**Period-1 Vanilla JavaScript, es2015/15.., Node.js, Babel + Webpack and TypeScript**

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

Explain and Reflect:

* Explain the differences between Java and JavaScript. You should include both topics related to the fact that Java is a compiled language and JavaScript a scripted language, and general differences in language features.

-Java

Java skal compiles til Java byte code som så kan køres via java virtual machine.

Strongly typed language. Dvs. man f.eks skal definere datatypen på en variabel inden brug.

Variabeltyper er checked i compile time.

-JavaScript

JavaScript kan køres direkte i en browser eller vha. Node.js

Weekly typed language. Dvs. man f.eks ikke skal definere datatypen på en variabel.

* Explain the two strategies for improving JavaScript: ES6 (es2015) + ES7, versus Typescript. What does it require to use these technologies: In our backend with Node and in (many different) Browsers?

Typescript er opensource og es har en komite som beslutter hvilke features der skal tilføjes.

Vi kan allerede bruge de nye teknologier med node. Og så kan vi bruge Babel til at transpilere koden til ES5 som der er større browser support for.

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

Npm er Node.js’ package manager. Den gør at du via CLI kan downloade en masse forskellige biblioteker ned til dit node projekt.

* Explain about the Event Loop in Node.js

Når man kalder en metode fra webapis(fetch, setTimeout etc) Bliver den kørt eksternt. Når metoden er færdig ender den i en callback queue. Event loopet tjekker så hele tiden om call stacken er tom. Hvis den er tom tager den den første metode fra callback queuen.

* Explain (some) of the purposes with the tools *Babel* and *WebPack*, using examples from the exercises

Babel bruges til at transpilere nye versioner af ecmascript til ældre versioner. Dette gør at ældre browsers også kan ”forstå” koden.

Webpack bruges til at bundle moduler.

* Explain the purpose of “use strict” and Linters, exemplified with ESLint

Formålet med use strict og linters er at den vil sige at der er en syntax fejl eller hvis du f.eks bruger en variabel som ikke er blevet deklareret.

Explain using sufficient code examples the following features in JavaScript.

* Variable/function-Hoisting

Hoisting betyder at hvis vi laver en variabel af typen ”var” vil den blive løftet op til toppen af dokumentet når koden bliver kørt. Det betyder at man kan bruge variablen før man har deklareret den.

Variablen bliver ikke hoisted hvis vi bruger ”const” eller ”let”

Hvis man deklarerer en funktion således vil man kunne kalde metoden inden man har deklareret den da metoden vil blive løftet op øverst i dokumentet.

|  |
| --- |
| function name(param1, param2, ...) {  [statements] } |

Hvis man i stedet laver et function expression som denne:

|  |
| --- |
| const notHoisted = function() {  console.log('foo') } |

Vil funktionen ikke blive hoisted.

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

this i Javascript peger på funktionen den er kaldt i. Hvor this i Java peger på objektet.

* Function Closures and the JavaScript Module Pattern

#### -Closures

Et closure er når en inner function har adgang til den ydre funktions variabler.

-Module Pattern

Module pattern er et design pattern brugt til at skabe private og public indkapsling.

|  |
| --- |
| (function() {  'use strict';  // Your code here  // All function and variables are scoped to this function }()); |

* Immediately-Invoked Function Expressions (IIFE)

Dette er en metode som bliver kaldt med det samme at den bliver deklareret. Dette er syntaxen:

|  |
| --- |
| void function() {  alert("Hello from IIFE");  }(); |
|  |

* JavaScripts Prototype

Hvis man laver en klasse i js og giver den nogle fields og metoder, vil det give mening hvis hver instans af den klasse har sine egne fields, men måske hver klasse behøver ikke hver sin instans af funktionerne. Man kan derfor når man laver en klasse skrive “<klassenavn>.prototype.<metodenavn> uden for constructor funktionen og denne funktion vil være tilgængelige for alle objekter med det givne navn, der vil bare kun være en instans af disse metoder.

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

En callback er en funktion som bliver givet som argument i en anden funktion hvor den så bliver kaldt.

|  |
| --- |
| function myFilter(callback){  let arr = []  for(let i = 0; i < this.length; i++){  if(callback(this[i])){  arr.push(this[i])  }  }  return arr } |

* Explain the methods map, filter and reduce

-Map metoden laver et nyt array med resultaterne hvor man har en kaldt en funktion på hvert element af det gamle.

