```

```
> pepito
[Function: f3]
> pepito(1)
3
> let pepito2 = f2(2)
undefined
> pepito2(3)
6
```

```
// converting f2 and f3 to arrow functions
> f2 = x => {
        const aux = 1;
        return b => x + aux + b
}
```

```
// Another example
const unitAdder = function(startBy){
        let i = startBy;
        return function(){
                i++;
                return i;
        }
}

const adder = unitAdder(2);
console.log(adder());
console.log(adder());
const adder2 = unitAdder(5);
console.log(adder2());
console.log(adder2());

Output:

3
4
6
7
```

```
//Let's shorten the code using arrow functions
const unitAdder = startBy=> ()=>++startBy
```

8. **Callback** functions: function to be executed when an asynchronous operation has finished or an event has triggered (see entry 1).

Classes

1. Classes, just some hints...

```
> t;
{ a: 1,  b: 2 }
a = 33
> t.getA = function() { return a; };
33
> t.getA  = function() { return this.a; };
> t.getA ()
1


//this: is a reference of the object accessing a property of the object
//or executing a method.
> t.getA  = function() {
        //const this = t //implicit
        return this.a;
};
> aFunc = t.getA
> t2 = {a:14}
> t2.getA = aFunc
//it would have been equivalent
//t2.getA = t.getA
> t2 = {a:14, getA: aFunc}
{ a: 14, getA: [Function (anonymous)] }
> t2.getA() //In getA func, this is t2
14
> t.getA() //in getA func, this is t
1
> t.m2 = function(){return this}
> t.m2()
{ a: 1, b: 2,  m2: [Function (anonymous)] }


//We cannot use arrow functions to define the object's methods!!!!!

// Whatever `this` is here...
> t.m3 = () => { this.a = 2 * 3 ;return this }; // ...is what `this` is here.
> console.log(this)


//checkout the value of 'this' when defining a method as a funcion
//and as an arrow.
o.printThis = function(){ console.log(this);}
o.printThis2 = () => console.log(this);
o.printThis()
o.printThis2()
```

2. Classes > Shadowing this:

```
o = { a : 3 }


//shadowing this
o.f2 = function(l){
```

```
        //let this = o
        return l.map(function(x){
                //let this = <general_object>
                return x*this.a})
}

> o.f2([1,2,3,4])
[ NaN, NaN, NaN, NaN ]  //why?

//how do we make it work?
//long solution
o.m1 = function(l){//let this = o
        let self = this;
        return l.map(function(x){
                return x*self.a)})
}


//short solution: using arrow functions
o.f = function(l) {
        //let this = o // Whatever `this` is here...
        return l.map(x => x * this.a) // // ...is what `this` is here.
}


//will this work?
o.m1 = (l) => l.map(x=> x*=this.a)
```

3. Classes > Constructor functions without syntactic sugar to define a class and create instances of this class.

```
//Constructor
const MyClass = function(x){//constructor
        //let this = {}
        this.a = x //{a:x}
        this.b = 2 //{a:x, b:2}
        this.getA = function(){ //{a:x, b:2, getA:function()}
                //let this = this
                return this.a
        }
        //whatever 'this' is here
        this.getB = ()=>this.b //is what 'this' is here
}
//keyword new to create a class instance
let mc = new MyClass(1)
mc.a
mc.getA()
mc.getB()

// The arrow functions are only callable and not constructible,
//i.e arrow functions can never be used as constructor functions.
// Hence, they can never be invoked with the new keyword.
// We can use them to define methods

// Another example
const Counter = function(){
        //let this = {}
        this.count = 1;
```

```
        this.inc = ()=> {this.count++}
        this.getValue = ()=>this.count
}
let c = new Counter()
c.getValue()
c.inc()
c.inc()
c.getValue()
```

4. Classes > Public and private scope (public and private attributes/methods)

