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

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

**Husk at læse dine noter fra kode opgaverne!**

**Explain and Reflect:**

**1: 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 both a language and a platform
  + General differences in language features.
  + Blocking vs. non-blocking

**Answer:**

(Remember to talk about asynchronous and synchronous – non-blocking and blocking)

**1.**[**JavaScript**](https://www.geeksforgeeks.org/javascript-tutorial/)**:**   
JavaScript is dynamic, meaning it has no type system. JavaScript is a lightweight programming language (“scripting language”) and used to make web pages interactive. It can insert dynamic text into HTML. JavaScript is also known as browser’s language. JavaScript (JS) is not similar or related to Java. Both the languages have a C like a syntax and are widely used in client-side and server-side Web applications, but there are few similarities only.

**2.**[**Java**](https://www.geeksforgeeks.org/java/)**:**   
Java is an object-oriented programming language and have virtual machine platform that allows you to create compiled programs that run on nearly every platform. Java promised, “Write Once, Run Anywhere”.

**Difference between Java and JavaScript :** 

| Java | JavaScript |
| --- | --- |
| Java is strongly typed language and variable must be declared first to use in program. In Java the type of a variable is checked at compile-time. | JavaScript is weakly typed language and have more relaxed syntax and rules. |
| Java is an object oriented programming language. | JavaScript is an object based scripting language. |
| Java applications can run in any virtual machine(JVM) or browser. | JavaScript code used to run only in browser, but now it can run on server via Node.js. |
| Objects of Java are class based even we can’t make any program in java without creating a class. | JavaScript Objects are prototype based. |
| Java program has file extension “.Java” and translates source code into bytecodes which is executed by JVM(Java Virtual Machine). | JavaScript file has file extension “.js” and it is interpreted but not compiled, every browser has the JavaScript interpreter to execute JS code. |
| Java is a Standalone language. | contained within a web page and integrates with its HTML content. |
| Java program uses more memory. | JavaScript requires less memory therefore it is used in web pages. |
| Java has a thread based approach to concurrency. | Javascript has event based approach to concurrency. |
| Java supports multithreading. | JavaScript doesn’t support multi-threading. |

**2: Explain** generally about node.js, when it “makes sense” and *npm*, and how it “fits” into the node echo system.

**Answer:**

Node is a runtime environment for executing JavaScript code outside of a browser. We often use node to build backend services also called API’s (application programming interfaces). Node is good for building highly-scalable, data-intensive and real-time apps. As we use JavaScript in the frontend and backend (fullstack) we can write cleaner and more consistent code.

**NPM** has the larges ecosystem of open-source libraries. This helps us with using those libraries so we don’t have to build everything ourselves and can focus on the core of our application. Npm is also used to handle the dependencies of the project. (project init – also remember .gitignore to “node\_modules”).

***Node*** allows developers to write JavaScript code that runs directly in a computer process itself instead of in a browser. ***Node*** can, therefore, be used to write server-side applications with access to the operating system, file system, and everything else required to build fully-functional applications.

**Node.js uses asynchronous programming!**

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

What Can Node.js Do?

* Node.js can generate dynamic page content
* Node.js can create, open, read, write, delete, and close files on the server
* Node.js can collect form data
* Node.js can add, delete, modify data in your database

What is a Node.js File?

* Node.js files contain tasks that will be executed on certain events
* A typical event is someone trying to access a port on the server
* Node.js files must be initiated on the server before having any effect
* Node.js files have extension ".js"

**3: 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.

**Answer:**

**https://olinations.medium.com/the-javascript-runtime-environment-d58fa2e60dd0**

*(Other API’s setTimeout(); setInterval(); fetch(); HTTP request)*

Let’s start with picturing how JavaScript works. First of all, there is a heap and a stack. The heap is for what and where things are saved in the memory. The stack is where our action is put as the interpreter reads the code. Now lets say we have something that takes some time to do, like doing a setTimeout(); or fetch(); if we have to wait for this to be finished before we move on with the code things will get slow and the application you are building will not run probably. So, this is where asynchronous code comes into the picture. setTimeout(); runs in the web API. What that means is that when the interpreter hits this function it will be moved from the stack to the web API. Here it will be executed and afterwards be moved to the callback queue. From here there is an event loop which is looking at the stack and the callback queue. When the stack is empty (main(); has been removed) then the event loop will move one item from the callback queue to the stack. When that has executed it will again look at the callback queue and move one thing up if the stack is clear and so on. Worth mentioning is that there is also a render queue which has higher priority than the callback queue as this is what renders the web page (every 16 frame as I recall it?).

