Script for Node.js training on 05.08.2020

This file holds all necessary code snippets and will be used as a reference during today's training. It is written in markdown and can easily be compiled into html or pdf. This can be done On the internet or on your local mashine, using a .md compiler. You might as well use Visual Studio code to get a preview. For that, open this file and press 'CTRL + SHIFT + V'

Introduction

What is Javascript?

From Wikipedia:

> JavaScript (/ˈdʒɑːvəˌskrɪpt/),[6] often abbreviated as JS, is a programming language that conforms to the ECMAScript specification.[7] JavaScript is high-level, often just-in-time compiled, and multi-

> paradigm. [...]. Alongside HTML and CSS, JavaScript is one of the core technologies of the World Wide Web.[8] JavaScript enables interactive web pages and is an essential part of web applications. [...]. As a multi-paradigm language, JavaScript supports event-driven, functional, and imperative programming styles. It has application programming interfaces (APIs) for working with text, dates, regular expressions, standard data structures, and the Document Object Model (DOM). However, the language itself does not include any input/output (I/O), such as networking, storage, or graphics facilities, as the host environment (usually a web browser) provides those APIs. JavaScript engines were originally used only in web browsers, but they are now embedded in some servers, usually via Node.js. They are also embedded in a variety of applications created with frameworks such as Electron and Cordova. Although there are similarities between JavaScript and Java, including language name, syntax, and respective standard libraries, the two languages are distinct and differ greatly in design.

What does it do, in a nutshell:

- Runs in the browser. Can alters what is shown in the window (document).
- Sends data back and forth from formulars (html - forms)
- Does animations, popups, validation, ... Also has access to browser APIs (e.g. localstorage, cookies, sometimes voice and camera)

Why is that useful and helpful to learn?

- It powers almost every webpage and the user's experience when using a browser.
- It can be used to write a variety of applications, with plugins also for mobile devices (Nativescript) and desktop computers (Electron)
- With Node.js, it also runs on the server => Just one language to learn and easy code reusage for frontend and backend

Javascript (JS) Basics

Variables and how to delcare them

Variables store values in memory for later usage. This is usually handled by the (browser's) JS engine.

With new additions to the language, there are several ways to delcare a variable. They can have a global scope (valid within the whole document) or a local scope (only available within a function / object)

```
// The 'vanilla' version can be overwritten and re-declared anytime in the later code.
var myName = "Max";
var myName = "Moritz";  // Works
myName = "Moritz"       // Works

// The 'local' version can be overwritten, but it cannot be redeclared. It's commonly used in functional programming
let myName = "Max";
let myName = "Moritz";  // Throws an error
myName = "Moritz"       // Works

// The 'unchangeable' version is a constant, usually used for configuration variables
```

```
const myName = "Max"
const myName = "Moritz" // Throws an error
myName = "Moritz"       // Throws an error
```

Javascript data types

The most significant data types in JS are Strings, Arrays, Numbers, Objects and Booleans.

- Strings contain characters, numbers and sometimes symbols
- Arrays are collections of different data type objects
- Numbers are ... numbers. They can be classified as Float (decimals, e.g. 5.4) or Integers (full numbers e.g. 5)
- Objects are collections of (related) key - value pairs.
- Booleans are interpreted as either true or false (0, 1)

```
let myName = "Max";                    // String
let students = ["Max", "Moritz"]       // Array
let five = 5                           // Number
let person = {name: "Max", age: 25}    // Object
let istrue = true                      // Boolean
```

Two points that are also noteworthy:

- A variable with a value is interpreted as true, unless that value is false or 'falsely' (read more on that here)

```
let myName = "Max"
if(myName === true) {
  console.log("Variable myName is declared");
}
```

- Each data - type is actually an object (they have methods, other variables can inherit from them and its prototype can be overwritten, details on that later)

```
let myName = new String;
console.log(myName.__proto__)    // Try this in the browser
```