```
let Counter = function() {
        let count = 1; //private
        this.inc = ()=>{count++}   //public
        this.getVal = ()=>count;   //public
}
const c2 = new Counter()
c2.increase()
c2.count
c2.getVal()
```

5. Classes > Static fields

```
// Static fields are linked to a class and not to an instance...
// ... so we add them to the constructor as a field.
let Counter = function() {
        let count = 1; //private
        this.inc = ()=>{count+=Counter.MAX_VALUE}   //public
        this.getVal = ()=>count;   //public
}
Counter.MAX_VALUE = 23;
const c2 = new Counter()
c2.increase()
c2.count
c2.getVal()
Counter.MAX_VALUE
```

6. Classes > Class declaration using syntactic sugar.

```
class Counter2 {
  constructor(){
    //let this = {}
    this.count = 1;
    this.inc = ()=> {this.count++}
    this.getValue = ()=>this.count
   }
}

const c2 = new Counter2()
c2.count
c2.inc()
c2.getValue()
```

//This code is equivalent

```
class Counter3 {
        constructor(){
                //let this = {}
                this.count = 0
        }

        inc = ()=>{return this.count++}
        getVal = ()=>this.count
}

let c3 = new Counter3()
c3.count
c3.inc(); c3.inc()
c3.getVal()
```

7. Classes > Class declaration using syntactic sugar > private scope

```
class Counter4 {
        #count = 0
        constructor(){}

        inc = ()=>{return this.#count++}
        getVal = ()=>this.#count
}

let c4 = new Counter4()
c4.count //undefined
c4.inc(); c4.inc()
c4.getVal()
```

8. ☞ TODO: **Inheritance**

Asynchronous programming

1. **Callback** functions: function to be executed when an asynchronous operation has finished or an event has triggered. Your first contact: JQuery

   See full example at https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_click

```
$(document).ready(function(){ //function executed when the DOM of the page is loaded.
 $("p").click(function(){ //function executed when the paragraph is clicked.
  $(this).hide();
 });
});
```

```
//Another example
//Let's use a callback function that will be called when
//a random generated number is even.
getRandomInt = function(max,cb) {
        const randInt = Math.floor(Math.random() * max);
        if(randInt % 2 != 0){cb()}
        return randInt
}

console.log(getRandomInt(5,function(){console.log("The number is odd")}));
```

```
//Now using arrows:
getRandomInt2 = (max,cb) => {
        const randInt = Math.floor(Math.random() * max);
        if(randInt % 2 != 0){cb()}
        return randInt
}

console.log(getRandomInt2(5,()=>console.log("The number is odd")));
```

```
//Now the callback function has to print "The number XX is odd".
getRandomInt3 = (max,cb) => {
        const randInt = Math.floor(Math.random() * max);
        if(randInt % 2 != 0){cb(randInt)}
        return randInt
}
console.log(getRandomInt3(5,(x)=>console.log(`The number ${x} is odd`)));
```

2. Asynchronous programming:

   ☞ Let's have a look to the NodeJS execution Model: Event Loop

   (see:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop)

   A good video resource: https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=773s.

   `setTimeout(functionRef, delay, param1, param2, /* …, */ paramN)`

```
setTimeout(()=>{console.log("Delayed for 3s.")}, 3000)
console.log("here");
```

```
//another example
setTimeout((a,b)=>{console.log(`Delayed ${a} ${b}`)}, 3000, 'Hello', 'World')
console.log("here");
```

NodeJS Non-blocking I/O API

1. NodeJS:

   Now, lets take a look at the other half of Node (the non-blocking I/O API). We can start by taking a look at the object **console**, which is available by default.

