

## Period-1 Vanilla JavaScript, Es-next, Node.js, Babel + Webpack and TypeScript

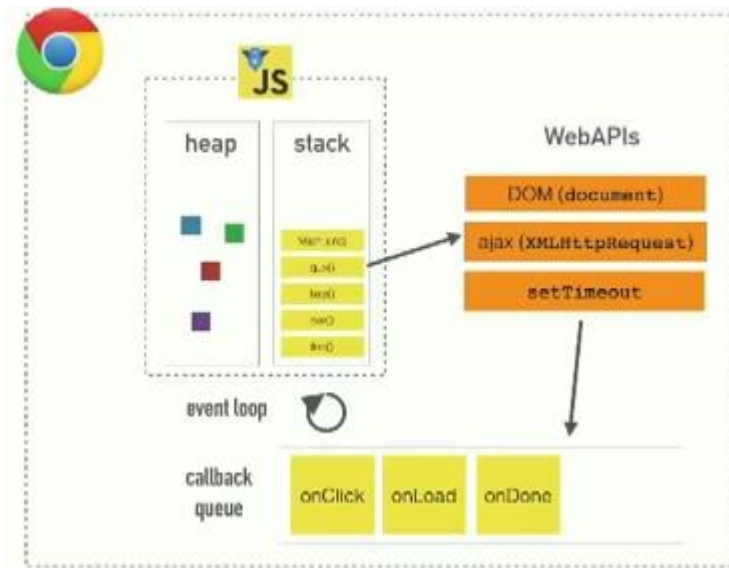


Note: This description is too big for a single exam-question. It will be divided up into several smaller questions for the exam

Explain and Reflect:

- Explain the differences between Java and JavaScript + node. Topics you could include:
  - That Java is a compiled language and JavaScript a scripted language.
  - Java is Object oriented and JS is functional.
  - Java is suited for complex web based concurrency projects, and Node.js is best suited for small size projects.
  - Java dominates server-side interaction while Node.js can be utilized on client and server side.
  - Java errors when compiling, JS errors show in runtime.
  - Java is both a language and a platform.
  - General differences in language features.
  - Blocking vs. non-blocking
    - **Blocking** refers to operations that block further execution until that operation finishes while **non-blocking** refers to code that doesn't block execution. Or as Node.js docs puts it, blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. Blocking methods execute **synchronously** while non-blocking methods execute **asynchronously**.
    - **Synchronous(sync)** – Blocking code. The execution usually refers to code executing in sequence. In sync programming, the program is executed line by line, one line at a time. Each time a function is called, the program execution waits until that function returns before continuing to the next line of code.
    - **Asynchronous(async)** – Non-blocking. The execution refers to execution that doesn't run in the sequence it appears in the code. In async programming the program doesn't wait for the task to complete and can move on to the next task.
- Explain generally about node.js, when it “makes sense” and *npm*, and how it “fits” into the node ecosystem.
  - Node.js is an open source project designed to help you write JavaScript programs that talk to **networks, file systems or other I/O** (input/output, reading/writing) sources.
  - Node package manager(npm) is a **package manager for Node.js**. It is one, if not the largest online repository where we can install from. Fx. `fetch()` isn't in Node as a default, therefore we need to “npm install node-fetch”. This means that npm makes it a lot easier to use node, and gives us a lot of functionality, like we know it from the browser.

- Explain about the Event Loop in JavaScript, including terms like; blocking, non-blocking, event loop, callback queue and "other" API's. Make sure to include why this is relevant for us as developers.