By having this setup in JavaScript we can run code asynchronis(non-blocking) which is very handy for us since JavaScript also only runs on a single thread. If we didn’t have this setup the code would run synchronous and would be blocking.

**4: Explain:** What does it mean if a method in nodes API's ends with xxxxxx**Sync**?

**Answer:**

When a method ends with Sync it means that it is synchronous and therefor is blocking. We should refrain from using it as told from my teacher Lars.

**Synchronous**

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

**Asynchronous**

const fs = require("fs");

fs.readFile("/file.md", (err, data) => {

if (err) throw err;

console.log(data);

});

moreWork(); // will run before console.log

**5:**      **Explain** the terms JavaScript Engine (name at least one) and JavaScript Runtime Environment (name at least two)

**Answer:** There are three engines, as far as I know, Edge = Chakra, Firefox = Spider monkey and Chrome = V8. The engines make it possible for us to be able to run JavaScript code in the browsers. So, in general the engines parse JavaScript code. A JavaScript Runtime Environment can be found in node, which is built upon Chromes V8 engine, and therefor we can do full stack development with JavaScript. Another Runtime Environment can be found in the browsers. The Runtime Environment has API’s which a developer can access to build a program.

*Window -> DOM is from the web api/runtime environment.*

*Global -> process is from nodes runtime environment.*

*AJAX, the DOM tree, and other API’s, are not part of JavaScript, they are just objects with properties and methods, provided by the browser and made available in the browser’s JS Runtime Environment.*

**6: Explain:** (some) of the purposes with the tools *Babel* and *WebPack and how they differ from each other*.       Use examples from the exercises.

**Answer:**

**- Babel:**

    - JavaScript compiler that converts ECMAScript 2015+ into a backwards compatible version of JavaScript in current and older browsers or environments.

    - It does the following:

      - Transform syntax

      - Polyfill features that are missing in your target environment (through @babel/polyfill)

      - Source code transformations (codemods)

To make a babel file you write .babelrc or babel.config.json

\*\*Example on how it would look like:\*\*

 {

  "presets": ["@babel/preset-env", "@babel/preset-react"]

}

\*remember that babel transpile code and not compile\*

[@babel/preset-env](https://babeljs.io/docs/en/babel-preset-env) is a smart preset that allows you to use the latest JavaScript without needing to micromanage which syntax transforms (and optionally, browser polyfills) are needed by your target environment(s).

**Code example:** flow 1 -> day 4 -> webpacktutorial -> babel.config.json

**- Webpack:**

-  is a module bundler that bundles your code into 1 or 2 files that you define. It also compiles TypeScript to JavaScript. You need a webpack config file to handle your configurations.

A webpack config file is written like: webpack.config.js. This file contains an entry point (where to start), an output (where to end and what to create). In between there are modules (rules for modules) and plugins (what to do with the overall code, the bundle).

1) the entry point: Where to start ( relative path ('./') )

2) the output: Where to end and what to create

var path = require('path')

path: path.resolve(\_\_dirname, 'dist'),

filename: 'bundle.js',

publicPath: '/dist' <-- absolute path

--------------------------------------------------------

IN BETWEEN DURING THIS JOURNEY

--------------------------------------------------------

3) modules: what to do with them and module rules - what should you do with each individual file you load

4) plugins: right before we create the bundle - what should you do with the prepared code, with the overall code

like should you minify it ( var webpack = require('webpack') -> new webpack.optimize.UglifyJsPlugin({}) )

or do something else like plugins to outsource your css code into seperate stylesheets