```
> console
{ log: [Function],  info: [Function],  warn: [Function],  error: [Function],
        dir: [Function],  time: [Function],  timeEnd: [Function],  trace: [Function],
        assert: [Function],  Console: [Function: Console] }
> console.log("hola");
'hola'
undefined
> a = function () {console.trace("hola"); };
> a();
```

   Then we also have other objects (modules), which we have to import. There are two ways to import the modules, using CommonJS:

```
> const fs = require('fs');
//checkout the attributes and menthods of the module fs
> fs. //press tab
```

   Using ES6(ECMAScript6) Modules syntax:

```
import {readFileSync,writeFileSync} from 'fs';
```

2. Synchronous I/O (blocking).

```
> fs.writeFileSync('a.txt', 'Hello World\n'); console.log('done');
undefined
> fs.readFileSync('a.txt','utf8');
'Hello World'
> fs.readFileSync('a.txt') + '';
// Not asynchonized code yet... we ask for something and we get it.
// Now let us try with asynchonized code...
```

3. Asynchronous I/O (non-blocking) > Callback functions

```
> const fs = require_('fs')
> fs.readFile('a.txt','utf8',(err,data)=>{console.log(err? err: data)})
> console.log("here")
'here'
'hellow world'

// ATTENTION HERE: THE FUNCTION CALL HAS NOT BLOCKED EXECUTION:
```

```
//We can ignore the callback parameters:
fs.readFile("a.txt", 'utf-8', ()=>{console.log("In the cb");})

//another example:
import { readFile, writeFile } from 'node:fs';

readFile('a.txt', 'utf-8', (err, data) => {
        console.log("Done reading");
        if (err) throw err;
        writeFile('b.txt', data,(err)=>{
                console.log(err? err : "Done writing");
        })
});
console.log("Task done");
Output:
"Task done"
"Done reading"
"Done writing"
```

4. Asynchronous I/O (non-blocking) > Streams

```
// Or with streams
> rs = fs.createReadStream('a.txt'); // Source
> ws = fs.createWriteStream('c.txt'); // Destination, Sink
> rs.pipe(ws); //automatically reads from rs and writes to ws.
```

5. Asynchronous I/O (non-blocking) > Streams > Events

Let's create **`streams.js`** and run as: **`node stream.js`** (we cannot type this code on the node shell as the **`.on`** method call placed in a new line, it does not work)

```
const fs = require('fs')
const rs = fs.createReadStream('a.txt', 'utf8')
const ws = fs.createWriteStream('c.txt')
rs
.on('end',()=>console.log('No more data to read.'))
.on('data',chunck => console.log(`Read: ${chunck}`))
.pipe(ws)
.on('finish',()=>console.log("No more data to write"))
.on('close',()=>console.log("Stream closed"))
```

HTTP Server

1. Let us move to a source file... **node server.js** and let us create an HTTP server sending a plain text to the client.

```
const http = require('http')
const port = 8000

const server = http.createServer((req, res)=>{
        res.writeHead(200, {"content-type": 'text/plain'})
        //res.write('Hello Wold'); res.end()
        res.end('Hello Wold!!')
})
server.listen(port)

console.log('Server running at http://localhost:8080/');
```

2. Shall we use **nodemon** application? **Nodemon** is a tool that automatically restarts the node application when detects changes in the files in the directory.

   npm install -g nodemon

   npm list -g

3. HTTP Server > sending a file to the client using streams:

```
        const http = require('http');
        const fs = require('fs');

        http.createServer((req, res) => {
                res.writeHead(200, {'Content-Type' : 'text/html'});
                const rs = fs.createReadStream('index.html')
                rs
                .pipe(res)
                .on('finish',()=>res.end()) //equivalent to: .on('finish',res.end)
        }) .listen(8080);
        console.log('Server started at port 8080');
```

```
//Let us try the server callback running some code that blocks execution...
for (i=0; i < 1E10; i++) { i + 1 };
// Remark: what happens if this loop takes a while?
```

ExpressJS

Express website: https://expressjs.com/

1. Creating an HTTP Server using ExpressJS:

```
const express = require('express')
const path = require('path')
const app = express()
const port = 3000

app.listen(port, () => {
        console.log(`Example app listening on port ${port}`)
})
```

2. Basic Routing: *Routing* refers to how an application's endpoints (URIs) respond to client requests (https://expressjs.com/en/guide/routing.html).

