NodeJS introduction

First session

1. How to install it: either you get it from its website or **`apt-get install nodejs`**.
2. What is **`node`**: a javascript interpreter, in particular Google's V8, plus a non-blocking I/O API.
3. Let us do a few examples to get up to speed with Javascript basics:
4. 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
> 0/0
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
```

5. Variables and the `typeof` operator.

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

6. Equality and Identity (Strict Equality), Truthy and Falsy.

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

7. Short-circuit evaluation and ternary operator.

```
> false && destroyWorld() // Short-circuit evaluation
false
> true && destroyWorld()
ReferenceError: destroyWorld is not defined
> 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.
```

8. Objectes and functions (which are objects too):

```
> l = [1,2,3,4]; // (a) a 'list'
> t = { a : "hi", b : "bye" } // (b) a 'map' or an 'object'.
> t.a
> t.b
> m = /3/ // (c) perl-like regular expression (regex)
> m.test(2)
> m.test(3)
> m = /3/g
> m.exec('33')
> m.exec('33')
> m.exec('33')
> function f1(a) { return a + 1 };
> f1 = function(a) { return a + 1 }; // (d) a function
> f1.toString();
> f1(1);
> f1p = function() { };
> f1p();
undefined;
```

9. Template strings

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

10. Functional programming

```
> l.map(f1)
> l.forEach(f1) // just for the side effects (nothing in this case)
// anonymous functions (lambda functions)
> (function(a) { return a + 1 })(1)
// arrow functions (different scope than regular functions)
> l.map(a => {return a + 1})
> l.map(a => a + 1)
```

11. Variable scopes

```
// (a) undeclared variable => global scope
> foo = 3;
> f_a = function() { foo = 4 }; f_a();
> foo;
4
// (b) declared variable => function scope
> f_b = function() { var foo = 5 }; f_b();
> foo;
4
> f_b = function() { var bar = 1 }; f_b();
> bar;
ReferenceError: bar is not defined
// (c) declared variable => block scope
> f_c = function() {
    let qux = 1;
    {
      let qux = 2;
    }
    // qux?
  };
// (d) declared variable => block scope, cannot be reassigned
> const quux = 6; // same scope as let
> quux = 7;
TypeError: Assignment to constant variable.
> const zoo = {}; // but careful here, only the assignment is constant
zoo.fox = 3
```

```
> zoo
{ fox: 3 }
```

12. Closures

```
// We can define a function inside another function (not a closure).
> f2 = function(a) {
    var f3 = function(b) {
      return b + 1;
    }
    return f3(a);
  }
> f2(3);

// And the closure...
> f2 = function(a) {
    return function (b) { return a + b; };
  }
> f2
[function]
> f2(1)
[function]
> f2(1)(3)
4
// Another example
> genRandom = function(seed) {
    var state = seed;
    return function() {
      // This is (was?) the congruential PSNR used glibc
      state = (state * 1103515245 + 12345) % (2*(1<<30));
      return state;
    };
}
> genRandom(1)
// -> Comment on LFSRs
```

13. Classes, just some hints...

```
> t;
{ a: 'hola',  b: 'adeu' }
> t.method = function() {};
> t.method();
> t.method = function() { return this.a; };
'hola'
// Javascript is quite more complicated than what it seems...
// Be careful with arrow functions here.
> t.method2 = () => { 2 * 3 ; return this };
> t.method3 = () => this;
// More about this `this' later.
```

14. 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();
```

15. Then we also have other objects (modules), which we have to import.

```
> os = require('os');
{ endianness: [Function], hostname: [Function], loadavg: [Function],
  uptime: [Function], freemem: [Function],totalmem: [Function],
  cpus: [Function], type: [Function], release: [Function],
  networkInterfaces:[Function], arch: [Function], platform: [Function],
  tmpdir: [Function], tmpDir: [Function],
  getNetworkInterfaces: [Function: deprecated], EOL: '\n' }
> os.type
[function]
> os.type();
'Linux'
> os.hostname();
'laptop'
> os.hostname.toString();
'function () { [native code] }'
> fs = require('fs');
// Per cert...
> require.toString();
```

