JavaScript Exercises - Period 2 Day-1

Tools for this, and the next period

For this exercise and we suggest you install nodejs and a new editor, to prepare yourself for period 3 (and make this periods JavaScript a bit more fun/realistic)

Install Node (important, install the LTS-version): https://nodejs.org/en/download/

Install a new editor/IDE for JavaScript development. We suggest (and will be using) Visual Studio Code: https://code.visualstudio.com/download

Note: If you feel you already know the following, that is cool ;-). But the ability to use map, filter and reduce, write your own functions that takes callbacks + all the other topics introduced in the following are required JavaScript knowledge for hardcore JS-programmers (and the exam).

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) Using existing functions that takes a callback as an argument

Using the **filter** method:
Declare a JavaScript array and initialize it with some names (Lars, Jan, Peter, Bo, Frederik etc.). Use the filter method to create a new array with only names that contains the letter 'a'.

Using the **map** method:
Use the names-array created above, and, using its `map` method, create a new array with all names reversed.

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

Now, assume the array did not offer these two methods. Then we would have to implement them by our self.

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 (this method should provide 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[1].

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) {…});
```

4) Getting really comfortable with `filter` and `map`

a) Given this array: `var numbers = [1, 3, 5, 10, 11];`

Use map + a sufficient callback to map numbers into this array:

```
var result = [4,8,15,21,11];
```

Hints: The map() callback can take me additional arguments, see here

b) Use `map()` to create to create the <a>'s for a navigation set and eventually a string like below (use `join()` to get the string of <a>'s):

```
<nav>
  <a href="">Lars</a>
  <a href="">Peter</a>
  <a href="">Jan</a>
  <a href="">Bo</a>
</nav>
```

c) Use `map()+(join + ..)` to create to create a string, representing a two column table, for the data given below:

```
var names = [{name:"Lars",phone:"1234567"}, {name: "Peter",phone:
"675843"}, {name: "Jan", phone: "98547"},{name: "Bo", phone: "79345"}];
```

d) Create a single html-file and test the two examples given above.

Hint: add a single div with an id=names, and use DOM-manipulation `(document.getElementById.innerHTML = theString)` to add the `nav` or `table`.

[1] 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) Add a button with a click-handler and use the `filter` method to find only names containing the letter ‘a’. Update the `nav` and the `table` to represent the filtered data.

reduce

In most literature (definitely not only JavaScript) you will see `map` and `filter` explained together with the `reduce` function (try this Google search: https://www.google.dk/search?q=map+filter+reduce&oq=map+filter+reduce&aqs=chrome..69i57j0l5.4472j0j7&sourceid=chrome&ie=UTF-8), 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 specific problems, so common, that the Array actually has a specific "reduce" method 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*.

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

Create a reducer callback that, with reduce(..), will return the sum (105) of all values in `numbers`

c) Given this array:

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

Create a reducer callback that, using the Array's reduce() method, will return the *average age* of all members (25 for the provided array).

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.

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 exist 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

READ: https://www.w3schools.com/js/js_hoisting.asp

Team up with another member of the class. Read about hoisting and implement at least two examples (individually) 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, now we know about hoisting

What is the difference between the keyword **`var`** and the ES6 keyword **`let`**?

this in JavaScript

Read: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

Team up with another member of 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
- The purpose of the methods `call(), apply()` and `bind()`

ES6 classes and Single Page Applications without Netbeans

Getting started

- This exercise assumes you have installed nodejs, and a lightweight JavaScript editor like vs-code
- Clone this project and navigate into the project folder: https://github.com/Cphdat3sem2018f/code_simple_SPA.git
- In this folder type **npm install** to fetch all dependencies (as had it been a Maven project)
- Type **npm run build** (yes before deployment this project has to be built)
- Take a quick look inside the generated **build** folder, and abstract away this folder for the rest of the exercise
- Now open the project in you favourite IDE (with vs code just (in the terminal) type code .)
- Back in the terminal type **npm start**.
- Now arrange your windows so you can see both your editor window (with the code) and the browser with the simple menu.
- Keep your windows arranged like this for the rest of the exercise

Finding individual jokes

In the public folder index.html file, add an input field, a button with the text get joke, and a p-tag to hold the joke you will find. Investigate the start code and implement functionality (in index.js) to find a joke, given it's id.

Adding new Jokes

Still only in the public folders index.html and in index.js, add the neccesary changes to add new jokes to the internal joke-facade.

Reusable Modules with Closures Read:

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

1)

Implement and test the Closure Counter Example from w3schools:

https://www.w3schools.com/js/js_function_closures.asp

2)

Implement a reusable function using the Module pattern 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)