**Code example:** flow 1 -> day 4 -> webpacktutorial -> webpack.config.js

***(see notes under day 4 folder for more details*)**

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

* **Variable/function-Hoisting**

**Answer:**

See flow 1 -> day 1 examples with hoisting (*remember to read the notes)*

* ***this* in JavaScript and how it differs from what we know from Java/.net.**

**Answer:**

See flow 1 -> day 1 examples with this *(remember to read the notes)*

* **Function Closures and the JavaScript Module Pattern**

**Answer:**

See flow 1 -> day 1 examples with closures *(remember to read the notes)*

**Extra answer not required: Example on JavaScript Module Pattern see example two in the closure file.**

Generally, it is a design pattern which is used to wrap a set of variables and function together in a single scope. It is used to define objects and specify the variables and the functions that can be accessed from outside the scope of the function. We expose certain properties and function as public and can also restrict the scope of properties and functions within the object itself, making them private. This means that those variables cannot be accessed outside the scope of the function. We can achieve data hiding an abstraction using this pattern in the JavaScript.

*(Tænk private access modifier fra java)*

Use Module pattern for the following benefits:

1. **Maintainability:**Module Patterns enable better maintainability since all the related code can be encapsulated inside a single logical block. These logically independent blocks are relatively easier to update.
2. **Reusability**: We single unit of code can be reused across the entire application. Functionality enclosed as a module can be reused and we do not need to define the same functions at multiple points.

https://www.youtube.com/watch?v=cRHQNNcYf6s&ab\_channel=WebDevSimplified:

To export a module you write export default function/class name and to import you write import function/class name from ‘path/to/the/file’.

In node you write Module.export = function/class name and to import you write const variable\_name = require(‘path/to/the/file’)

* **User-defined Callback Functions (writing functions that take a callback)**

**Answer:**

See flow 1 -> day 1 examples with callbacks. Talk about the first exercise and show how I pass one function to another and use that function in the other function. *(remember to read notes)*

* **Explain the methods map, filter and reduce**

**Answer:**

See flow 1 -> day 1 examples with callbacks. *(remember to read notes)*

Here I will find examples on map (loops), filter (loops by Boolean) and reduce (several elements to one element)

* Provide examples of user-defined reusable modules implemented in Node.js (learnynode - 6)

**Answer:**

See flow 1 -> day 1 examples from learnyounode 1-6 (hello-world, my-first-io, my-first-async-io**, make-it-modular** and **mymodule**)

Basically remember to import const fs = require(‘fs’) and to export Module.exports = (function)

* **Provide examples and explain the es2015 features: let, arrow functions, this, rest parameters, destructuring objects and arrays,   maps/sets etc.**

**Answer:**

See flow 1 -> day 1 examples with let\_arrow\_rest\_destructuring. *(remember to read notes)*

See flow 1 -> day 1 examples with this. *(remember to read notes)*

***Note that there is also this example with arrow function in the*** let\_arrow\_rest\_destructuring.js file.

**Example on set/maps:**

 \*\*sets\*\*

  A collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur once; it is unique in the Sets collection.

  ```

  // To initialize a set, we can pass an array of values to

  // the Set constructor, this will create a Set with those values:

  const confectioneries = new Set(['oreo', 'marshmallow','oreo', 'kitkat', 'gingerbread']);

  console.log(confectioneries); // result: Set { 'oreo', 'marshmallow', 'kitkat', 'gingerbread' }

  // In the snippet above, the duplicate value “oreo” is quietly removed from the Set and only unique values are returned.

  ```

  \*\*maps\*\*

  The Map object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values) may be used as either a key or a value. A Map object iterates its elements in insertion order — a for...of loop returns an array of [key, value] for each iteration.

  ```

  // create a Map using the Map constructor

  const users = new Map();

  console.log(users); // Map {}

  ```

  Key-value pairs are added to a Map using the set() method. This method takes in two arguments, the first being the key and the second, the value, which is referenced by the key

  ```

  // Adding items

  users.set('John Doe', {

  email: 'johndoe@example.com',

  });

  users.set('Jane Doe', {

email: 'janedoe@example.com',

  });

  console.log(users);

  /\_\_ console.log result

  Map {

'John Doe' => { email: 'johndoe@example.com'},

'Jane Doe' => { email: 'janedoe@example.com'} }

  \_\_/

  ```

<br>

* Provide an example of ES6 inheritance and reflect over the differences between Inheritance in Java and in ES6.

**Answer:**

See flow 1 -> day 5 examples from TypeScript exercises *(remember to read notes)*

Class foo {}

Typeof Foo // ‘Function’

Inheritance is when you design your types around what they are.