Javascript functions

Like with variables, there are also several ways of declaring functions

```
// The 'vanilla' version
function getMyName(name) {
  console.log(name)
}

// Assign a function like a variable. const - rules apply, the  function cannot be
overwritten
const printMyName = function(name) {
  console.log(name)
}

// Same goes with ES6 Arrow function
const writeMyName = (name) => console.log(name);

// Calling functions always works the same
getMyName("Max");
printMyName("Max");
writeMyName("Max");

// You can also assign functions to objects. It then becomes an object's method
const myself = {
  name: "Max",
  age: "25",
  speak: function(what) {
    console.log("Hello. This is what I have to say: " + what)
  }
}

myself.speak("A method is a object - specific function!");
```

Javascript object methods and 'this'

Methods of objects are scoped, that means they can only be called when referencing to the declaring object. This is particulary useful if you solely want to use the declared object's values. The object's instance can then be acces with the 'this' keyword. It's particulary useful if you work a lot with classes

```
const person = {
  name: "Moritz",
  age: "26",
  introduce: function() {
    console.log("Hello. My name is " + this.name + " and I am " + this.age + "
years old" )
  }
}

person.introduce()
```

In comparison, when trying to use 'this' globally in the browser console, by default it refers to the window object. In node, it refers either to the object created by the node process or the object that is exported in the current module.

Data types in detail

Each JS data type is an object that has its own properties and methods. In the following, some of these methods are shown.

Strings in detail and string methods

Contents of a string can be directly targetted by their index. As with arrays, the index does not start at 1, but at 0. To get the fourth item of a string, you need to get the item with index 3

```
// expected output: "g"
let fourthLetter = "Target the fourth letter"
fourthLetter[3]
```

String methods: Concat two strings

```
// Expected Output: "Max Mueller"
let myName = "Max"
myName.concat(" Mueller");
```

String methods: Split a string at a given character. Result is an array of strings

```
// Split string at all whitespaces
// Expected output: ["Max", "Mueller"]
let myFullNameArray = myFullName.split(" ")

// Split a URL or a path into its params
// Expected output: ['https:', '', 'devdocs.io', 'javascript', 'global_objects',
'string', 'split']
let myUrl = "https://devdocs.io/javascript/global_objects/string/split"
myUrlFragments = myUrl.split("/");
```

String methods: Slice a string from a given index to another

```
let myFox = 'The quick brown fox jumps over the lazy dog.';
myFox.slice(10, 19) // Expected output: brown fox
```

String methods: Replace letters

```
// Replace the x in myName with rc and change my name to Marc
// Expected output: "Marc"
myName.replace("x", "rc")
```

Replace also works with regular expressions, allowing you to for instance change all occurences of a letter in a string or work with more complex requirements.

```
// Replace all b - letters in the string with a's
// Expected output: aaaa
let regex = /b/gi;
let randomString = "aabb"
randomString.replace(regex, "a");
```

String methods: Get the length of a string

```
// Expected output: 4
myName.length()
```

String methods: Get the index of a certain character

> Note: This method is case sensitive. Also, for more complex queries, it's better to use a regular expression

```
// Expected output: 0
myName.indexOf("M")

// This method can also be used to check if a character exists within a string.
// If it doesn't, the method will return a value of -1
myName.indexOf("z");

// This comes in handy for some cases, e.g. for simple checks if a string contains
certain characters.
// Expected output: "String contains no z"
myName.indexOf("z") === -1 ? "String contains no z" : "String contains a z";
```

String methods: To upper and to lower case

> Note: These two methods come in handy for form handling, e.g. if you don't know whether to expect an upper case or a lower case user input.

```
let mySearchword = "How do I change ALL letters HERE lowercase or UPPERCASE?";
mySearchword.toLowerCase();
mySearchword.toUpperCase();
```

Read more about strings and their methods on devdocs

Arrays in detail and array methods

As with strings, content of an array can be targetted by its index

```
let myGoons = ["Max", "Moritz", "Ansgar", "Bertha"];
myGoons[0] // result: "Max"
```

Array propertiy (not method!): length

```
// Not to be confused with the string method length(), an array has not a methods,
but a property that defined its length
// Expected output: 4
myGoons.length
```