16. More on I/O.

```
> fs.writeFileSync('hola.txt', 'text\n');
undefined
> fs.readFileSync('hola.txt');
<Buffer 74 65 78 74>
> fs.readFileSync('hola.txt') + '';
> fs.unlinkSync('hola.txt');
// Not asynchonized code yet... we ask for something and we get it.
// Now let us try with asynchonized code...
> f = function() { console.log('done'); }
> fs.readFile('hola.txt', f); console.log('here');
// ATTENTION HERE: THE FUNCTION CALL HAS NOT BLOCKED EXECUTION
> f = function(err, result) { console.log(result); }
> f = function(err, result) { fs.writeFile('hola2.txt', result); }
> f = function(err, result) { fs.writeFile('hola2.txt', result,
  function() {console.log('yes');}); console.log('task done?'); }
// Or with streams
> a = fs.createReadStream('hola.txt'); // Source
> b = fs.createWriteStream('hola3.txt'); // Destination, Sink
> a.pipe(b);
```

17. Let us move to a source file... `node file.js`

```
http = require('http');
http.createServer(null).listen(8080);
```

```
--
http = require('http');

f = function(request, response) {
        response.writeHead(200, { "Content-Type" : "text/plain" });
        response.write("Hello World\n");
        response.end();
}

http.createServer(f).listen(8080);

console.log('Server running at http://127.0.0.1:8080/');
```

```
// Let us try with something that blocks execution...
for (i=0; i < 1E9; i++) { i + 1 };
// Remark: what happens if this loop takes a while?
```

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

f = function(request, response) {
        response.writeHead(200, { 'Content-Type' : 'text/plain' });
        fs.readFile('Hola.txt',
                function(err, result) {
                        response.write(result);
                        response.end();
                }
        );
}
```

```
url = require('url');

url.parse(request.url).pathname;
```

Second session

An interesting example to start with:

```
o = { a : 3 }

o.f = function(l) {
        return l.map(x => x + this.a) // why 'this' is 'o' and not 'l'?
}
```

For this session we ignore `let` and `const`, and all syntactic sugar for classes, and focus on understanding what is going on.

1. Private Scope

```
// 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; }

// To create a private scopr we create a function and we run it:
private = function () {
}
private();

// To do it right: lambda (anonymous) function + run it right there.

// This:
> private = function() { /* private scope */ }
> private();

// Turns into:
> private = (function() { /* private scope */ })
> private();

// Hence:
>   (function() { /* private scope */ })();

// Example 1:

var a = 1;

(function() {
        var privateVariable = 1;
        // we do something with this variable...
 })();

// We cannot access this the private variable here

// Example 2:
function test() {
var a = 1;

(function(){
var b = 23;
a *= b;
})();

return a;
}

// Example 3:

// We have this piece of code:

var a = '<html><head></head><body>some text</body></html>'

// [...]

var m = /<[^>]*>/g

// list tags
while(true) {
        var t = m.exec(a)
        if (! t) { break }
        console.log(t)
}

// [...]

// Variables 'm' and 't' are leaked:

> typeof m
'object'
> typeof t
'object'

// We want to hide them so that nobody can touch them.

var a = '<html><head></head><body>sometext</body></html>'

(function() {

var m = /<[^>]*>/g

// list tags
while(true) {
        var t = m.exec(a)
        if (! t) { break }
        console.log(t)
}

})()

> typeof m
'undefined'
> typeof t
'undefined'
```

2. Module Pattern

```
var testModule = (function() {
        var a = 1;
        return {
                increase : function() { a++; },
                value: function() { return a; }
        };
})();
```

3. Revealing Module Pattern

```
var testModule = (function() {
        var _a = 1;

        var _increase = function () { _a++; },
        var _value = function() { return _a; }

        return {
                increase :  _increase,
                value:  _value
        };
})();
```

4. Classes

```
> f1 = function() { return this.a; }
> o1 = { a :  33, b : f1 }
> o2 = { a : 44, b : f1 }
// We use 'this' to obtain the object where function 'b' hangs from.
> o1.b()
> o2.b()
// With this we can make fields and methods.

