



# JavaScript Exercises (Day-1)

## Node – Getting started

*Important: These are the most important exercises for today, since they provide a lot of very important basic information related to how to use the node-API. I do not cover this in my lectures, since all the information is given in the exercises.*

Hopefully, you have already installed *Node.js*. If not, do it ;-)

Open a command prompt and type: `npm install -g learnyounode`

This will install a small Node module with a series of Node.js workshops and the ability to test and mark your solution as solved.

Create a folder in which to do the following exercises and type: `learnyounode`.

Complete (at least) 1-4 just to get a feeling about how to use node (we will complete more next time)

## The magic of callbacks:

The JavaScript array has a number of cool iteration methods, that all take a callback as a parameter, like `forEach()`, `filter()`, `map()`, `reduce()` and many more.

In the following exercises we are first going to use these basic methods, to recap our knowledge about what they do, and then, we are going to implement the methods by our self, as if they did not already exist.

### 1) What you are expected to know before this semester

Basic JavaScript and an understanding of the DOM

`var`, `let` and `const`

`map`, `filter`, `join` and (perhaps) `reduce`

*Spread* Operator and *destructuring* arrays and objects in JavaScript

React

### 2) Implement user defined functions that take callbacks as an argument

Assume the JavaScript-array did not offer the `filter` method. Then you could implement it by yourself.

a) Implement a function: `myFilter(array, callback)` that takes an array as the first argument, and a callback as the second and returns a **new** (filtered) array according to the code provided in the callback (that is with the same behaviour as the original `filter` method).

Test the method with the same array and callback as in the example with the original `filter` method.

b) Implement a function: `myMap(array, callback)` that, provided an array and a callback, provides the same functionality as calling the existing `map` method on an array.

Test the method with the same array and callback as in the example with the original `map` method.

### 3) Using the Prototype property to add new functionality to existing objects

Every JavaScript function has a *prototype* property (this property is empty by default), and you can attach properties and methods on this **prototype** property. You add methods and properties on an object's prototype property to make those methods and properties available to all instances of that Object. You can even implement (classless) inheritance hierarchies with this property.

The problem with our two user defined functions above (myFilter and myMap) is that they are not really attached to the Array Object. They are just functions, where we have to pass in both the array and the callback<sup>1</sup>.

Create a new version of the two functions (without the array argument) which you should add to the Array prototype property so they can be called on any array as sketched below:

```
var names = ["Lars", "Peter", "Jan", "Bo"];
var newArray = names.myFilter(function(name) {...});
```

## the reduce-method

In most literature (definitely not only JavaScript) you will see map and filter explained together with the reduce function (try to google map filter and reduce), so obviously, this is a method we need to learn.

reduce is used to reduce an array into a single item (a number, string, object, etc). This is a very common problem in all languages. For some specific problems, so common that the array actually has a specific "reduce" function called **join**, which can reduce an array into a string separated by whatever we choose.

```
var all= ["Lars", "Peter", "Jan", "Bo"];
```

a) Use **join** to create a single string from all, with names: *comma-, space. and # - separated*.

Now let's create our own reducer functions (see here for [info](#)).

b) Given this array: `var numbers = [2, 3, 67, 33];`

Create a reducer function that will return the sum (105) of all values in numbers

c) Given this array:

```
let members = [
  {name : "Peter", age: 18},
  {name : "Jan", age: 35},
  {name : "Janne", age: 25},
  {name : "Martin", age: 22},
]
```

Create a reducer function that will return the *average age* of all members.

**Hint:** The reduce callback takes two additional arguments as sketched below:


```
var reducer = function(accumulator, member, index, arr) {
```

*Index* is the current index for which the value (member) are passed in, and *arr* is the array.

*Use this to return different values from your reduce-function, according to whether you have reached the last element or not.*

---

<sup>1</sup> It's a generally accepted design rule that you should **NEVER** add new behaviour to JavaScript's built in objects. We do it here, only to introduce the prototype property

d)  Imagine you were to create a system that could count votes for the presidential election in USA.

Given this array of votes:

```
var votes = [ "Clinton", "Trump", "Clinton", "Clinton", "Trump", "Trump", "Trump", "None"];
```

Create a reduce function that will return a single object like {Clinton: 3, Trump: 4, None: 1 }

**Hints:** You can check whether a property exists in a JavaScript, and add new properties as sketched below:

```
var a = {}  
if (a["clinton"])  
  console.log("I Will Not Print")  
a["clinton"] = 1;  
console.log("You will see me")  
console.log("Value of clinton "+ a["clinton"]);
```

## Hoisting

Team up with another member in the class and implement at least two examples to illustrate that:

- Function declarations are completely hoisted
- var declarations are also hoisted, but not assignments made with them

Explain to each other (as if it was the exam):

- What hoisting is
- A design rule we could follow when using var, now we know about hoisting

## this in JavaScript

Team up with another member in the class. Read about `this` in JavaScript and implement at least three examples (individually) to illustrate how *this* in JavaScript differs from what we know from Java. One of the examples should include an example of explicit setting `this` using either `call()`, `apply()` or `bind()`.

Explain to each other, using the examples (as if it was the exam):

- How `this` in JavaScript differ from `this` in Java
- Why we (because we did not explain about `this`) followed a pattern in our third semester controller and stored a reference to `this` (`var self = this`)
- The purpose of the methods `call()`, `apply()` and `bind()`

## Reusable Modules with Closures

Read: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

1) Implement and test the Closure Counter Example from today's lecture

2) Implement a reusable function using the closure feature, that should encapsulate information about a person (name, and age) and returns an object with the following methods:

- `setAge`
- `setName`
- `getInfo` (should return a string like Peter, 45)

3) Implement an ES6 class with a similar functionality as requested in part 2. Don't use `getXX` or `setXX` but use ES6 properties.

**Continue with learnyounode (make sure to complete 1-6)**