Array methods similiar to strings

```
// Get the index of an index entry
// Expected output: 2
myGoons.indexOf("Ansgar")

// High order function with the same result
myGoons.findIndex(goon => goon === "Ansgar")

// Concat two arrays ( returns a new array, leaves the original unchanged )
// Expected output: ['Max', 'Moritz', 'Ansgar', 'Bertha', 'Harriet',
'Robert','Juliet']
let myWatchmen = ["Harriet", "Robert", "Juliet"];
myGoons.concat(myWatchmen)

// ES6 syntax with the same result ( REST parameter )
// For comparison, try doing it without the ... before each array
[...myGoons, ...myWatchmen]
```

Array methods: push + pop / shift + unshift

The mentioned methods are the primary ones to add and remove elements from an array

- push adds an element to the end of an array and returns the new array's length
- pop removes an element from the end of an array and returns it
- unshift adds an element from the beginning of an array and returns the new array's length
- shift removes an element from the beginning of an array and returns it

```
// Add a new goon to the array
// Expected result: [ 'Max', 'Moritz', 'Ansgar', 'Bertha', 'Anna' ]
myGoons.push("Anna");
```

```
// Remove Anna again
myGoons.pop();

// Now add Anna to the beginning of the array
// Expected result: [ 'Anna', 'Max', 'Moritz', 'Ansgar', 'Bertha' ]
myGoons.unshift("Anna");

// Remove Anna again
myGoons.shift();
```

Array methods to work with its content

Traditional approaches to work with arrays were usually encapsulating it into a for ... in or simple for - function. High order functions make work with arrays easier on that matter, especially when writing simple functions.

```
// The usual approach with a for - loop goes like this
// Expected result: "Max" , "Moritz", "Ansgar", "Bertha"
for(let i = 0; i < myGoons.length; i ++) {
  console.log(myGoons[i])
}

// High order function with the same result
// forEach executes a callback function FOR each goon in myGoons ( each element in the array )
myGoons.forEach(goon => console.log(goon))

// Another High order function with the same result, but different functionality
// map  executes a callback function ON each goon in myGoons AND returns that element ( each element in the array )
myGoons.map(goon => console.log(goon))
```

Usually, array items are more complex than simple strings. Let's give our goons more complexity and worth with the array, then.

```
let myGoonsDetail =
[
  { name: "Max", age: 12,place: "Amsterdam" },
  { name: "Moritz", age: 14, place: "Rotterdam" },
  { name: "Ansgar", age: 11, place: "Amsterdam" },
  { name: "Bertha", age: 10, place: "Muenster" }
]
```

A common task is finding or filtering items. This works the traditional way, we will focus on High order function, however.

```
// Filter out all goons that are 11 years old or younger
// Expected result:
// [
//   { name: 'Max', age: 12, place: 'Amsterdam' },
//   { name: 'Moritz', age: 14, place: 'Rotterdam' }
// ]
myGoonsDetail.filter(goon => goon.age > 11)

// Find all goons that are living in Amsterdam
myGoonsDetail.filter(goon => goon.place === "Amsterdam");

// Or find the first goon that's living in Amsterdam using its index.
myGoonsDetail.filter(goon => goon.place === "Amsterdam")[0];

// Also works with queries. Now let's find all goonies with a capital M in their
name
myGoonsDetail.filter(goon => goon.name.indexOf("M") !== -1)

// Or maybe find all goons that are not from a place that contains a "dam" in it
myGoonsDetail.filter(goon => goon.place.indexOf("dam") !== -1)
```

Now imagine you want to reduce the complexity in the array back to only the goon's names. For that, you can use .map() again. As already stated, it executes a function on each element, returns the results as an array AND changes the original array accordingly. The first function here is to simply return the name - property of each element in the myGoonsDetail - array, while the second will, on purpose, mess up the array by overwriting all names with "Not found".

```
// Expected result: [ 'Max', 'Moritz', 'Ansgar', 'Bertha']
myGoonsDetail.map(goon => goon.name);

// Expected result: [ 'Not found', 'Not found', 'Not found', 'Not found' ]
// Also, the original myGoonsDetail will change. Be cautious when to use it
myGoonsDetail.map(goon => goon.name = "Not found");
```