-Filter metoden kalder en metode på alle elementerne i et array og tilføjer dem som returnere true til et nye array.

-Reduce metoden kalder en metode på hver element af arrayet og laver det om til en enkelt værdi

* Provide examples of user-defined reusable modules implemented in Node.js

|  |
| --- |
| module.exports.MyModule = () => { console.log("My module") } |

ES6,7,8... and TypeScript

* Provide examples and explain the es2015 features: let, arrow functions, this, rest parameters, de-structuring assignments, maps/sets etc.

-let

let laver en variabel som fungere ligesom vi kender det i Java. Den bliver nemlig ikke hoisted ligesom var gør.

|  |
| --- |
| let arr = [1,2,3] |

-Arrow functions

arrow functions er en kompakt måde at skrive en funktion på. udover det har den ikke bindet til this.

|  |
| --- |
| let foo = () => {console.log("foo") } |

-Rest parameters

Gør det muligt at repræsentere et ubestemt antal af parametre som et array.

|  |
| --- |
| function sum(...theArgs) {  return theArgs.reduce((previous, current) => {  return previous + current;  }); } |

-Destructering assignments

En feature fra es6 som gør det muligt at udpakke værdier fra arrays lettere.

|  |
| --- |
| var introduction = ["Hello", "I" , "am", "Sarah"];  var [greeting, pronoun] = introduction;   console.log(greeting);//"Hello"  console.log(pronoun);//"I" |

-maps

fungere som vi kender maps fra Java.

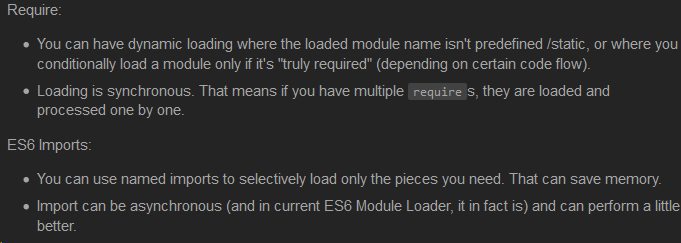
|  |
| --- |
| var map1 = new Map(); map1.set('bar', 'foo');  console.log(map1.get('bar')); // expected output: "foo" |

-Sets

Lader os opbevare værdier af enhver type.

|  |
| --- |
| const set1 = new Set([1, 2, 3, 4, 5]); |

* Explain and demonstrate how es2015 supports modules (import and export) similar to what is offered by NodeJS.

import/export måden at gøre det på fylder meget mindre og gør det lettere at lave moduler. module.export og require er en mere bloated måde at loade moduler på.   
  
Her er en sammenligning af hvordan man bruger de to forskellige metoder:  


Venstre = require vs. højre = import/export

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

Inheritance i es6 og java ligner hinanden rigtig meget, med bl.a. “extends” key word’ det egentlig bare “syntaktisk sukker” inheritance er stadig prototype baseret mens java er klassebaseret. Js har “prototype kæden” hvor at når man søger efter en egenskab på et objekt, vil det ikke kun blive søgt efter på prototypen af objektet, men også prototypen af prototypen osv. indtil man finder egenskaben eller når slutningen af kæden.

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

Du kan give statiske datatyper som bliver checket ved compile time og ikke ved runtime som almindelig Javascript.

Der er interfaces som vi kender dem i java.

|  |
| --- |
| interface IPerson {   firstName:string,   lastName:string,   sayHi: ()=>string  }   var customer:IPerson = {   firstName:"Tom",  lastName:"Hanks",   sayHi: ():string =>{return "Hi there"}  } |

En klasse i Typescript:

|  |
| --- |
| class Car {   //field   engine:string;     //constructor   constructor(engine:string) {   this.engine = engine   }    //function   disp():void {   console.log("Engine is : "+this.engine)   }  } |

En generic funktion:

|  |
| --- |
| function identity<T>(arg: T): T {  return arg; } |

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

Der er 5 stages som en feature skal igennem før den bliver godkendt. Der er forskellige krav til hver stage. TC39 kommiteen skal godkende for hver stage.