```
const express = require('express')
const path = require('path')
const app = express()
const port = 3000

app.get('/', (req, res) => {
        res.sendFile(path.join(__dirname,"public/index.html"))
})

app.listen(port, () => {
        console.log(`Example app listening on port ${port}`)
})
```

3. Routing > Using a middleware to serve static files:

```
const express = require('express')
...
//move index.html to public dir
app.use(express.static(path.join(__dirname,'public')))

app.listen(port, () => {
        console.log(`Example app listening on port ${port}`)
})
```

4. Routing > Handling a REST API > Route Parameters:
   (https://expressjs.com/en/guide/routing.html)
   Example for:

   Request URL: http://localhost:3000/info/1
   Route path: /info/:studentID
   req.params: { "studentID": "1"}

```
const express = require('express')
const path = require('path')
const app = express()
const port = 3000

const students = {
        1: {name: 'Pol Llop', mail: 'pol@uab.cat', subjects: {
                        1:{code:'TDIW', mark:9},
                        2:{code:'STW', mark:10},
        }},
        2: {name: 'Mar Roca',mail: 'mar@uab.cat', subjects: {
                        1:{code:'TDIW', mark:8},
                        2:{code:'STW', mark:9},
        }},
        3: {name: 'Nil Lopez',mail: 'nil@uab.cat', subjects: {
                        1:{code:'TDIW', mark:7},
                        2:{code:'STW', mark:8},
        }},
}

app.use(express.static(path.join(__dirname,'public')))

app.get('/info/:studentID', (req, res)=>{
        console.log(`studentID: ${req.params.studentID}`);
        res.json(students[req.params.studentID])
})

app.listen(port,()=>{console.log(`Server listening at ${port}`)})
```

Another example for:

Request URL: http://localhost:3000/mark/1/subject/2
Route path: /mark/:studentID/subject/:subjectID
req.params: {“studentID”:“1”,“subjectID”:“2”}

```
app.get('/mark/:studentID/subject/:subjectID', (req, res) => {
  console.log(`student: ${req.params.studentID} subject: ${req.params.subjectID}`);
  res.json(students[req.params.studentID]['subjects'][req.params.subjectID])
})
```

5. Routing > Handling a query request:

☞Query strings are not part of the route path.

Request URL: http://localhost:3000/mark2?studentID=1&subjectID=2
Route path: /mark2
☞ **req.query**(Query request): “studentID=1&subjectID=2”

```
app.get('/mark2', (req, res)=>{
  res.json(students[req.query.studentID]['subjects'][[req.query.subjectID]])
})
```

6. Now refactoring the code into MVC architecture

```
//models/students.js
const students = {
        1: {name: 'Pol Llop', mail: 'pol@uab.cat', subjects: {...}
        ...

module.exports={students}
```

```
//controller/students.js
const studentsModule = require('../models/students.js')

const students = studentsModule.students;

const getInfo = (req, res)=>{
        res.json(students[req.params.studentID])
}

const getMark = (req, res) => {
        res.json(students[req.params.studentID]['subjects'][req.params.subjectID])
}

module.exports = {getInfo, getMark}
```

```
//app.js
const studentsController = require('./controllers/students.js')

app.get('/info/:studentID', studentsController.getInfo)
app.get('/mark/:studentID/subject/:subjectID', studentsController.getMark)
```

Promises

- A **Promise** is an **object** representing the eventual completion or failure of an asynchronous operation[1] [2].
- **At the time the promise is returned to the caller, the operation often isn't finished**
  ☞ Promises are *futures* implementing the *observer pattern*
- Promises can have three different states:

  ```
  Promise { <state>: "pending" }
  ```

  Where "state" can be:

  pending : The promise has been created, and the asynchronous function that created the promise has not succeeded or failed yet.

  fulfilled : The asynchronous function has succeeded.
  When a promise is fulfilled, its `then()` handler is called.

  rejected : The asynchronous function has failed.
  When a promise is rejected, its `catch()` handler is called.

- A promise is **resolved** if it is settled, that is if has either been fulfilled or rejected.
- The promise object provides methods (callbacks) to handle the eventual success or failure of the operation. **We attach those callbacks to the promise** instead of passing callbacks into a function. To do so, we use the functions `then(callback)` and `catch(callback)`.
- What do the methods: `then(callback)` and `catch(callback)` return?
  ☞ Always a **new pending promise**:
  - For `p2 = p.then(cb1)`, `p2` fullfills to:

    ```
    //p2 = p.then((x)=>{... return x}) -> p2 fulfills to x
    //p2 = p.then((x)=>{...}) -> p2 fulfills to undefined
    //p2 = p.then((x)=>{ return p4 }) -> p2 will be fulfilled/rejected
      when p4 fulfills/rejects to its values.
    //p2 = p.then((x)=>{...throw err ...}) p2 rejects to err
    //p2 = p.then((x)=>{...}) if p rejects, p2 rejects to p's rejected value
      and the callback is registered.
    ```

  - For `p3 = p.catch(cb2)`, `p3` fullfills to:

    ```
    //p3 = p.catch((x)=>{....}) if p fulfills -> p3 fulfills to p's fulfillment value
    //p3 = p.catch((x)=>{return x}) if p rejects -> p3 fulfills to
     catch callback's returning value (x)
    //p3 = p.catch((x)=>{ return p2 }) -> if p rejects, p3 will be fulfilled/rejected
        when p2 rejects or fulfills
    //p3= p.catch((x)=>{...throw err ...}) p3 rejects to err
    ```

[1] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Promises (use)

1. Let's read a file using callbacks and using promises:

```
//using callbacks
const fs = require('fs')
fs.readFile('a.txt','utf8',(err,data)=>{
        console.log(err ? err: data)
});
```

Using promises:

```
const fsp = require('fs/promises')
let p = fsp.readFile('a.txt', 'utf8')
let p2 = p.then((data)=>console.log(data))
let p3 = p.catch(console.log)
```

NOTE: that p.then() and p.catch() return both a promise!

2. Let's see how the state of the promise changes.

```
const fsp = require('fs/promises')
const p = fsp.readFile('a.txt', 'utf-8') // returns a promise object (a future)
const p2 = p.then(x=>{console.log(x); return x}) //adds a cb to p and returns p2
const p3 = p.catch(console.log) //adds a cb to p and returns p3
console.log(p);console.log(p2);console.log(p3);

setTimeout(console.log, 1000, p)
setTimeout(console.log, 1000, p2)
setTimeout(console.log, 1000, p3)
```

Promise chains

We chain promises for two different purposes:

(a) To execute two or more asynchronous operations sequentially, i.e. the second operation is executed once the first one is finished.
(b) To transform values.

```
p.then(cb1).then(cb2).then(cb3).catch(cb4).then(cb5).finally(()=>{...})
```

From this chain:

- Methods `Promise.then` and `Promise.catch` **return a promise** which has the methods `Promise.then` and `Promise.catch`. We say a promise is *thenable*.
- In the above promise chain we have created six promises, to perform six asynchronous operations sequentially. Therefore, `cb1`, `cb2`, `cb3` and `cb4` have to return a promise!
- `p` repersents an asynchronous operation and when resolved/rejected, it will be executed the first most suitable callback in the chain: `cb1` if `p` is fulfilled, or `cb4` if rejected. `cb1` creates a promise related to an asynchronous operation and must return that promise, which will be chained to `then(cb2)....catch(cb4)....` chain. When **the promise returned by** `cb1` will be resolved/rejected, it will be executed the first most suitable callback in the chain: `cb2` if the promise returned by `cb1` is fulfilled, or `cb4` if it is rejected rejected, and so on.

- All the above handlers, **cb1**, **cb2**,... do not go to the *event queue* but to a *microtask queue*[3] which is accessed when the execution stack is empty before polling on the event's queue.
- If **cb1** .. **cb5** (aka handlers) perform an asynchronous operation, they must create and return a promise. By doing so we are chaining promises and avoiding having *floating* promises.

3. Chaining: Use case for reading a file