// Constructor + keyword new
function Counter() {
        this.a = 1;
}
o3 = new Counter();
o3.a

// In Counter we can add function too...
function Counter() {
        this.a = 1;
        this.increase = function () { this.a++; },
        this.value = function() { return this.a; }
}
o3 = new Counter();
o3.increase()
o3.value()
```

5. Static fields

```
// These are variable link to a class and not to an instance...
// ... so we add them to the constructor as a field.
Counter.MAX_VALUE = 23;
```

6. 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-type 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 tab]
// We can see all methods inherited by list 'a'.
> a.__proto__.__proto__ ...

// A small example:
function ClassA() {
        this.a = 1;
}
ClassA.prototype = { b : 2 };
// Note that '{ b : 2 }' is an object, not a constructor.
o = new ClassA();

// Another example:
function ClassA() {
        this.a = 1;
}
function ClassB() {
        this.b = 2;
}
ClassA.prototype =  new ClassB();
o = new ClassA();
// o.a, o.b

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

Third session

Streams and event handlers (Zip Archiver example)

Introduce example case: we want a pice of code that takes some files, and sends them to a client in a single zip file. We want this to happen on the fly. We do not want to store the zip file anywhere.

1. First of all, we need to create zip files from Node somehow...
   NPM (node package manager) +100k packages!
   `npm install archiver` (a module to create zip files), this installs it in the local project folder at **`node_modules`**.
2. We take a look at the API and the usage example...
3. We will do this:
   (a) Create a server
   (b) Create a list of readStreams that we want to append to the zip file (**`names.map`** with **`fs.createReadStream`**).
   (c) Create a function that takes the streams and creates the zip file (**`zip.append`**).
   (d) We register a callback on **`'finish'`** in the **`zip`** archiver to close the **`response`**.

```
var http = require('http');
var archiver = require('archiver');

// needs to be switched to a database
var fs = require('fs');

function sendFilesInAZip(streams, response) {
  // Create new zip file
  var zip = archiver('zip');

  // Set up some callbacks
  zip.on('finish', function() {
    console.log('Zip file has been sent, with a total of '
      + zip.pointer() + ' bytes.');
    response.end();
  });

  // Put the result in...
  zip.pipe(response);

  console.log('Setting up zip file contents.');

  // Queue files to archive
  streams.forEach(function(s) {
    zip.append(s.stream, { name: s.name });
  });

  // Signal end of queue
  zip.finalize();

  console.log('Zip creation has been started.');
}

function serverFunction(request, response) {
  response.writeHead(200, {'Content-Type': 'application/zip',
   'Content-Disposition' : 'attachment; filename=files.zip' });

  // pop ids from db using fields from request
  var names = ['file1.txt','file2.txt'];
  var streams = names.map(function(name) {
    var s = {
      name : name,
      stream : fs.createReadStream(name),
    };

    return s;
  });

  sendFilesInAZip(streams, response);
};
http.createServer(serverFunction).listen(8080);
console.log('Server running at http://127.0.0.1:8080/');
```

4. Test via **wget** and **unzip -t**

Routing (Express)

```
// HOWTO: create a .js file and put things at the right places
// 1-
var express = require('express');

var app = express();

app.listen(8080);

// 2-
app.get('/link.txt', function(req, res) {
        console.log('here');
        res.end();
}

// 3- Subset of regex (no . or -)
app.get('/lin(k|e).txt', ...

// 4- perl-like regex
app.get('/[A-z][a-z]*[^0-9]$/', ...

// -> First matching function takes the job

// 5- To deal with parameters (things after ?)
app.get('/link', function(req, res) {
        res.end("ID was: " + req.query.id);
}

// 6- Middleware (this goes before the .get)
app.use('/link', function(req, res, next) {
        console.log(req.query.id);
        next();
}

// 7- This creates a middleware that serves static files
app.use('/public', express.static(path.join(__dirname, 'public')));

// 8 - This is a middleware that parses cookies
var cookieParser = require('cookie-parser')
```

```
app.use(cookieParser())

request.cookies = {name : key}

// 9 - This is a middleware that parses bode requests
var bodyParser = require('body-parser')
app.use(bodyParser())

request.body = {name : key}
```

// See express.json and res.json to send json back and forth.

Concurrency tests

1. Recall how to create a server:

```
var http = require('http');

f = function(request, response) {
        response.writeHead(200);
        response.write("hola!");
        response.end();
}

http.createServer(f).listen(8080);

console.log("Server running!");
```

2. Ho provem `class-ff.sh`
3. Test this server with a stress testing tool (explain ApacheBenchmark).

   `ab -n 100 localhost:8080/`

   `-n` is the number of requests
4. We make it wait a bit more:

   `for (var i = 0; i < 1E7; i++) { i * 2 };`

   `ab -n 100 -c 2  localhost:8080/`

   `-c` is the concurrency level (how many requests in parallel)

   Note the time per request: 10.259 ms
5. Ok, let us complicate things a bit further...