<https://www.youtube.com/watch?v=wfMtDGfHWpA&ab_channel=FunFunFunction>

**Difference:**

https://www.programcreek.com/2014/02/java-vs-javascript/

In Java, there are classes, and instances, as separate concepts. In order to do inheritance, you have to use the base class to create a new class, which can then be used to produce derived instances.

Like Java, JavaScript is also an object oriented language, but JavaScript does not use classes.  
In JavaScript you don't define classes and create objects from these classes. JavaScript is prototype based, not class based. To do inheritance, you can use any object instance as a prototype. The following example gives a taste of JavaScript Inheritance.

**Code ex:**

<script>

*// define the Person Class*

**function** Person() {}

Person.**prototype**.sayHello = **function**(){

alert ('Hello!');

};

*// define the Student class*

**function** Student() {}

*// inherit Person*

Student.**prototype** = **new** Person();

*// replace the sayHello method*

Student.**prototype**.sayHello = **function**(){

alert('Hello, I am a student!');

};

*// add a new method*

Student.**prototype**.sayGoodbye = **function**(){

alert('Goodbye!');

};

**var** student = **new** Student();

student.sayHello();

student.sayGoodbye();

*// check inheritance*

alert(student **instanceof** Person); *// true*

alert(student **instanceof** Student); *// true*

</script>

* Explain and demonstrate, how to implement event-based code, how to emit events and how to listen for such events.

**Answer:**

See flow 1 -> day 2 and especially my-first-node-app for emit and listening on those events *(remember to read the notes)*

https://www.youtube.com/watch?v=TlB\_eWDSMt4&ab\_channel=ProgrammingwithMosh

ES6,7,8,ES-next and TypeScript

* Provide examples with es-next, running in a browser, using Babel and Webpack

**Answer:**

* 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

**Answer:**

* Provide **examples** to demonstrate the benefits of using TypeScript, including, types, interfaces, classes and generics

**Answer:**

See flow 1 -> day 5 and use examples from Intro\_exercise(does not have generics) and even better from TypeScript\_exercises which contains all of the above *(remember to read the notes)*

* Explain how we can get typescript code completion for external imports.

**Answer:**

Third Party Declaration Files

When using a library that was originally designed for regular JavaScript, we need to apply a declaration file to make that library compatible with TypeScript (Get all the cool intellisense in our editor). A declaration file has the extension ***.d.ts*** and contains various information about the library and its API.

Include node-fetch in a TypeScript Project

npm install --save **@types/**node-fetch

* Explain the ECMAScript Proposal Process for how new features are added to the language (the TC39 Process)

**Answer:**

[TC39](https://tc39.es/) or the 39nth Technical Committee by its full name, is a group under [ECMA International](https://www.ecma-international.org/) containing ECMA members - which are JavaScript developers, implementers, academics and more; from various companies and institutions. The committee collaborates with the community to maintain and evolve the definition of the [ECMAScript specification](https://www.ecma-international.org/ecma-262/).

The committee meets regularly with a neat [agenda](https://github.com/tc39/agendas), in order to plan and make decisions regarding the specification. It’s worth mentioning that the decisions are made by consensus, meaning most of the members should agree when nobody extremely disagrees or veto.

**####\*\*The T39 Process\*\***

The process of making changes in the ECMAScript specification is done by the TC39, and naturally called the [TC39 process](https://tc39.es/process-document). This process is built from five stages, starting with stage zero. Any proposal for a change in the specification goes through these stages without exception, when the committee must approve the progress from one stage to the next one.

The process was born due to the conduct of [ECMAScript 2015](http://www.ecma-international.org/ecma-262/6.0/) edition, also known as ES6, which was a pretty huge release lasting very long without delivery (actually almost 6 years). Therefore, [as of ECMAScript 2017](https://github.com/tc39/notes/blob/master/meetings/2014-01/jan-28.md#process-for-ratifying-es6), the editions have become smaller and are delivered every year, containing all proposals which are accepted at all stages of the process, since the previous edition.

**####\*\*Stages\*\***

**\*\*Stage 0 - Strawperson\*\***

This is the first [stage](https://github.com/tc39/proposals/blob/master/stage-0-proposals.md), called “Strawperson”, representing an initial idea for addition or change to the specification that isn’t considered as a formal proposal. Suggestions for this stage must come from a TC39 member or registered contributor.