```
fsp = require('fs/promises')
fsp.readFile('a.txt', 'utf8') // promises are ``then-able''...
.then((data)=>console.log(`data: ${data}`))
.catch((err)=>console.log(`data: ${err}`))
```

4. Chaining: use case of three asynchronous depending operations:

   Let us "remove" a file asynchronously the "old" fashioned way.

```
const fs = require('fs');
fs.readFile('a.txt', (err, res) => {
        fs.writeFile('b.txt', res, (err) => {
                fs.unlink('a.txt', (err) => {
                        console.log("done");
                });
        });
}); // Too many nested functions (code becomes unreadeable/error-prone)
```

   Chaining two asynchronous operations where each subsequent operation starts when the previous operation succeeds using callbacks is called "pyramid of doom".

5. Promises help with this with a **promise chain**. How do we get a promise chain?

   Let us copy a **a.txt** to **b.txt** using promises **without chaining promises**.

```
        const fsp = require('fs/promises')
        let p = fsp.readFile('a.txt','utf8')
        p.then((data)=>{
                let p2 = fsp.writeFile('b.txt', data)
                p2.then(()=>{
                        let p3 = fsp.unlink('a.txt')
                        p3.then(()=>{console.log('Done');})
                        p3.catch((err)=>console.log(err))
                })
                p2.catch((err)=>console.log(err))
        })
        p.catch((err)=>{})
```

   Let us copy a **a.txt** to **b.txt chaining promises**.

```
        fsp = require('fs/promises')
        fsp.readFile('a.txt','utf8')
        .then(data=>fsp.writeFile('b.txt', data))
        .then(()=>{return fsp.unlink('a.txt')})
        .then(()=> console.log('done'))
        .catch(console.log)
```

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining

6. Chaining: Attention with the chaining!

   Taking into account the previous example:

```
fsp = require('fs/promises')
fsp.readFile('noFile.txt','utf8')
.then(data=>fsp.writeFile('b.txt', data))
.catch(console.log)
.then(()=>{return fsp.unlink('a.txt')})
.then(()=> console.log('done'))
.catch(console.log)
```

7. Syntactic sugar: **async/await**

```
let moveFile = async ()=>{
        try{
                let data = await fsp.readFile('a.txt','utf8')
                await fsp.writeFile('b.txt', data)
                await fsp.unlink('a.txt')
                console.log("done");
        }catch(err){
                console.log(err);
        }
}

moveFile()
```

8. Syntactic sugar: async/await > let's see how it works:

   Using a `try catch` block:

```
let moveFile = async () => {
        console.log(1);
        try {
                let x = await fsp.readFile('a.txt', 'utf8')
                console.log(2);
                await fsp.writeFile('b.txt', x)
                await fsp.unlink('a.txt')
        } catch (err) {
                console.log(err)
        }
        console.log(3)
}
p = moveFile()
console.log(0)
setTimeout(console.log,1000,p)

> 1
> p<pending>
> 0
> 2
> 3
> p<fulfilled to: undefined>
```

   Using a `catch`

   ☞ The async function returns a Promise.

```
const moveFile2 = async () => {
        console.log(1);
        const data = await fsp.readFile('a.txt', 'utf-8')
        console.log(2);
        await fsp.writeFile('b.txt', data)
        await fsp.unlink('a.txt')
        console.log('Done!!!')
        console.log(3)
}
const p = moveFile2().catch(console.log)
console.log(p);
setTimeout(()=>{console.log(p)}, 1000)
```

Promises (creation)

1. What is a promise?