```
// function that writes the response
var r = function() {
        response.write("Time is " + new Date().toString());
        response.end();
};

// active waiting
var d = new Date();
while(new Date().getSeconds() == d.getSeconds()) { }
r();
```

`ab -n 10 localhost:8080/` (why does it say a 1000?)

`ab -n 10 -c 2 localhost:8080/` (why does it say a 2000?)

```
// event based waiting
setTimeout(r, 1000 - new Date().getMilliseconds());
```

`ab -n 10 localhost:8080/` (why does it say a 1000?)

`ab -n 10 -c 2 localhost:8080/` (why does it say a 1000?)

6. We need to keep response time under control.

Fourth session

Promises (use)

1. What is a promise?

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

   (let us put a pin in this 'executor' thing for now)

2. (again) What is a promise?

   ```
   const fsPromises = require('fs').promises; // like fs but callbacks -> promises
   // fs.readFile --> fsPromises.readFile

   p = fsPromises.readFile('a1.txt'); // returns a promise object (a future)
   p // show p
   p.then((res) => { console.log(res) }); // add callback to promise
   p.catch((err) => { console.log(err) }); // add callback to promise
   ```

3. Example (let us copy a **a1.txt** to **a2.txt**)

   ```
   p = fsPromises.readFile('a1.txt');
   let callback = function(res) { fsPromises.writeFile('a2.txt', res); }
   p.then(callback);
   // or...
   p.then(res => fsPromises.writeFile('a2.txt', res));
   ```

4. Two questions... first: promises are "then-able"...

   ```
   p.then(console.log);
   ```

   but why do we not see the **then** and **catch** functions in **p**?

5. Second: what are these things that we keep seeing when we call **then**? ⟹ Promises!?!?

6. Example: let us add a callback to **writeFile**.

   ```
   p = fsPromises.readFile('a1.txt');
   callback = function(res) {
           let p2 = fsPromises.writeFile('a2.txt', res);
           p2.then(() => console.log("done"));
   }
   p.then(callback);
   ```

   ```
   p = fsPromises.readFile('a1.txt');
   p.then(res => fsPromises.writeFile('a2.txt', res).then(() => console.log("done")));

   // ...but better
   fsPromises.readFile('a1.txt')
           .then(res => fsPromises.writeFile('a2.txt', res))
           .then(() => console.log("done"));
   ```

   Note: if result of function given to **then** does not return a **Promise**, returned value is "promisified" automatically.

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) {
        // do stuff

        if (ok) {
                resolve(result)
        } else { // error
                reject(result)
        }
}
```

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. Is even?

```
let isEven = x => new Promise((resolve, reject) => resolve(x % 2 == 0));
```

5. Pre-resolved promises

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

Promise.resolve(anotherPromise).then(console.log))
```

6. Promise states are... (let us focus on how we talk about promises)

```
let executor = function(resolve, reject) {
        // do stuff <--- promise is 'pending' (pendent)

        if (ok) {
                resolve(result) // <--- promise is 'fulfilled' (complerta)
        } else { // error
                reject(result) // <--- promise is 'rejected' (refusada)
        }
}
```

7. Promise fates are... (let us focus on how we talk about promises)
   a) resolved (fate is set; locked in), or
   b) unresolved (fate is uncertain yet).

```
let a = new Promise(...) // let us suppose that this stays pending (unresolved)
let b = Promise.resolve(a) // this is resolved, but not resolved or rejected
```

8. Questions:
   - Com s'accedeix al resultat que representa una `Promise`?
   - Que *retorna* la funció `then`?
   - Perquè `.then(() => Promise.resolve(0))` és el mateix que `.then(() => 0)` ?
   - Que volen dir els conceptes de Fulfilled, Rejected, Pending, Settled, Then-able.
   - Perquè `p.then(f).catch(g)` no és el mateix que `p.catch(g).then(f)`?
   - Que passa quan es fa `p.then(f)` dos cops sobre la mateixa promise?
   - Perquè `p.then(f).then(g)` és molt diferent de `p.then(f); p.then(g)`?