- Node.js is a single-threaded application, but it can support concurrency via the concept of event and callbacks.
- The above picture is basically the same for us in Node. The only difference is instead of; "WebAPIs" it is "**C++APIs**".
- In the first example given below, `console.log` will be called before `moreWork()`. In the second example `fs.readFile()` is non-blocking so JavaScript execution can continue and `moreWork()` will be called before the `console.log`.

```
// Blocking
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log



// Non-blocking
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

- In the picture above, once the file has been read it will put the data on hold in the **callback queue**. The **event loop** waits for and dispatches events when it finds some from the **callback queue**. However the **event loop** first sends the data back onto the **stack** when there isn't anything that **obstructs the stack**. Therefore in the picture above the `console.log` with the data from the file will be thrown back onto the **stack** after the `moreWork()` has given an opportunity to let the event loop do its magic.
- In Node, **non-blocking** primarily refers to I/O operations(**C++APIs**). While **Blocking** refers to the JS stack doing the work.


- What does it mean if a method in nodes API's ends with xxxxxx**Sync**?
  - When a node API ends with Sync, it means that **it is blocking**. It will run the code synchronous with everything else that is on the stack. See the picture from before with the blocking example.
-  Explain the terms JavaScript Engine (name at least one) and JavaScript Runtime Environment (name at least two)
  - A JavaScript engine is a computer program that executes JavaScript code.
  - “**V8**” is the name of an open-source JavaScript engine, developed by **the chromium project**, for **Google Chrome(WebBrowser)**. And all other chromium based web browsers.
  - **Google chrome and Node.js** both run on chromiums V8 engine and are both JavaScript Runtime Environments.
  - Mozilla firefox uses; “**SpiderMonkey**”, Safari uses; “**Nitro**” as their respective JS engines.
- Explain (some) of the purposes with the tools *Babel* and *WebPack* and how they differ from each other.  Use examples from the exercises.
  - **Webpack** – Webpack is a module bundler for modern JavaScript applications. When webpack processes our application, it internally builds a dependency graph witch maps every module our project needs and generates one or more bundles.(Dependency graph – any time a file depends on another, webpack treats this as a dependency. <-Imports and exports)  
Code example; “**Period1\ (18-09-2020) - ES-next, Babel, Typescript and WebPack\webpack-demo-Development**” <- npm install. npm start
  - **Babel** - Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.

Explain using sufficient code examples the following features in JavaScript (and node)

- Variable/function-Hoisting
  - A strict definition of **hoisting** suggests that **variable and function declarations** are physically moved to the top of your code, but this is not in fact what happens. Instead, the variable and function declarations are **put into memory during the compile phase**, but stay exactly where you typed them in your code.  
Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript**” <- 4
- *this* in JavaScript and how it differs from what we know from Java/.net.
  - In JavaScript this always refers to the “owner” of the function we're executing, or rather, to the object that a function is a method of.
  - In Java, this refers to the current instance object on which the method is executed.  
Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript**” <- 5
- Function Closures and the JavaScript Module Pattern
  - **Module pattern** is used to define objects and specify the variables and the functions that **can be accessed from outside the scope of the function**.  
Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript**” <- 6 & 6.1
  - Closures are the only way to make private methods in JS.
  - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>  
Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript**” <- 6, 6.1, 6.2 & 6.3.
- User-defined Callback Functions (writing functions that take a callback)
  - Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript**” <- 2&3
- Explain the methods `map`, `filter` and `reduce`
  - The **map()** method creates a new array populated with the results of calling a provided function on every element in the calling array.
  - The **filter()** method creates a new array with all elements that pass the test implemented by the provided function.
  - The **reduce()** method executes a reducer function (that you provide) on each element of the array, resulting in single output value.
  - Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript**” <- What\_We\_Are\_Expected\_To\_Know.js
- Provide examples of user-defined reusable modules implemented in Node.js (learnyounode - 6)
  - Code example; “**Period1\28-08-2020 - Introduction to the Full Stack JavaScript\learnyounode**” <- make-it-modular.js & mymodule.js

- Provide examples and explain the es2015 features: let, arrow functions, this, rest parameters, destructuring objects and arrays,  maps/sets etc.
  - The **let** statement declares a block-scoped local variable, optionally initializing it to a value.
  - An **arrow function** expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.
    - Does not have its own bindings to this or super, and should not be used as methods.