**\*\*Stage 1 - Proposal\*\***

This [stage](https://github.com/tc39/proposals/blob/master/stage-1-proposals.md) is a formal proposal that describes a discrete problem or general need, suggests a shape of the solution and points out potential challenges - such as “cross-cutting” concerns with other features or complex implementation. The solution’s description should contain a high-level API with concrete examples; and also discuss algorithms, abstractions and semantics.

On top of that, one of the TC39 members is defined as the owner that is responsible to advance the proposal, and practically named champion. Typically the champion is the original author of the proposal, but not always. If the proposal meets the criteria of stage 1, and hereby representing the committee’s will of going forward with the proposal, then it moves to the draft stage.

**\*\*Stage 2 - Draft\*\***

This [stage](https://github.com/tc39/proposals#stage-2) is the **initial draft** of the proposal in the specification, phrased by the ECMAScript language.

The draft should describe the syntax, semantics and APIs precisely as much as possible, although it can have “TODO” comments or placeholders. An experimental implementation is also needed, runnable by a browser or a build-time transpiler like Babel.

Moving forward from this stage means that the committee expects that the proposal would be developed and included eventually in the official specification - when only incremental changes (and mostly fixes) are expected.

**\*\*Stage 3 - Candidate\*\***

This [stage](https://github.com/tc39/proposals#stage-3) is a candidate proposal that’s **almost final** - but ready for feedback and refinements from implementations and users. The proposal defined as completely final when there is neither further work with the specification nor external feedback.

All ECMAScript editors and designated reviewers should sign off on this specification. In addition, it should include two independent spec-compatible implementations passing the [acceptance tests](https://github.com/tc39/test262).

After this stage, changes would be made only for critical issues.

**\*\*Stage 4 - Finished\*\***

This is the last [stage](https://github.com/tc39/proposals/blob/master/finished-proposals.md), called “finished” obviously, indicating that the proposal is **ready** to be included in the latest draft of the specification - and be **delivered** with its next edition.

**\*\*Summary\*\***

The ECMAScript specification evolves every year through the TC39 process.

Key points to remember:

* TC39 is a diverse group of members working with JavaScript under ECMA International
* TC39 collaborates with the community to maintain and evolve the ECMAScript specification
* The TC39 process allows making changes in the ECMAScript specification using five regulated stages
* TC39 must approve each stage of the TC39 process for any change in the specification
* New editions of the specification are delivered every year
* Stage 0 represents an initial idea for addition or change to the specification
* Stage 1 is a formal proposal describing a problem and suggesting a proper solution
* Stage 2 is an initial draft of the proposal specification
* Stage 3 represents the draft when it’s almost final but ready for last feedback
* Stage 4 is when the proposal specification completely ready and included within the next edition

**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:

**Answer:**

* ~~Example(s) that demonstrate how to avoid the callback hell  (“Pyramid of Doom")~~
* Example(s) that demonstrate how to execute asynchronous (promise-based) code in **serial** or **parallel**

**Answer:**

Flow 1 -> day 3 ex-from-school folder -> makeFolders.js and apiFetcher.js are good examples. Ex\_3.js and ex\_2\_b can also be shown if more examples need to be shown. *(remember to read notes)*

* Example(s) that demonstrate how to implement **our own** promise-solutions.

**Answer:**

Flow 1 -> day 3 Promises.js for a simple overview/example on promises. In the folder ex-from-school there is also examples in filterdir.js *(remember to read notes)*

* Example(s) that demonstrate error handling with promises

**Answer:**

Flow 1 -> day 3 Promises.js for a simple overview/example on promises. In the folder ex-from-school there is also examples in filterdir.js *(remember to read notes)*

***return new Promise ((Resolve, reject) => -> .then().catch()***

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

**Answer:**

Provide examples to demonstrate

* Why this often is the preferred way of handling promises

**Answer:**

* Error handling with async/await

**Answer:**

Flow 1 -> day 3 ex-from-school folder -> makeFolders.js *(remember to read notes)*

* Serial or parallel execution with async/await.

**Answer:**

low 1 -> day 3 ex-from-school folder -> makeFolders.js *(remember to read notes)*

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