**Callbacks, Promises and async/await**

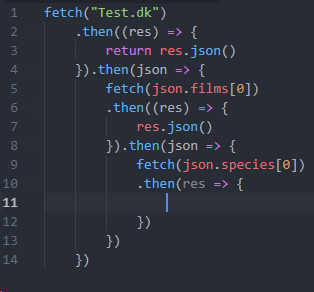
Explain about promises in ES-6 including, the problems they solve, a quick explanation of the Promise API and:

* Example(s) that demonstrate how to avoid the callback hell (“Pyramid of Doom")

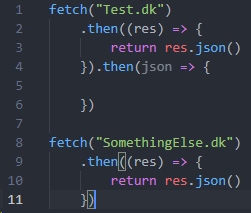
Callback hell/pyramid of doom er et navn på det der meget let kan ske når man har med a asynkron kode at gøre. Det kan f.eks. opstå når man sidder med et fetch kald og fejlhåndtere på hvert callback.  
Det er muligt at undgå callback hell med promises og .then, f.eks ved at returnere promises før man kalder næste .then: <https://stackoverflow.com/questions/40569825/i-still-get-the-pyramid-of-doom-when-using-promises-what-am-i-doing-wrong>

Derudover kan man også bruge async/await, hvor man “automatisk” vil undgå det.

* Example(s) that demonstrate how to execute asynchronous (promise-based) code in **serial** or **parallel**

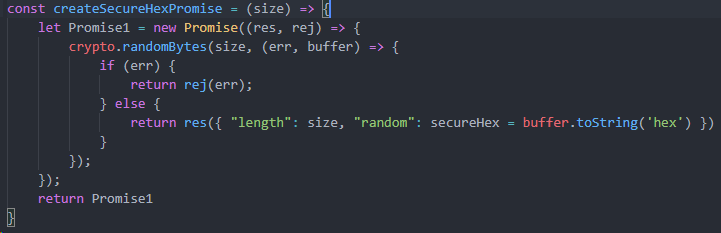
Serial eksempel:   


Parallelt eksempel:



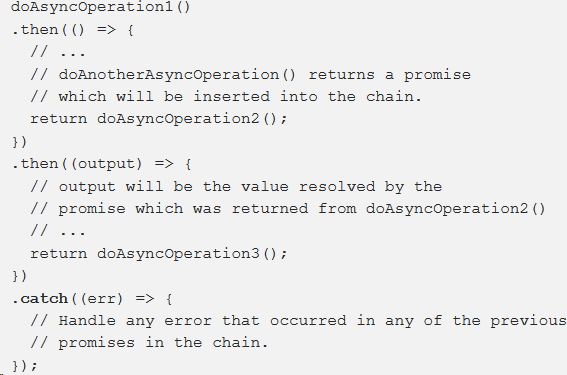
* Example(s) that demonstrate how to implement our own promise-solutions.

Her bliver der lavet et promise ud fra funktionen “crypto.randomBytes” For at implementere et promise har man to cases: rejected eller resolved, som man så implementere og derefter kan man returne et promise.



* Example(s) that demonstrate error handling with promises

Med promises kan man lave en .catch() metode som bliver kørt når man for en error:



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

Async/await er en feature i js som blev implementeret i ES6, man kan sige at det fungere som syntaktisk sukker. Den tidligere måde at håndtere asynkron kode på kan hurtigt blive uoverskuelig, mens async/await kode, bliver meget lettere at kigge på. Udover at koden bliver mere simpel at læse, er fejlhåndtering også gjort lettere, da man kan lave en try/catch blok og gribe fejl der skulle forekomme. Ved at bruge async/await bliver det derimod sværere at se at man har med asynkron kode at gøre, og det er derfor noget man skal lægge mærke til når man ser async/await.

Provide examples to demonstrate :

Why this often is the preferred way of handling promises

Alt efter hvem man er kan async/await gøre koden lettere at læse og mere overskuelig, samme tid med det kan man bruge try- catch-blokke som gør fejlhåndtering meget pænere og overskuelig end .then metoden.



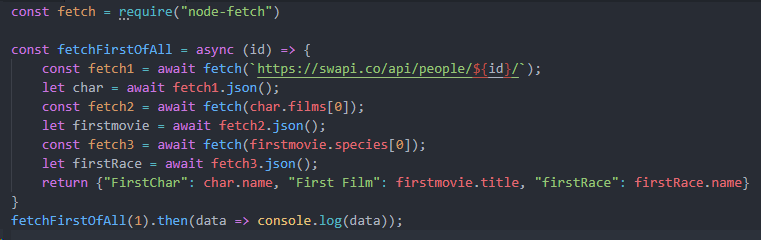
Error handling with async/await

kan gøres med en simpel try/catch blok.



Serial or parallel execution with async/await.

Sekventielt:



Sammenligning af sekventielt vs. parallelt