Array methods: some

Instead of actually returning an element, there might be a case in which you simply want to find out if an element is present in an array, or whether an element within an array has a certain propery. That works with the .some() methods

```
// Find out if there is a goon with the age of 12 in the myGoonsDetail array
// Expected result: true
myGoonsDetail.some(goon => goon.age === 12);

// Find if there is a goon that's from Hamburg.
// Expected result: false
myGoonsDetail.some(goon => goon.place === "Hamburg")
```

Array methods: sort

Sorts the content of an array either from small to large for numbers, or A - Z for characters

```
// Expected result: ['Ansgar', 'Bertha', 'Max', 'Moritz']
myGoons.sort()
```

Sorting also works in more complex arrays when passing in a comparison function

```
// Source: https://stackoverflow.com/questions/1129216/sort-array-of-objects-by-string-property-value#1129270
myGoonsDetail.sort((a,b) => (a.age > b.age) ? 1 : ((b.age > a.age) ? -1 : 0));
```

Read more about arrays and their methods on devdocs

Numbers, objects and booleans reference

Objects might be covered at a later point, numbers and booleans are rather straightforward.

Objects Numbers Booleans

Asynchronous programming and Promises

You can find a brief introduction in the general concept on MDN here

For a synchronous program, whenever you execute a function, it blocks the thread it's running on till a result or an error is returned. Since Javascript is single threaded, any interaction would be blocked whenever any code execution on the client happens. That means: While data is being fetched from a database and the server is busy with the query, the webapp cannot be used.

Promises in browsers and the server

To resolve this, modern browsers interpret code asynchronously. In a nutshell, async programming is event driven and non - blocking during code execution, meaning while the data is being returned, the webapp can execute different functions.

This feature comes with a downside the following code will show.

```
// Expected result: "My server's data"
let serverData;

setTimeout(() => {
  serverData = "My server's data";
}, 1000);
```

```
// Will not work like that
console.log(serverData);
```

The log says the variable is undefined as console.log did not wait for the variable's value to be reassigned. That's a problem, because it means JS will, for instance, not wait for the server to respond with data and instead just continue running the nexxt code lines. That's where promises come into play.

```
// Expected result: "My server's data"
let serverData;

const getServerData = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      serverData = "My server's data";
      // Here, we pass on the string right after it has been reassigned
      resolve(serverData);
    }, 1000);
  });
};

// serverData then becomes available for other functions, as a local variable, to
use within its callback
getServerData().then((serverDataResponse) => console.log(serverDataResponse));

// This will still be undefined
console.log(serverData);
```

Now the first time serverData is logged, it will still be undefined. After one second, however, the reassigned value is correctly logged.

Let's now do this with a real world example. fetch() is a modern and lightweight, promise based, function to get data from a remote server. It follows the same logic as shown above

> Note: fetch() is not available within node, it's a browser feature.

```
fetch("https://jsonplaceholder.typicode.com/posts/1").then(response =>
response.json()).then(data => console.log(data))
```

If you input this to your browser console, you will notice a brief time interval passing before the result is logged. This is the equivalent to the second the above setTimeout() function simulated.

Declaring async functions

As you saw, the variables returned by fetch and the promise are only available on the local scope by default. A workaround are async - await functions and / or assigning the resolved value to a global variable. Again, this works only in the browser with fetch.

> Note: Async - await is syntactic sugar for promises. It's nothing new, really, just a convenient way to keep sync - function structure with async code execution.

```
// Expected result: object { userId, id, title, body }
const getServerDataAsync = async () => {
  let response = await fetch("https://jsonplaceholder.typicode.com/posts/1")
  data = await response.json();
  console.log(data)
  return data;
}

// The returned data will only be available within the promise's callback or
within another async function.
const data = getServerDataAsync();
data.then(data => console.log(data))

// You can, however, assign the server's data to a global variable within the
callback
let globalData;
data.then(data => globalData = data);
```

Additional ressources for repetition / getting started

- W3Schools Tutorials and reference
- Freecodecamp tutorials
- Traversy media JS Crash course
- Full 3 - hour Freecodecamp JS Course
- Blogpost on high order functions
- ES6 Syntax