```
        let p = new Promise(executor) // Answer: an object
```

2. And the executor function is...

```
let executor = function (resolve, reject) {
//here goes the asynchronous execution
  doSomethingAsync((err, data)=>{
    if(!err){
      resolveFunc(data)
    }else{
      rejectFunc(err)
    }
  })
}
```

☞ The executor runs in the constructor.

☞ The promise constructor takes an executor function that lets us resolve or reject a promise manually.

3. How is this useful?

```
        p = new Promise((resolve, reject) => {
                // do stuff
                setTimeout(() => {
                        if (true) { // ok?
                                resolve("ok")
                        } else { // error
                                reject("ko")
                        }
                }, 1000);
        })

        p.then(console.log)
        p.catch(console.log)

        // or
        p.then(console.log, console.log)
```

```
// or (not exactly the same)
p.then(console.log).catch(console.log)
// first callback able to handle fulfillment or rejection takes the result.
p.catch(console.log).then(console.log)
```

4. Pre-resolved promises:

```
Promise.resolve(23).then(console.log))
Promise.reject("error").catch(console.log))
```

5. `finally`

```
p.finally(f) // Same thing as p.then(f,f), except...
// (a) f receives no arguments
// (b) can use anonymous functions
```

Module Pattern

1. Remembering: Private Scope

   We use it to encapsulate things (variables, functions, objects, etc...) for access control (to hide things). This pattern is used when we export an API.

```
// A 'var' scope is a whole function.
> f1 = function() { var a = 1; { var b = 2; } return b; }
// Careful here: brakets do not have any effect here.
// We can use them with a label with break and continue.
// E.g, block: { break block; }

// Private scope with a block + let
test = function(){
        let a = 2
        { //private scope
                let b = 2
                a *= b
        }
        //console.log(b); //undefined
        return a;
}
test();

//Private scope with an anonymous function
test2 = function(){
        let a = 3;
        //private scope
        (function(){ //private scope
                let b = 4
                a *= b
        })();

        //console.log(b); //undefined
        return a;
}
test2()
```

2. Module Pattern:

   We use it to export out things (vars, function objects) from a private scope to use them.

```
//the structure of a module pattern is:
(()=>{ //defining the private scope
        return {} //returns an Object
})()
```

An example:

```
        const myModule = (()=>{
                let a = 3
                const multiply = () => {let b = 3; a*=b}
                const getA = () => a

                return {multiply: multiply, getA:getA}
        })()

        myModule.multiply()
        console.log(myModule.getA())
```

☞ Is there any closure here?

When the property/ties of the returning object matches the names of classes/functions, you just need to specify the property without its value:

```
return {multiply, getA}
```

Inheritance

1. Prototype-based inheritance:

```
// Main reasons to do inheritance:
// (A) As a subtyping mechanism; Interfaces: E.g., this function takes
//     things that implement this interface.
// (B) Code-reuse: use existing code, override some parts.
//
// In javascript the first option makes little sense, because it is a
// weakly-typed language (there are few restrictions on what goes where).
// Then, only option (B) remains !?.

// Prototype-based programming
// Every object has a pointer to another object: its prototype.
// When a method or field (same thing) is not found in an object, it is
// looked for in its prototype, recursively.
> a = [1]
> a.__proto__ // [press intro]
// We can see which is the prototype of a
> a.__proto__. // [press tab]
// We can see all methods inherited by list 'a'.
> a.__proto__.__proto__ ...
```

2. Prototype-based inheritance for simple objects:

```
// We can understand prototypes as a way to obtain a quick copy of an object
// to make changes... (reuse code).
// Example: Simple Object inheritance
let vehicle = {tires: 25, wheel: true };
let car = {tires: 100, __proto__: vehicle };
let limousine = {tires: 200, __proto__: car};
// This can be done too with Object.create(), but it is not necessary
// to understand inheritance.
```