    - Can not be used as constructors.
  - In JavaScript **this** always refers to the “owner” of the function we're executing, or rather, to the object that a function is a method of.
  - The **rest parameter** syntax allows us to represent an indefinite number of arguments as an array.  
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest\\_parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)
  - The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.  
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)
  - Code example; “**Period1\28-08-2020**) - Introduction to the Full Stack JavaScript” <- Most js files and specific - Rest\_Destructuring\_Demo.js
-  Provide an example of ES6 inheritance and reflect over the differences between Inheritance in Java and in ES6.
  - **Skal finde et godt eksempel**
- Explain and demonstrate, how to implement event-based code, how to emit events and how to listen for such events
  - The EventEmitter is a module that facilitates communication between objects in Node. EventEmitter is at the core of Node asynchronous event-driven architecture. Many of Node's built-in modules inherit from EventEmitter.
  - Emitter objects emit named events that cause previously registered listeners to be called. So, an emitter object basically has two main features:
    - Emitting name events.
    - Registering and unregistering listener functions.
  - Code example; “**(02-09-2020)** - Getting started with node-js” <- Dos detector mini server

## ES6,7,8,ES-next and TypeScript


- Provide examples with es-next, running in a browser, using Babel and Webpack
  - Code example; “**Period1\18-09-2020 - ES-next, Babel, Typescript and WebPack\usingNodeWithBabel**”
- Explain the two strategies for improving JavaScript: Babel and ES6 + ES-Next, versus Typescript. What does it require to use these technologies: In our backend with Node and in (many different) Browsers.
  - Babel doesn't care about TypeScript types.  
const myCoolString : string = 9; <- This code would compile without any errors or warnings in babel, but not in type script.
  - TypeScript helps us in a way where we can transpile our TS code into JS code. We can transpile it into all versions from ES3 to ES-Next.
  - Skal finde bedre / mere præcise definitioner
- Provide **examples** to demonstrate the benefits of using TypeScript, including, types, interfaces, classes and generics
  - Code example; “**Period1\25-09-2020 - An introduction to TypeScript\TypeScriptExercise**”
-  Explain the ECMAScript Proposal Process for how new features are added to the language (the TC39 Process)
  - ECMA TC39, the committee which handles making ECMAScript better and easy to use. The TC39 process has 5 stages to get the specifications live.
  - Stage 0: Strawperson  
Allow input into the specification
  - Stage 1: Proposal  
Make the case for the addition, describe the solution and identify the potential challenges
  - Stage 2: Draft  
Precisely describe the syntax and semantics using formal spec language
  - Stage 3: Candidate  
Indicate that further refinement will require feedback from implementations and users
  - Stage 4: Finished  
States that the addition is ready for inclusion in the formal ECMAScript standard

## Callbacks, Promises and async/await

Explain about (ES-6) promises in JavaScript including, the problems they solve, a quick explanation of the Promise API and:

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.


This resolves the problem of having to run all the code line by line. We can make promises and continue our code until the promise is finished. We “**don’t block**” our code.

-  Example(s) that demonstrate how to avoid the callback hell (“Pyramid of Doom”)
  - Code example; “**Period1\11-09-2020 - Promises and asyncawait\exercises**” <- exercise1(pyramidOfDoom), ex1-crypto-module.js & ex1-crypto-module-async-await.js.
- Example(s) that demonstrate how to execute asynchronous (promise-based) code in **serial** or **parallel**
  - Code example; “**Period1\11-09-2020 - Promises and asyncawait\exercises**” <- exercise3
- Example(s) that demonstrate how to implement **our own** promise-solutions.
  - **Period1\11-09-2020 - Promises and asyncawait\exercises**” <- exercise1, ex1-crypto-module.js & ex1-crypto-module-async-await.js.
  - **Skal finde / lave en bedre version af vores eget promise**
- Example(s) that demonstrate error handling with promises
  - **Period1\11-09-2020 - Promises and asyncawait\exercises**” <- exercise1, ex1-crypto-module.js & ex1-crypto-module-async-await.js.
  - **Skal finde / lave en bedre version af error handling**

Explain about JavaScripts **async/await**, how it relates to promises and reasons to use it compared to the plain promise API.

- One of the big reasons as to why we should use **async/await** over normal promises is because it makes the code easier to read. There isn’t a lot of .then().then().then(), but more of a await.
- Async/Await does the same as normal promises. It is just easier to read for us developers.

Provide examples to demonstrate

- Why this often is the preferred way of handling promises
  - Code example; “**Period1\11-09-2020 - Promises and asyncawait\exercises**” <- exercise2.js
- Error handling with async/await
  - **Skal finde / lave en bedre version af error handling**
-  Serial or parallel execution with async/await.
  - Code example; “**Period1\11-09-2020 - Promises and asyncawait\exercises**” <- exercise3.js

Se the exercises for Period-1 to get inspiration for relevant code examples